

IBM WebSphere eXtreme Scale
Version 8.5

Programming Guide

April 27, 2012



This edition applies to version 8, release 5, of WebSphere eXtreme Scale and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright IBM Corporation 2009, 2012.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Contents

Figures v

Tables vii

About the *Programming Guide* ix

Chapter 1. Tutorials. 1

Tutorial: Querying a local in-memory data grid.	1
ObjectQuery tutorial - step 1	1
ObjectQuery tutorial - step 2	2
ObjectQuery tutorial - step 3	3
ObjectQuery tutorial - step 4	5
Tutorial: Storing order information in entities	9
Entity manager tutorial: Creating an entity class	9
Entity manager tutorial: Forming entity relationships	10
Entity manager tutorial: Order Entity Schema	12
Entity manager tutorial: Updating entries	15
Entity manager tutorial: Updating and removing entries with an index	16
Entity manager tutorial: Updating and removing entries by using a query	17
Tutorial: Running eXtreme Scale bundles in the OSGi framework	17
Introduction: Starting and configuring the eXtreme Scale server and container to run plug-ins in the OSGi framework	18
Module 1: Preparing to install and configure eXtreme Scale server bundles	19
Module 2: Installing and starting eXtreme Scale bundles in the OSGi framework	23
Module 3: Running the eXtreme Scale sample client	29
Module 4: Querying and upgrading the sample bundle	31

Chapter 2. Scenarios. 35

Using an OSGi environment to develop and run eXtreme Scale plug-ins	35
OSGi framework overview	35
Installing the Eclipse Equinox OSGi framework with Eclipse Gemini for clients and servers.	36
Running eXtreme Scale containers with non-dynamic plug-ins in an OSGi environment	40
Administering eXtreme Scale servers and applications in an OSGi environment.	41
Building and running eXtreme Scale dynamic plug-ins for use in an OSGi environment	42
Running eXtreme Scale containers with dynamic plug-ins in an OSGi environment	49
Using JCA to connect transactional applications to eXtreme Scale clients	58
Transaction processing in Java EE applications.	59
Installing an eXtreme Scale resource adapter	61

Configuring eXtreme Scale connection factories	63
Configuring Eclipse environments to use eXtreme Scale connection factories.	64
Configuring applications to connect with eXtreme Scale.	65
Securing J2C client connections.	65
Developing eXtreme Scale client components to use transactions	67
Administering J2C client connections.	70

Chapter 3. Getting started 73

Tutorial: Getting started with WebSphere eXtreme Scale.	73
Getting started tutorial lesson 1: Defining data grids with configuration files	73
Getting started tutorial lesson 2: Creating a client application	75
Getting started tutorial lesson 3: Running the getting started sample client application.	76
Getting started tutorial lesson 4: Monitor your environment	78
Getting started with developing applications	81

Chapter 4. Planning 83

Planning the topology	83
Local in-memory cache	83
Peer-replicated local cache	85
Embedded cache	87
Distributed cache	88
Database integration: Write-behind, in-line, and side caching	90
Planning multiple data center topologies	104
Planning to develop WebSphere eXtreme Scale applications	116
API overview	116
Plug-ins overview	117
REST data services overview	119
Spring framework overview	121
Class loader and classpath considerations	122
Relationship management	122
Cache key considerations	123
Data for different time zones	124

Chapter 5. Developing applications 125

Setting up the development environment	125
Setting up a stand-alone development environment.	125
Running a WebSphere eXtreme Scale client or server application with Apache Tomcat in Rational Application Developer	126
Running an integrated client or server application with WebSphere Application Server in Rational Application Developer	129
Accessing data with client applications.	130

Connecting to distributed ObjectGrid instances programmatically	130
Tracking map updates by an application	132
Interacting with an ObjectGrid using the ObjectGridManager interface	135
Accessing data with indexes (Index API)	142
Using Sessions to access data in the grid	145
Caching objects with no relationships involved (ObjectMap API)	152
Caching objects and their relationships (EntityManager API)	164
Retrieving entities and objects (Query API)	190
Programming for transactions	214
Configuring clients programmatically	253
Accessing data with the REST data service	255
Operations with the REST data service	255
Optimistic concurrency in the REST data service	257
Request protocols for the REST data service	258
System APIs and plug-ins	279
Managing plug-in life cycles	279
Plug-ins for multimaster replication	282
Plug-ins for versioning and comparing cache objects	284
Plug-ins for serializing cached objects	288
Plug-ins for providing event listeners	296
Plug-ins for indexing data	304
Plug-ins for communicating with databases	313
Plug-ins for managing transaction life cycle events	351
Programming to use the OSGi framework	361
Building eXtreme Scale dynamic plug-ins	361
Programming for JPA integration	365
JPA Loaders	365
Developing client-based JPA loaders	367
Example: Using the Hibernate plug-in to preload data into the ObjectGrid cache	375
Starting the JPA time-based updater	375
Developing applications with the Spring framework	379
Spring framework overview	380
Managing transactions with Spring	381
Spring managed extension beans	382
Spring extension beans and namespace support	383
Starting a container server with Spring	386
Configuring clients in the Spring framework	388
Chapter 6. Tuning performance.	391
Tuning the cache sizing agent for accurate memory consumption estimates	391
Cache memory consumption sizing	392
Tuning and performance for application development	395
Tuning the copy mode	395

Tuning evictors	404
Tuning locking performance	405
Tuning serialization performance	407
Tuning query performance	409
Tuning EntityManager interface performance	420
Chapter 7. Security	425
Configuring security profiles for the xscmd utility	425
Securing J2C client connections	426
Programming for security	427
Security API	427
Client authentication programming	429
Client authorization programming	446
Data grid authentication	454
Local security programming	454
Chapter 8. Troubleshooting	461
Troubleshooting and support for WebSphere eXtreme Scale	461
Techniques for troubleshooting problems	461
Searching knowledge bases	463
Getting fixes	464
Contacting IBM Support	465
Exchanging information with IBM	466
Subscribing to Support updates	467
Enabling logging	468
Collecting trace	469
Trace options	470
Analyzing log and trace data	472
Log analysis overview	473
Running log analysis	473
Creating custom scanners for log analysis	475
Troubleshooting log analysis	476
Troubleshooting client connectivity	477
Troubleshooting cache integration	478
Troubleshooting the JPA cache plug-in	479
Troubleshooting IBM eXtremeMemory and IBM eXtremeIO	479
Troubleshooting administration	480
Troubleshooting multiple data center configurations	481
Troubleshooting loaders	481
Troubleshooting deadlocks	483
IBM Support Assistant for WebSphere eXtreme Scale	488
Notices	491
Trademarks	493
Index	495

Figures

1. Order Schema	6	19. Loader	97
2. Order Entity Schema	12	20. Loader plug-in	99
3. Eclipse Equinox process for including all configuration and metadata in an OSGi bundle	52	21. Client loader	100
4. Eclipse Equinox process for specify configuration and metadata outside of an OSGi bundle	53	22. Periodic refresh	101
5. Local in-memory cache scenario.	84	23. Microsoft WCF Data Services	119
6. Peer-replicated cache with changes that are propagated with JMS	85	24. WebSphere eXtreme Scale REST data service	120
7. Peer-replicated cache with changes that are propagated with the high availability manager.	86	25. The interaction of the query with the ObjectGrid object maps and how a schema is defined for classes and associated with an ObjectGrid map.	196
8. Embedded cache	87	26. The interaction of the query with the ObjectGrid object maps and how the entity schema is defined and associated with an ObjectGrid map.	201
9. Distributed cache	89	27. Loader	314
10. Near cache	89	28. Write-behind caching	331
11. ObjectGrid as a database buffer	91	29. JPA Loader architecture	366
12. ObjectGrid as a side cache	91	30. Client loader that uses JPA implementation to load the ObjectGrid	368
13. Side cache	92	31. Periodic refresh	379
14. In-line cache	93	32. Flow of client authentication and authorization	428
15. Read-through caching	94		
16. Write-through caching	94		
17. Write-behind caching	95		
18. Write-behind caching	96		

Tables

1. Custom properties for configuring connection factories.	63	13. List of methods and the required ObjectGridPermission.	449
2. Arbitration approaches	112	14. Permissions to a server-hosted ObjectMap	449
3. Other methods.	193	15. Single key deadlocks scenario	484
4. Key to BNF summary.	212	16. Single key deadlocks, continued	485
5. Lock mode compatibility matrix	235	17. Single key deadlocks, continued	485
6. Support for range index	308	18. Single key deadlocks, continued	485
7. Status value and response	326	19. Ordered multiple key deadlock scenario	486
8. Commit sequence on the primary.	327	20. Ordered multiple key deadlock scenario, continued.	486
9. Synchronous commit processing	328	21. Out of order with U lock scenario.	487
10. Some write-behind options	330		
11. Client loader modes	369		
12. List of methods and the required MapPermission	448		

About the *Programming Guide*

The WebSphere® eXtreme Scale documentation set includes three volumes that provide the information necessary to use, program for, and administer the WebSphere eXtreme Scale product.

WebSphere eXtreme Scale library

The WebSphere eXtreme Scale library contains the following books:

- The *Product Overview* contains a high-level view of WebSphere eXtreme Scale concepts, including use case scenarios, and tutorials.
- The *Installation Guide* describes how to install common topologies of WebSphere eXtreme Scale.
- The *Administration Guide* contains the information necessary for system administrators, including how to plan application deployments, plan for capacity, install and configure the product, start and stop servers, monitor the environment, and secure the environment.
- The *Programming Guide* contains information for application developers on how to develop applications for WebSphere eXtreme Scale using the included API information.

To download the books, go to the WebSphere eXtreme Scale library page.

You can also access the same information in this library in the WebSphere eXtreme Scale Version 8.5 information center..

Using the books offline

All of the books in the WebSphere eXtreme Scale library contain links to the information center, with the following root URL: <http://publib.boulder.ibm.com/infocenter/wxsinfo/v8r5>. These links take you directly to related information. However, if you are working offline and encounter one of these links, you can search for the title of the link in the other books in the library. The API documentation, glossary, and messages reference are not available in PDF books.

Who should use this book

This book is intended primarily for application developers.

Getting updates to this book

You can get updates to this book by downloading the most recent version from the WebSphere eXtreme Scale library page.

How to send your comments

Contact the documentation team. Did you find what you needed? Was it accurate and complete? Send your comments about this documentation by e-mail to wasdoc@us.ibm.com.

Chapter 1. Tutorials



You can use tutorials to help you understand product usage scenarios, including entity manager, queries, and security.

Tutorial: Querying a local in-memory data grid

You can develop a local in-memory ObjectGrid that can store order information for a website, and use the ObjectQuery API to query the data grid.

Before you begin

Be sure to have `objectgrid.jar` file in the classpath.

About this task

Each step in the tutorial builds on the previous step. Follow each of the steps to build a simple Java Platform, Standard Edition Version 5 or later application that uses an in-memory, local data grid.

ObjectQuery tutorial - step 1

With the following steps, you can continue to develop a local, in-memory ObjectGrid that stores order information for an online retail store using the ObjectMap APIs. You define a schema for the map and run a query against the map.

Procedure

1. Create an ObjectGrid with a map schema.

Create an ObjectGrid with one map schema for the map, then insert an object into the cache and later retrieve it using a simple query.

OrderBean.java

```
public class OrderBean implements Serializable {
    String orderNumber;
    java.util.Date date;
    String customerName;
    String itemName;
    int quantity;
    double price;
}
```

2. Define the primary key.

The previous code shows an OrderBean object. This object implements the `java.io.Serializable` interface because all objects in the cache must (by default) be Serializable.

The `orderNumber` attribute is the primary key of the object. The following example program can be run in stand-alone mode. You should follow this tutorial in an Eclipse Java project that has the `objectgrid.jar` file added to the class path.

Application.java

```

package querytutorial.basic.step1;

import java.util.Iterator;

import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.config.QueryConfig;
import com.ibm.websphere.objectgrid.config.QueryMapping;
import com.ibm.websphere.objectgrid.query.ObjectQuery;

public class Application
{
    static public void main(String [] args) throws Exception
    {
        ObjectGrid og = ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.defineMap("Order");

        // Define the schema
        QueryConfig queryCfg = new QueryConfig();
        queryCfg.addQueryMapping(new QueryMapping("Order", OrderBean.class.getName(),
"orderNumber", QueryMapping.FIELD_ACCESS));
        og.setQueryConfig(queryCfg);

        Session s = og.getSession();
        ObjectMap orderMap = s.getMap("Order");

        s.begin();
        OrderBean o = new OrderBean();
        o.customerName = "John Smith";
        o.date = new java.util.Date(System.currentTimeMillis());
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;
        orderMap.put(o.orderNumber, o);
        s.commit();

        s.begin();
        ObjectQuery query = s.createObjectQuery("SELECT o FROM Order o WHERE o.itemName='Widget'");
        Iterator result = query.getResultIterator();
        o = (OrderBean) result.next();
        System.out.println("Found order for customer: " + o.customerName);
        s.commit();
        // Close the session (optional in Version 7.1.1 and later) for improved performance
        s.close();
    }
}

```

This eXtreme Scale application first initializes a local ObjectGrid with an automatically generated name. Next, the application creates a BackingMap and a QueryConfig that defines what Java type is associated with the map, the name of the field that is the primary key for the map, and how to access the data in the object. You then obtain a Session to get the ObjectMap instance and insert an OrderBean object into the map in a transaction.

After the data is committed into the cache, you can use ObjectQuery to find the OrderBean using any of the persistent fields in the class. Persistent fields are those that do not have the transient modifier. Because you did not define any indexes on the BackingMap, ObjectQuery must scan each object in the map using Java reflection.

What to do next

“ObjectQuery tutorial - step 2” demonstrates how an index can be used to optimize the query.

ObjectQuery tutorial - step 2

With the following steps, you can continue to create an ObjectGrid with one map and an index, along with a schema for the map. Then you can insert an object into the cache and later retrieve it using a simple query.

Before you begin

Be sure that you have completed “ObjectQuery tutorial - step 1” on page 1 before proceeding with this step of the tutorial.

Procedure

Schema and index

Application.java

```
// Create an index
  HashIndex idx= new HashIndex();
  idx.setName("theItemName");
  idx.setAttributeName("itemName");
  idx.setRangeIndex(true);
  idx.setFieldAccessAttribute(true);
  orderBMap.addMapIndexPlugin(idx);
}
```

The index must be a `com.ibm.websphere.objectgrid.plugins.index.HashIndex` instance with the following settings:

- The Name is arbitrary, but must be unique for a given BackingMap.
- The AttributeName is the name of the field or bean property which the indexing engine uses to introspect the class. In this case, it is the name of the field for which you will create an index.
- RangeIndex must always be true.
- FieldAccessAttribute should match the value set in the QueryMapping object when the query schema was created. In this case, the Java object is accessed using the fields directly.

When a query runs that filters on the `itemName` field, the query engine automatically uses the defined index. Using the index allows the query to run much faster and a map scan is not needed. The next step demonstrates how an index can be used to optimize the query.

Next step

ObjectQuery tutorial - step 3

With the following step, you can create an ObjectGrid with two maps and a schema for the maps with a relationship, then insert objects into the cache and later retrieve them using a simple query.

Before you begin

Be sure you have completed “ObjectQuery tutorial - step 2” on page 2 prior to proceeding with this step.

About this task

In this example, there are two maps, each with a single Java type mapped to it. The Order map has OrderBean objects and the Customer map has CustomerBean objects in it.

Procedure

Define maps with a relationship.

OrderBean.java

```
public class OrderBean implements Serializable {
    String orderNumber;
    java.util.Date date;
    String customerId;
    String itemName;
    int quantity;
    double price;
}
```

The OrderBean no longer has the customerName in it. Instead, it has the customerId, which is the primary key for the CustomerBean object and the Customer map.

CustomerBean.java

```
public class CustomerBean implements Serializable{
    private static final long serialVersionUID = 1L;
    String id;
    String firstName;
    String surname;
    String address;
    String phoneNumber;
}
```

The relationship between the two types or Maps follows:

Application.java

```
public class Application
{
    static public void main(String [] args)
        throws Exception
    {
        ObjectGrid og = ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.defineMap("Order");
        og.defineMap("Customer");

        // Define the schema
        QueryConfig queryCfg = new QueryConfig();
        queryCfg.addQueryMapping(new QueryMapping(
            "Order", OrderBean.class.getName(), "orderNumber", QueryMapping.FIELD_ACCESS));
        queryCfg.addQueryMapping(new QueryMapping(
            "Customer", CustomerBean.class.getName(), "id", QueryMapping.FIELD_ACCESS));
        queryCfg.addQueryRelationship(new QueryRelationship(
            OrderBean.class.getName(), CustomerBean.class.getName(), "customerId", null));
        og.setQueryConfig(queryCfg);

        Session s = og.getSession();
        ObjectMap orderMap = s.getMap("Order");
        ObjectMap custMap = s.getMap("Customer");

        s.begin();
        CustomerBean cust = new CustomerBean();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";
        custMap.insert(cust.id, cust);

        OrderBean o = new OrderBean();
        o.customerId = cust.id;
        o.date = new java.util.Date();
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;
        orderMap.insert(o.orderNumber, o);
        s.commit();

        s.begin();
        ObjectQuery query = s.createObjectQuery(
            "SELECT c FROM Order o JOIN o.customerId as c WHERE o.itemName='Widget'");
        Iterator result = query.getResultIterator();
        cust = (CustomerBean) result.next();
    }
}
```

```

        System.out.println("Found order for customer: " + cust.firstName + " " + cust.surname);
    s.commit();
    // Close the session (optional in Version 7.1.1 and later) for improved performance
    s.close();
    }
}

```

The equivalent XML in the ObjectGrid deployment descriptor follows:

```

<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="CompanyGrid">
      <backingMap name="Order"/>
      <backingMap name="Customer"/>

      <querySchema>
        <mapSchemas>
          <mapSchema
            mapName="Order"
            valueClass="com.mycompany.OrderBean"
            primaryKeyField="orderNumber"
            accessType="FIELD"/>
          <mapSchema
            mapName="Customer"
            valueClass="com.mycompany.CustomerBean"
            primaryKeyField="id"
            accessType="FIELD"/>
        </mapSchemas>
        <relationships>
          <relationship
            source="com.mycompany.OrderBean"
            target="com.mycompany.CustomerBean"
            relationField="customerId"/>
        </relationships>
      </querySchema>
    </objectGrid>
  </objectGrids>
</objectGridConfig>

```

What to do next

“ObjectQuery tutorial - step 4,” expands the current step by including field and property access objects and additional relationships.

ObjectQuery tutorial - step 4

The following step shows how to create an ObjectGrid with four maps and a schema for the maps. Some of the maps maintain a one-to-one (unidirectional) and one-to-many (bidirectional) relationship. After creating the maps, you can then run the sample `Application.java` program to insert objects into the cache and run queries to retrieve these objects.

Before you begin

Be sure to have completed “ObjectQuery tutorial - step 3” on page 3 prior to continuing with the current step.

About this task

You are required to create four JAVA classes. These are the maps for the ObjectGrid:

- OrderBean.java
- OrderLineBean.java
- CustomerBean.java
- ItemBean.java

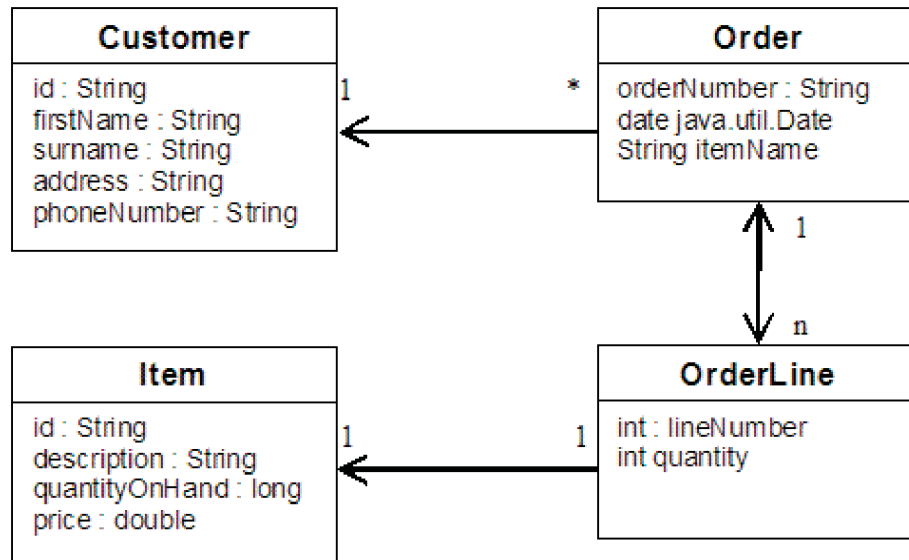


Figure 1. Order Schema. An Order schema has a one-to-one relationship with Customer and a one-to-many relationship with OrderLine. The OrderLine map has a one-to-one relationship with Item and includes the quantity ordered.

After creating these JAVA classes with these relationships, you can then run the sample Application.java program. This program lets you insert objects into the cache and retrieve these using several queries.

Procedure

1. Create the following JAVA classes:

OrderBean.java

```

public class OrderBean implements Serializable {
    String orderNumber;
    java.util.Date date;
    String customerId;
    String itemName;
    List<Integer> orderLines;
}
  
```

OrderLineBean.java

```

public class OrderLineBean implements Serializable {
    int lineNumber;
    int quantity;
    String orderNumber;
    String itemId;
}
  
```

CustomerBean.java

```

public class CustomerBean implements Serializable{
    String id;
    String firstName;
}
  
```



```

        String surname;
        String address;
        String phoneNumber;
    }

```

ItemBean.java

```

public class ItemBean implements Serializable {
    String id;
    String description;
    long quantityOnHand;
    double price;
}

```

2. After creating the classes, you can run the sample Application.java:

Application.java

```

public class Application static public void main(String [] args)throws Exception
    // Configure programatically
    ObjectGrid og = ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
    og.defineMap("Order");
    og.defineMap("Customer");
    og.defineMap("OrderLine");
    og.defineMap("Item");

    // Define the schema
    QueryConfig queryCfg = new QueryConfig();
    queryCfg.addQueryMapping(new QueryMapping("Order", OrderBean.class.getName(), "orderNumber", QueryMapping.FIELD_ACCESS));
    queryCfg.addQueryMapping(new QueryMapping("Customer", CustomerBean.class.getName(), "id", QueryMapping.FIELD_ACCESS));
    queryCfg.addQueryMapping(new QueryMapping("OrderLine", OrderLineBean.class.getName(), "lineNumber", QueryMapping.FIELD_ACCESS));
    queryCfg.addQueryMapping(new QueryMapping("Item", ItemBean.class.getName(), "id", QueryMapping.FIELD_ACCESS));
    queryCfg.addQueryRelationship(new QueryRelationship(OrderBean.class.getName(), CustomerBean.class.getName(), "customerId", null));
    queryCfg.addQueryRelationship(new QueryRelationship(OrderBean.class.getName(), OrderLineBean.class.getName(), "orderLines", "lineNumber"));
    queryCfg.addQueryRelationship(new QueryRelationship(OrderLineBean.class.getName(), ItemBean.class.getName(), "itemId", null));
    og.setQueryConfig(queryCfg);

    // Get session and maps;
    Session s = og.getSession();
    ObjectMap orderMap = s.getMap("Order");
    ObjectMap custMap = s.getMap("Customer");
    ObjectMap itemMap = s.getMap("Item");
    ObjectMap orderLineMap = s.getMap("OrderLine");

    // Add data
    s.begin();
    CustomerBean aCustomer = new CustomerBean();
    aCustomer.address = "Main Street";
    aCustomer.firstName = "John";
    aCustomer.surname = "Smith";
    aCustomer.id = "C001";
    aCustomer.phoneNumber = "5555551212";
    custMap.insert(aCustomer.id, aCustomer);

    // Insert an order with a reference to the customer, but without any OrderLines yet.
    // Because we are using CopyMode.COPY_ON_READ_AND_COMMIT, the
    // insert won't be copied into the backing map until commit time, so
    // the reference is still good.

    OrderBean anOrder = new OrderBean();
    anOrder.customerId = aCustomer.id;
    anOrder.date = new java.util.Date();
    anOrder.itemName = "Widget";
    anOrder.orderNumber = "1";
    anOrder.orderLines = new ArrayList();
    orderMap.insert(anOrder.orderNumber, anOrder);

    ItemBean anItem = new ItemBean();
    anItem.id = "AC0001";
    anItem.description = "Description of widget";
    anItem.quantityOnHand = 100;
    anItem.price = 1000.0;
    itemMap.insert(anItem.id, anItem);

    // Create the OrderLines and add the reference to the Order
    OrderLineBean anOrderLine = new OrderLineBean();
    anOrderLine.lineNumber = 99;
    anOrderLine.itemId = anItem.id;
    anOrderLine.orderNumber = anOrder.orderNumber;
    anOrderLine.quantity = 500;
    orderLineMap.insert(anOrderLine.lineNumber, anOrderLine);
    anOrder.orderLines.add(Integer.valueOf(anOrderLine.lineNumber));

    anOrderLine = new OrderLineBean();
    anOrderLine.lineNumber = 100;

```

```

        anOrderLine.itemId = anItem.id;
        anOrderLine.orderNumber = anOrder.orderNumber;
        anOrderLine.quantity = 501;
        orderLineMap.insert(anOrderLine.lineNumber, anOrderLine);
        anOrder.orderLines.add(Integer.valueOf(anOrderLine.lineNumber));
        s.commit();

    s.begin();
    // Find all customers who have ordered a specific item.
    ObjectQuery query = s.createObjectQuery("SELECT c FROM Order o JOIN o.customerId as c WHERE o.itemName='Widget'");
    Iterator result = query.getResultIterator();
    aCustomer = (CustomerBean) result.next();
    System.out.println("Found order for customer: " + aCustomer.firstName + " " + aCustomer.surname);
    s.commit();

    s.begin();
    // Find all OrderLines for customer C001.
    // The query joins are expressed on the foreign keys.
    query = s.createObjectQuery("SELECT ol FROM Order o JOIN o.customerId as c JOIN o.orderLines as ol WHERE c.id='C001'");
    result = query.getResultIterator();
    System.out.println("Found OrderLines:");
    while(result.hasNext()) {
        anOrderLine = (OrderLineBean) result.next();
        System.out.println(anOrderLine.lineNumber + ", qty=" + anOrderLine.quantity);
    }
    // Close the session (optional in Version 7.1.1 and later) for improved performance
    s.close();
}
}

```

3. Using the XML configuration below (in the ObjectGrid deployment descriptor) is equivalent to the programmatic approach above.

```

<?xml version="1.0" encoding="UTF-8"?><objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://ibm.com/ws/objectgrid/config
../objectGrid.xsd" xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
  <objectGrid name="CompanyGrid">
    <backingMap name="Order"/>
    <backingMap name="Customer"/>
    <backingMap name="OrderLine"/>
    <backingMap name="Item"/>
  </objectGrid>
</objectGrids>

<querySchema>
<mapSchemas>
  <mapSchema
    mapName="Order"
    valueClass="com.mycompany.OrderBean"
    primaryKeyField="orderNumber"
    accessType="FIELD"/>
  <mapSchema
    mapName="Customer"
    valueClass="com.mycompany.CustomerBean"
    primaryKeyField="id"
    accessType="FIELD"/>
  <mapSchema
    mapName="OrderLine"
    valueClass="com.mycompany.OrderLineBean"
    primaryKeyField="
    lineNumber"
    accessType="FIELD"/>
  <mapSchema
    mapName="Item"
    valueClass="com.mycompany.ItemBean"
    primaryKeyField="id"
    accessType="FIELD"/>
</mapSchemas>

<relationships>
<relationship
  source="com.mycompany.OrderBean"
  target="com.mycompany.CustomerBean"
  relationField="customerId"/>
<relationship
  source="com.mycompany.OrderBean"
  target="com.mycompany.OrderLineBean"
  relationField="orderLines"
  invRelationField="lineNumber"/>
<relationship
  source="com.mycompany.OrderLineBean"
  target="com.mycompany.ItemBean"
  relationField="itemId"/>
</relationships>
</querySchema>
</objectGrid>
</objectGrids>
</objectGridConfig>

```

Tutorial: Storing order information in entities

The tutorial for the entity manager shows you how to use WebSphere eXtreme Scale to store order information on a Web site. You can create a simple Java Platform, Standard Edition 5 application that uses an in-memory, local eXtreme Scale. The entities use Java SE 5 annotations and generics.

Before you begin

Ensure that you have met the following requirements before you begin the tutorial:

- You must have Java SE 5.
- You must have the `objectgrid.jar` file in your classpath.

Entity manager tutorial: Creating an entity class

Create a local ObjectGrid with one entity by creating an Entity class, registering the entity type, and storing an entity instance into the cache.

Procedure

1. Create the Order object. To identify the object as an ObjectGrid entity, add the `@Entity` annotation. When you add this annotation, all serializable attributes in the object are automatically persisted in eXtreme Scale, unless you use annotations on the attributes to override the attributes. The `orderNumber` attribute is annotated with `@Id` to indicate that this attribute is the primary key. An example of an Order object follows:

Order.java

```
@Entity
public class Order
{
    @Id String orderNumber;
    Date date;
    String customerName;
    String itemName;
    int quantity;
    double price;
}
```

2. Run the eXtreme Scale Hello World application to demonstrate the entity operations. The following example program can be issued in stand-alone mode to demonstrate the entity operations. Use this program in an Eclipse Java project that has the `objectgrid.jar` file added to the class path. An example of a simple Hello world application that uses eXtreme Scale follows:

Application.java

```
package emtutorial.basic.step1;

import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.em.EntityManager;

public class Application
{
    static public void main(String [] args)
        throws Exception
    {
        ObjectGrid og =
        ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.registerEntities(new Class[] {Order.class});

        Session s = og.getSession();
        EntityManager em = s.getEntityManager();

        em.getTransaction().begin();

        Order o = new Order();
```

```

        o.customerName = "John Smith";
        o.date = new java.util.Date(System.currentTimeMillis());
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;

        em.persist(o);
        em.getTransaction().commit();

        em.getTransaction().begin();
        o = (Order)em.find(Order.class, "1");
        System.out.println("Found order for customer: " + o.customerName);
        em.getTransaction().commit();
    }
}

```

This example application performs the following operations:

- a. Initializes a local eXtreme Scale with an automatically generated name.
- b. Registers the entity classes with the application by using the registerEntities API, although using the registerEntities API is not always necessary.
- c. Retrieves a Session and a reference to the entity manager for the Session.
- d. Associates each eXtreme Scale Session with a single EntityManager and EntityTransaction. The EntityManager is now used.
- e. The registerEntities method creates a BackingMap object that is called Order, and associates the metadata for the Order object with the BackingMap object. This metadata includes the key and non-key attributes, along with the attribute types and names.
- f. A transaction starts and creates an Order instance. The transaction is populated with some values. The transaction is then persisted by using the EntityManager.persist method, which identifies the entity as waiting to be included in the associated map.
- g. The transaction is then committed, and the entity is included in the ObjectMap instance.
- h. Another transaction is made, and the Order object is retrieved by using the key 1. The type cast on the EntityManager.find method is necessary. The Java SE 5 capability is not used to ensure that the objectgrid.jar file works on a Java SE Version 5 and later Java virtual machine.

Entity manager tutorial: Forming entity relationships

Create a simple relationship between entities by creating two entity classes with a relationship, registering the entities with the ObjectGrid, and storing the entity instances into the cache.

Procedure

1. Create the customer entity, which is used to store customer details independently from the Order object. An example of the customer entity follows:

```

Customer.java
@Entity
public class Customer
{
    @Id String id;
    String firstName;
    String surname;
    String address;
    String phoneNumber;
}

```

This class includes information about the customer such as name, address, and phone number.

2. Create the Order object, which is similar to the Order object in the “Entity manager tutorial: Creating an entity class” on page 9 topic. An example of the order object follows:

Order.java

```
@Entity
public class Order
{
    @Id String orderNumber;
    Date date;
    @ManyToOne(cascade=CascadeType.PERSIST) Customer customer;
    String itemName;
    int quantity;
    double price;
}
```

In this example, a reference to a Customer object replaces the customerName attribute. The reference has an annotation that indicates a many-to-one relationship. A many-to-one relationship indicates that each order has one customer, but multiple orders might reference the same customer. The cascade annotation modifier indicates that if the entity manager persists the Order object, it must also persist the Customer object. If you choose to not set the cascade persist option, which is the default option, you must manually persist the Customer object with the Order object.

3. Using the entities, define the maps for the ObjectGrid instance. Each map is defined for a specific entity, and one entity is named Order and the other is named Customer. The following example application illustrates how to store and retrieve a customer order:

Application.java

```
public class Application
{
    static public void main(String [] args)
        throws Exception
    {
        ObjectGrid og =
        ObjectGridManagerFactory.getObjectGridManager().createObjectGrid();
        og.registerEntities(new Class[] {Order.class});

        Session s = og.getSession();
        EntityManager em = s.getEntityManager();

        em.getTransaction().begin();

        Customer cust = new Customer();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";

        Order o = new Order();
        o.customer = cust;
        o.date = new java.util.Date();
        o.itemName = "Widget";
        o.orderNumber = "1";
        o.price = 99.99;
        o.quantity = 1;

        em.persist(o);
        em.getTransaction().commit();

        em.getTransaction().begin();
        o = (Order)em.find(Order.class, "1");
        System.out.println("Found order for customer: "
        + o.customer.firstName + " " + o.customer.surname);
        em.getTransaction().commit();
        // Close the session (optional in Version 7.1.1 and later) for improved performance
        s.close();
    }
}
```

This application is similar to the example application that is in the previous step. In the preceding example, only a single class Order is registered. WebSphere eXtreme Scale detects and automatically includes the reference to the Customer entity, and a Customer instance for John Smith is created and referenced from the new Order object. As a result, the new customer is automatically persisted, because the relationship between two orders includes the cascade modifier, which requires that each object be persisted. When the Order object is found, the entity manager automatically finds the associated Customer object and inserts a reference to the object.

Entity manager tutorial: Order Entity Schema

Create four entity classes by using both single and bidirectional relationships, ordered lists, and foreign key relationships. The EntityManager APIs are used to persist and find the entities. Building on the Order and Customer entities that are in the previous parts of the tutorial, this tutorial step adds two more entities: the Item and OrderLine entities.

About this task

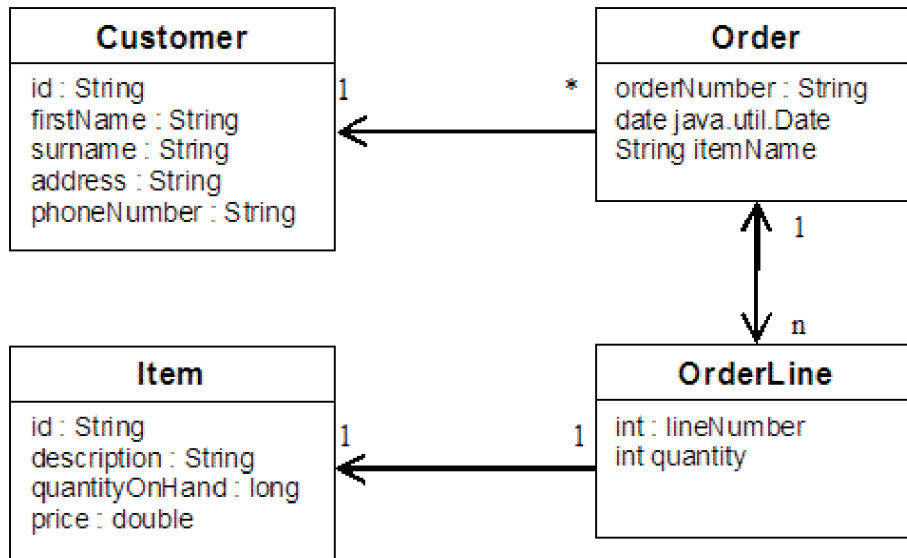


Figure 2. Order Entity Schema. An Order entity has a reference to one customer and zero or more OrderLines. Each OrderLine entity has a reference to a single item and includes the quantity ordered.

Procedure

1. Create the customer entity, which is similar to the previous examples.

```

Customer.java
@Entity
public class Customer
{
    @Id String id;
    String firstName;
    String surname;
    String address;
    String phoneNumber;
}
  
```

2. Create the Item entity, which holds information about a product that is included in the store's inventory, such as the product description, quantity, and price.

```

Item.java
@Entity
public class Item
{
    @Id String id;
    String description;
    long quantityOnHand;
    double price;
}

```

3. Create the OrderLine entity. Each Order has zero or more OrderLines, which identify the quantity of each item in the order. The key for the OrderLine is a compound key that consists of the Order that owns the OrderLine and an integer that assigns the order line a number. Add the cascade persist modifier to every relationship on your entities.

```

OrderLine.java
@Entity
public class OrderLine
{
    @Id @ManyToOne(cascade=CascadeType.PERSIST) Order order;
    @Id int lineNumber;
    @OneToOne(cascade=CascadeType.PERSIST) Item item;
    int quantity;
    double price;
}

```

4. Create the final Order Object, which has a reference to the Customer for the order and a collection of OrderLine objects.

```

Order.java
@Entity
public class Order
{
    @Id String orderNumber;
    java.util.Date date;
    @ManyToOne(cascade=CascadeType.PERSIST) Customer customer;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="order")
    @OrderBy("lineNumber") List<OrderLine> lines;
}

```

The cascade ALL is used as the modifier for lines. This modifier signals the EntityManager to cascade both the PERSIST operation and the REMOVE operation. For example, if the Order entity is persisted or removed, then all OrderLine entities are also persisted or removed.

If an OrderLine entity is removed from the lines list in the Order object, the reference is then broken. However, the OrderLine entity is not removed from the cache. You must use the EntityManager remove API to remove entities from the cache. The REMOVE operation is not used on the customer entity or the item entity from OrderLine. As a result, the customer entity remains even though the order or item is removed when the OrderLine is removed.

The mappedBy modifier indicates an inverse relationship with the target entity. The modifier identifies which attribute in the target entity references the source entity, and the owning side of a one-to-one or many-to-many relationship. Typically, you can omit the modifier. However, an error is displayed to indicate that it must be specified if WebSphere eXtreme Scale cannot discover it automatically. An OrderLine entity that contains two of type Order attributes in a many-to-one relationship typically causes the error.

The @OrderBy annotation specifies the order in which each OrderLine entity should be in the lines list. If the annotation is not specified, then the lines display in an arbitrary order. Although the lines are added to the Order entity by issuing ArrayList, which preserves the order, the EntityManager does not necessarily recognize the order. When you issue the find method to retrieve the Order object from the cache, the list object is not an ArrayList object.

5. Create the application. The following example illustrates the final Order object, which has a reference to the Customer for the order and a collection of OrderLine objects.
 - a. Find the Items to order, which then become Managed entities.
 - b. Create the OrderLine and attach it to each Item.
 - c. Create the Order and associate it with each OrderLine and the customer.
 - d. Persist the order, which automatically persists each OrderLine.
 - e. Commit the transaction, which detaches each entity and synchronizes the state of the entities with the cache.
 - f. Print the order information. The OrderLine entities are automatically sorted by the OrderLine ID.

Application.java

```

static public void main(String [] args)
    throws Exception
    {
        ...

        // Add some items to our inventory.
        em.getTransaction().begin();
        createItems(em);
        em.getTransaction().commit();

        // Create a new customer with the items in his cart.
        em.getTransaction().begin();
        Customer cust = createCustomer();
        em.persist(cust);

        // Create a new order and add an order line for each item.
        // Each line item is automatically persisted since the
        // Cascade=ALL option is set.
        Order order = createOrderFromItems(em, cust, "ORDER_1",
        new String[]{"1", "2"}, new int[]{1,3});
        em.persist(order);
        em.getTransaction().commit();

        // Print the order summary
        em.getTransaction().begin();
        order = (Order)em.find(Order.class, "ORDER_1");
        System.out.println(printOrderSummary(order));
        em.getTransaction().commit();
    }

    public static Customer createCustomer() {
        Customer cust = new Customer();
        cust.address = "Main Street";
        cust.firstName = "John";
        cust.surname = "Smith";
        cust.id = "C001";
        cust.phoneNumber = "5555551212";
        return cust;
    }

    public static void createItems(EntityManager em) {
        Item item1 = new Item();
        item1.id = "1";
        item1.price = 9.99;
        item1.description = "Widget 1";
        item1.quantityOnHand = 4000;
        em.persist(item1);

        Item item2 = new Item();
        item2.id = "2";
    }

```



```

        item2.price = 15.99;
        item2.description = "Widget 2";
        item2.quantityOnHand = 225;
        em.persist(item2);
    }

    public static Order createOrderFromItems(EntityManager em,
        Customer cust, String orderId, String[] itemIds, int[] qty) {

        Item[] items =.getItems(em, itemIds);

        Order order = new Order();
        order.customer = cust;
        order.date = new java.util.Date();
        order.orderNumber = orderId;
        order.lines = new ArrayList<OrderLine>(items.length);
        for(int i=0;i<items.length;i++){
            OrderLine line = new OrderLine();
            line.lineNumber = i+1;
            line.item = items[i];
            line.price = line.item.price;
            line.quantity = qty[i];
            line.order = order;
            order.lines.add(line);
        }
        return order;
    }

    public static Item[] getItems(EntityManager em, String[] itemIds) {
        Item[] items = new Item[itemIds.length];
        for(int i=0;i<items.length;i++){
            items[i] = (Item) em.find(Item.class, itemIds[i]);
        }
        return items;
    }
}

```

The next step is to delete an entity. The EntityManager interface has a remove method that marks an object as deleted. The application should remove the entity from any relationship collections before calling the remove method. Edit the references and issue the remove method, or `em.remove(object)`, as a final step.

Entity manager tutorial: Updating entries

If you want to change an entity, you can find the instance, update the instance and any referenced entities, and commit the transaction.

Procedure

Update entries. The following example demonstrates how to find the Order instance, change it and any referenced entities, and commit the transaction.

```

public static void updateCustomerOrder(EntityManager em) {
    em.getTransaction().begin();
    Order order = (Order) em.find(Order.class, "ORDER_1");
    processDiscount(order, 10);
    Customer cust = order.customer;
    cust.phoneNumber = "5075551234";
    em.getTransaction().commit();
}

public static void processDiscount(Order order, double discountPct) {
    for(OrderLine line : order.lines) {
        line.price = line.price * ((100-discountPct)/100);
    }
}

```

Flushing the transaction synchronizes all managed entities with the cache. When a transaction is committed, a flush automatically occurs. In this case, the Order becomes a managed entity. Any entities that are referenced from the Order, Customer, and OrderLine also become managed entities. When the transaction is flushed, each of the entities are checked to determine if they have been modified. Those that are modified are updated in the cache. After the transaction completes, by either being committed or rolled back, the entities become detached and any changes that are made in the entities are not reflected in the cache.

Entity manager tutorial: Updating and removing entries with an index

You can use an index to find, update, and remove entities.

Procedure

Update and remove entities by using an index. Use an index to find, update, and remove entities. In the following examples, the Order entity class is updated to use the @Index annotation. The @Index annotation signals WebSphere eXtreme Scale to create a range index for an attribute. The name of the index is the same name as the name of the attribute and is always a MapRangeIndex index type.

Order.java

```
@Entity
public class Order
{
    @Id String orderNumber;
    @Index java.util.Date date;
    @OneToOne(cascade=CascadeType.PERSIST) Customer customer;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="order")
    @OrderBy("lineNumber") List<OrderLine> lines; }
}
```

The following example demonstrates how to cancel all orders that are submitted within the last minute. Find the order by using an index, add the items in the order back into the inventory, and remove the order and the associated line items from the system.

```
public static void cancelOrdersUsingIndex(Session s)
throws ObjectGridException {
    // Cancel all orders that were submitted 1 minute ago
    java.util.Date cancelTime = new
    java.util.Date(System.currentTimeMillis() - 60000);
    EntityManager em = s.getEntityManager();
    em.getTransaction().begin();
    MapRangeIndex dateIndex = (MapRangeIndex)
    s.getMap("Order").getIndex("date");
    Iterator<Tuple> orderKeys = dateIndex.findGreaterEqual(cancelTime);
    while(orderKeys.hasNext()) {
        Tuple orderKey = orderKeys.next();
        // Find the Order so we can remove it.
        Order curOrder = (Order) em.find(Order.class, orderKey);
        // Verify that the order was not updated by someone else.
        if(curOrder != null && curOrder.date.getTime() >= cancelTime.getTime()) {
            for(OrderLine line : curOrder.lines) {
                // Add the item back to the inventory.
                line.item.quantityOnHand += line.quantity;
                line.quantity = 0;
            }
            em.remove(curOrder);
        }
    }
    em.getTransaction().commit();
}
```

Entity manager tutorial: Updating and removing entries by using a query

You can update and remove entities by using a query.

Procedure

Update and remove entities by using a query.

Order.java

```
@Entity
public class Order
{
    @Id String orderNumber;
    @Index java.util.Date date;
    @OneToOne(cascade=CascadeType.PERSIST) Customer customer;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="order")
    @OrderBy("lineNumber") List<OrderLine> lines;
}
```

The order entity class is the same as it is in the previous example. The class still provides the `@Index` annotation, because the query string uses the date to find the entity. The query engine uses indices when they can be used.

```
public static void cancelOrdersUsingQuery(Session s) {
    // Cancel all orders that were submitted 1 minute ago
    java.util.Date cancelTime =
    new java.util.Date(System.currentTimeMillis() - 60000);
    EntityManager em = s.getEntityManager();
    em.getTransaction().begin();

    // Create a query that will find the order based on date. Since
    // we have an index defined on the order date, the query
    // will automatically use it.
    Query query = em.createQuery("SELECT order FROM Order order
    WHERE order.date >= ?1");
    query.setParameter(1, cancelTime);
    Iterator<Order> orderIterator = query.getResultIterator();
    while(orderIterator.hasNext()) {
        Order order = orderIterator.next();
        // Verify that the order wasn't updated by someone else.
        // Since the query used an index, there was no lock on the row.
        if(order != null && order.date.getTime() >= cancelTime.getTime()) {
            for(OrderLine line : order.lines) {
                // Add the item back to the inventory.
                line.item.quantityOnHand += line.quantity;
                line.quantity = 0;
            }
            em.remove(order);
        }
    }
    em.getTransaction().commit();
}
```

Like the previous example, the `cancelOrdersUsingQuery` method intends to cancel all orders that were submitted in the past minute. To cancel the order, you find the order using a query, add the items in the order back into the inventory, and remove the order and associated line items from the system.

Tutorial: Running eXtreme Scale bundles in the OSGi framework

The OSGi sample builds on the Google Protocol Buffers serializer samples. When you complete this set of lessons, you will have run the serializer sample plug-ins in the OSGi framework.

Learning objectives

This sample demonstrates the OSGi bundles. The serializer plug-in is incidental and is not required. The OSGi sample is available on the WebSphere eXtreme Scale samples gallery. You must download the sample, and extract it into the `wxs_home/samples` directory. The root directory for the OSGi sample is `wxs_home/samples/OSGiProto`.

The command examples in this tutorial assume that you are running on the UNIX operating system. You must adjust the command example to run on a Windows operating system.

After completing the lessons in this tutorial, you will understand the OSGi sample concepts and know how to complete the following objectives:

- Install the WebSphere eXtreme Scale server bundle into the OSGi container to start the eXtreme Scale server.
- Set up your eXtreme Scale development environment to run the sample client.
- Use the `xscmd` command to query the service ranking of the sample bundle, upgrade it to a new service ranking, and verify the new service ranking.

Time required

This module takes approximately 60 minutes to complete.

Prerequisites

In addition to downloading and extracting the serializer samples, this tutorial also has the following prerequisites:

- Install and extract the eXtreme Scale product
- Set up the Eclipse Equinox Environment

Introduction: Starting and configuring the eXtreme Scale server and container to run plug-ins in the OSGi framework

In this tutorial you start an eXtreme Scale server in the OSGi framework, start an eXtreme Scale container, and wire the sample plug-ins with eXtreme Scale runtime environment.

Learning objectives

After completing the lessons in this tutorial you will understand the OSGi sample concepts and know how to complete the following objectives:

- Install the WebSphere eXtreme Scale server bundle into the OSGi container to start the eXtreme Scale server.
- Set up your eXtreme Scale development environment to run the sample client.
- Use the `xscmd` command to query the service ranking of the sample bundle, upgrade it to a new service ranking, and verify the new service ranking.

Time required

This tutorial takes approximately 60 minutes to finish. If you explore other concepts related to this tutorial, it might take longer to complete.

Skill level

Intermediate.

Audience

Developers and administrators who want to build, install, and run eXtreme Scale bundles into the OSGi framework.

System requirements

- Luminis OSGi Configuration Admin command line client, version 0.2.5
- Apache Felix File Install, version 3.0.2
- When using Eclipse Gemini as the Blueprint container provider, the following are required:
 - Eclipse Gemini Blueprint, version 1.0.0
 - Spring Framework, version 3.0.5
 - SpringSource AOP Alliance API, version 1.0.0
 - SpringSource Apache Commons Logging, version 1.1.1
- When using Apache Aries as the Blueprint Container provider, you must have the following requirements:
 - Apache Aries, latest snapshot
 - ASM library
 - PAX logging

Prerequisites

To complete this tutorial, you must download the sample, and extracted it into the `wxs_home/samples` directory. The root directory for the OSGi sample is `wxs_home/samples/OSGiProto`.

Expected results

When you complete this tutorial, you will have installed the sample bundles and run an eXtreme Scale client to insert data into the grid. You can also expect to query and update those sample bundles using the dynamic capabilities that the OSGi container provides.

Module 1: Preparing to install and configure eXtreme Scale server bundles

Complete this module to explore OSGi sample bundles and examine configuration files that you use to configure the eXtreme Scale server.

Learning objectives

After completing the lessons in this module, you will understand the concepts and know how to complete the following objectives:

- Locate and explore the bundles that are included in the OSGi sample.
- Examine configuration files that are used to configure the eXtreme Scale grid and server.

Lesson 1.1: Understand the OSGi sample bundles

Complete this lesson to locate and explore the bundles that are provided in the OSGi sample.

OSGi sample bundles:

Other than the bundles that are configured in the `config.ini` file, which is shown in the topic about setting up the Eclipse Equinox environment, the following additional bundles are used in the OSGi sample:

objectgrid.jar

The WebSphere eXtreme Scale server runtime bundle. This bundle is located in the `wxs_home/lib` directory.

com.google.protobuf_2.4.0a.jar

The Google Protocol Buffers, version 2.4.0a bundle. This bundle is located in the `wxs_sample_osgi_root/lib` directory.

ProtoBufSamplePlugins-1.0.0.jar

Version 1.0.0 of the user plug-in bundle with sample `ObjectGridEventListener` and `MapSerializerPlugin` plug-in implementations. This bundle is located in the `wxs_sample_osgi_root/lib` directory. The services are configured with service ranking 1.

This version uses the standard Blueprint XML to configure the eXtreme Scale plug-in services. The service class is a user-implemented class for WebSphere eXtreme Scale interface, `com.ibm.websphere.objectgrid.plugins.osgi.PluginServiceFactory`. The user-implemented class creates a bean for each request and works similar to a prototype-scoped bean.

ProtoBufSamplePlugins-2.0.0.jar

Version 2.0.0 of the user plug-in bundle with sample `ObjectGridEventListener` and `MapSerializerPlugin` plug-in implementations. This bundle is located in the `wxs_sample_osgi_root/lib` directory. The services are configured with service ranking 2.

This version uses the standard Blueprint XML to configure the eXtreme Scale plug-in services. The service class is using a WebSphere eXtreme Scale, built-in class, `com.ibm.websphere.objectgrid.plugins.osgi.PluginServiceFactoryImpl`, which uses the `BlueprintContainer` service. Using the standard Blueprint XML configuration, the beans can be configured either as a prototype scope or singleton scope. The bean is not configured as a shard scope.

ProtoBufSamplePlugins-Gemini-3.0.0.jar

Version 3.0.0 of the user plug-in bundle with sample `ObjectGridEventListener` and `MapSerializerPlugin` plug-in implementations. This bundle is located in the `wxs_sample_osgi_root/lib` directory. The services are configured with service ranking 3.

This version uses the Eclipse Gemini-specific Blueprint XML to configure the eXtreme Scale plug-in services. The service class is using a WebSphere eXtreme Scale built-in class, `com.ibm.websphere.objectgrid.plugins.osgi.PluginServiceFactoryImpl`, which uses the `BlueprintContainer` service. The way to configure a shard scope bean is using a Gemini-specific approach. This version configures the `myShardListener` bean as a shard scope bean by providing `{http://www.ibm.com/schema/objectgrid}shard` as the scope value, and configuring a dummy attribute so that the custom scope is recognized by

Gemini. This is due to the following Eclipse issue: https://bugs.eclipse.org/bugs/show_bug.cgi?id=348776

ProtoBufSamplePlugins-Aries-4.0.0.jar

Version 4.0.0 of the user plug-in bundle with sample ObjectGridEventListener and MapSerializerPlugin plug-in implementations. This bundle is located in the *wxs_sample_osgi_root/lib* directory. The services are configured with service ranking 4.

This version uses standard Blueprint XML to configure the eXtreme Scale plug-in services. The service class is using a WebSphere eXtreme Scale, built-in class,

com.ibm.websphere.objectgrid.plugins.osgi.PluginServiceFactoryImpl, which uses the BlueprintContainer service. Using the standard Blueprint XML configuration, the beans can be configured using a custom scope. This version configures the myShardListenerbean as a shard scoped bean by providing {http://www.ibm.com/schema/objectgrid}shard as the scope value.

ProtoBufSamplePlugins-Activator-5.0.0.jar

Version 5.0.0 of the user plug-in bundle with sample ObjectGridEventListener and MapSerializerPlugin plug-in implementations. This bundle is located in the *wxs_sample_osgi_root/lib* directory. The services are configured with service ranking 5.

This version does not use Blueprint container at all. In this version, the services are registered using OSGi service registration. The service class is a user-implemented class for the WebSphere eXtreme Scale interface, com.ibm.websphere.objectgrid.plugins.osgi.PluginServiceFactory. The user-implemented class creates a bean for each request. It works similar to a prototype-scoped bean.

Lesson checkpoint:

By exploring the bundles that are provided with the OSGi sample, you can better understand how to develop your own implementations that will run in the OSGi container.

You learned:

- About bundles that included with the OSGi sample
- The location of those bundles
- The service ranking that each bundle has been configured with

Lesson 1.2: Understand the OSGi configuration files

The OSGi sample includes configuration files that you use to start and configure the WebSphere eXtreme Scale grid and server.

OSGi configuration files:

In this lesson, you will explore the following configuration files that are included with the OSGi sample:

- collocated.server.properties
- protoBufObjectGrid.xml
- protoBufDeployment.xml
- blueprint.xml

collocated.server.properties

A server configuration is required to start a server. When the eXtreme Scale server bundle is started, it does not start a server. It waits for the configuration PID, `com.ibm.websphere.xs.server`, to be created with a server property file. This server property file specifies the server name, port number, and other server properties.

In most cases, you create a configuration to set the server property file. In rare cases, you might want only to start a server, with every property set to a default value. In that case, you can create a configuration called `com.ibm.websphere.xs.server` with value set to `default`.

For more details about the server property file, see the [Server properties file](#) topic.

The OSGi sample server properties file starts a single catalog. This sample property file starts a single catalog service and a container server in the OSGi framework process. eXtreme Scale clients connect to port 2809 and JMX clients connect to port 1099. The content of the sample server property file is:

```
serverName=collocatedServer
isCatalog=true
catalogClusterEndpoints=collocatedServer:localhost:6601:6602
traceSpec=ObjectGridOSGi=all=enabled
traceFile=logs/trace.log
listenerPort=2809
JMXServicePort=1099
```

protoBufObjectGrid.xml

The sample `protoBufObjectGrid.xml` ObjectGrid descriptor XML file contains the following content, with comments removed.

```
<objectGridConfig
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">

  <objectGrids>
    <objectGrid name="Grid" txTimeout="15">

      <bean id="ObjectGridEventListener"
        osgiService="myShardListener"/>

      <backingMap name="Map" readOnly="false"
        lockStrategy="PESSIMISTIC" lockTimeout="5"
        copyMode="COPY_TO_BYTES"
        pluginCollectionRef="serializer"/>

    </objectGrid>
  </objectGrids>

  <backingMapPluginCollections>
    <backingMapPluginCollection id="serializer">
      <bean id="MapSerializerPlugin"
        osgiService="myProtoBufSerializer"/>
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

There are two plug-ins configured in this ObjectGrid descriptor XML file:

ObjectGridEventListener

The shard-level plug-in. For each ObjectGrid instance, there is an instance of ObjectGridEventListener. It is configured to use the OSGi service `myShardListener`. That means when the grid is created, the ObjectGridEventListener plug-in uses the OSGi service `myShardListener` with the highest service ranking available.

MapSerializerPlugin

The map-level plug-in. For the backing map named `Map`, there is a `MapSerializerPlugin` plug-in configured. It is configured to use the OSGi service `myProtoBufSerializer`. That means when the map is created, the `MapSerializerPlugin` plug-in uses the service, `myProtoBufSerializer`, with the highest ranked service ranking available.

protoBufDeployment.xml

The deployment descriptor XML file describes the deployment policy for the grid named `Grid`, which uses five partitions. See the following code example of the XML file:

```
<deploymentPolicy
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/deploymentPolicy ../deploymentPolicy.xsd"
  xmlns="http://ibm.com/ws/objectgrid/deploymentPolicy">

  <objectgridDeployment objectgridName="Grid">
    <mapSet name="MapSet" numberOfPartitions="5">
      <map ref="Map"/>
    </mapSet>
  </objectgridDeployment>
</deploymentPolicy>
```

blueprint.xml

As an alternative to using the `collocated.server.properties` file in conjunction with configuration PID, `com.ibm.websphere.xs.server`, you can include the ObjectGrid XML and deployment XML files in an OSGi bundle, along with a Blueprint XML file as shown in the following example:

```
<blueprint
  xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:objectgrid="http://www.ibm.com/schema/objectgrid"
  default-activation="lazy">

  <objectgrid:server id="server" isCatalog="true"
    name="server"
    tracespec="ObjectGridOSGi=all=enabled"
    tracefile="C:/Temp/logs/trace.log"
    workingDirectory="C:/Temp/working"
    jmxport="1099">
    <objectgrid:catalog host="localhost" port="2809"/>
  </objectgrid:server>

  <objectgrid:container id="container"
    objectgridxml="/META-INF/objectgrid.xml"
    deploymentxml="/META-INF/deployment.xml"
    server="server"/>
</blueprint>
```

Lesson checkpoint:

In this lesson, you learned about the configuration files that are used in the OSGi sample. Now, when you start and configure the eXtreme Scale grid and server, you will understand which files are being used in these processes and how these files interact with your plug-ins in the OSGi framework.

Module 2: Installing and starting eXtreme Scale bundles in the OSGi framework

Use the lessons in this module to install the eXtreme Scale server bundle into the OSGi container, and start the WebSphere eXtreme Scale server.

Starting the server in the OSGi framework does not mean that your OSGi bundles are ready to run. You must configure the server properties and containers so that the OSGi bundles that you install are recognized and can run correctly.

Learning objectives

After completing the lessons in this module, you will understand the concepts and know how to complete the following tasks:

- Install eXtreme Scale bundles using the Equinox OSGi console.
- Configure the eXtreme Scale server.
- Configure the eXtreme Scale container.
- Install and start eXtreme Scale sample bundles.

Prerequisites

To complete this module, the following tasks are required before you begin:

- Install and extract the eXtreme Scale product
- Set up the Eclipse Equinox Environment

You must also prepare to access the following files to complete the lessons in this module:

- `objectgrid.jar` bundle. You install this eXtreme Scale bundle.
- `collocated.server.properties` file. You add the server properties to this configuration file.

You can expect to install and start the following bundles:

- `protobuf-java-2.4.0a-bundle.jar` bundle
- `ProtoBufSamplePlugins-1.0.0.jar` bundle

Lesson 2.1: Start the console and install the eXtreme Scale server bundle

In this lesson, you use the Equinox OSGi console to install the WebSphere eXtreme Scale server bundle.

1. Use the following command to start the Equinox OSGi console:

```
cd equinox_root
java -jar plugins\org.eclipse.osgi_3.6.1.R36x_v20100806.jar -console
```

2. After the OSGi console is started, issue the `ss` command in the console, and the following bundles are started:

Attention: If you completed the task, Installing eXtreme Scale bundles, then the bundle has already been activated. If the bundle is started, then stop the bundle before you complete this step.

Eclipse Gemini output:

```
osgi> ss
Framework is launched.
id State Bundle
0 ACTIVE org.eclipse.osgi_3.6.1.R36x_v20100806
1 ACTIVE org.eclipse.osgi.services_3.2.100.v20100503
2 ACTIVE org.eclipse.osgi.util_3.2.100.v20100503
3 ACTIVE org.eclipse.equinox.cm_1.0.200.v20100520
4 ACTIVE com.springsource.org.apache.commons.logging_1.1.1
5 ACTIVE com.springsource.org.aopalliance_1.0.0
6 ACTIVE org.springframework.aop_3.0.5.RELEASE
7 ACTIVE org.springframework.asm_3.0.5.RELEASE
8 ACTIVE org.springframework.beans_3.0.5.RELEASE
9 ACTIVE org.springframework.context_3.0.5.RELEASE
```

```

10 ACTIVE org.springframework.core_3.0.5.RELEASE
11 ACTIVE org.springframework.expression_3.0.5.RELEASE
12 ACTIVE org.apache.felix.fileinstall_3.0.2
13 ACTIVE net.luminis.cmc_0.2.5
14 ACTIVE org.eclipse.gemini.blueprint.core_1.0.0.RELEASE
15 ACTIVE org.eclipse.gemini.blueprint.extender_1.0.0.RELEASE
16 ACTIVE org.eclipse.gemini.blueprint.io_1.0.0.RELEASE

```

Apache Aries output:

```

osgi> ss
Framework is launched.
id State Bundle
0 ACTIVE org.eclipse.osgi_3.6.1.R36x_v20100806
1 ACTIVE org.eclipse.osgi.services_3.2.100.v20100503
2 ACTIVE org.eclipse.osgi.util_3.2.100.v20100503
3 ACTIVE org.eclipse.equinox.cm_1.0.200.v20100520
4 ACTIVE org.ops4j.pax.logging.pax-logging-api_1.6.3
5 ACTIVE org.ops4j.pax.logging.pax-logging-service_1.6.3
6 ACTIVE org.objectweb.asm.all_3.3.0
7 ACTIVE org.apache.aries.blueprint_0.3.2.SNAPSHOT
8 ACTIVE org.apache.aries.util_0.4.0.SNAPSHOT
9 ACTIVE org.apache.aries.proxy_0.4.0.SNAPSHOT
10 ACTIVE org.apache.felix.fileinstall_3.0.2
11 ACTIVE net.luminis.cmc_0.2.5

```

3. Install the `objectgrid.jar` bundle. To start a server in the Java virtual machine (JVM), you need to install an eXtreme Scale server bundle. This eXtreme Scale server bundle can start a server and create containers. Use the following command to install the `objectgrid.jar` file:

```
osgi> install file:///wxs_home/lib/objectgrid.jar
```

See the following example:

```
osgi> install file:///opt/wxs/ObjectGrid/lib/objectgrid.jar
```

Equinox displays its bundle ID; for example:

```
Bundle id is 19
```

Remember: Your bundle ID might be different. The file path must be an absolute URL to the bundle path. Relative paths are not supported.

Lesson checkpoint:

In this lesson, you used the Equinox OSGi console to install the `objectgrid.jar` bundle, which you will use to start a server and create a container later in this tutorial.

Lesson 2.2: Customize and configure the eXtreme Scale server

Use this lesson to customize and add the server properties to the WebSphere eXtreme Scale server.

1. Edit the `wxs_sample_osgi_root/projects/server/properties/collocated.server.properties` file.
 - a. Change the `traceFile` property to `equinox_root/logs/trace.log`.
2. Save the file.
3. Enter the following lines of code in the OSGI console to create the server configuration from the file. The following example is displayed on multiple lines for publication purposes.

```
osgi> cm create com.ibm.websphere.xs.server
```

```
osgi> cm put com.ibm.websphere.xs.server objectgrid.server.props wxs_sample_osgi_root/projects/server/properties/collocated
.server.properties
```

4. To view the configuration, run the following command:

```

osgi> cm get com.ibm.websphere.xs.server
Configuration for service (pid) "com.ibm.websphere.xs.server"
(bundle location = null)
key                               value
----                               -
objectgrid.server.props           wxs_sample_osgi_root/projects/server/properties/collocated.server.properties
service.pid                       com.ibm.websphere.xs.server

```

Lesson checkpoint:

In this lesson, you edited the `wxs_sample_osgi_root/projects/server/properties/collocated.server.properties` file to specify server settings, such as the working directory and the location for the trace log files.

Lesson 2.3: Configure the eXtreme Scale container

Complete this lesson to configure a container, which includes the WebSphere eXtreme Scale ObjectGrid descriptor XML file and ObjectGrid deployment XML file. These files include the configuration for the grid and its topology.

To create a container, first create a configuration service using the managed service factory process identification number (PID), `com.ibm.websphere.xs.container`. The service configuration is a managed service factory, so you can create multiple service PIDs from the factory PID. Then, to start the container service, set the `objectgridFile` and `deploymentPolicyFile` PIDs to each service PID.

Complete the following steps to customize and add the server properties to the OSGi framework:

1. In the OSGI console, enter the following command to create the container from the file:

```

osgi> cm createf com.ibm.websphere.xs.container
PID: com.ibm.websphere.xs.container-1291179621421-0

```

2. Enter the following commands to bind the newly created PID to the ObjectGrid XML files.

Remember: The PID number will be different from what is included in this example.

```

osgi> cm put com.ibm.websphere.xs.container-1291179621421-0 objectgridFile wxs_sample_osgi_root/projects/server/META-INF/protoBufObjectgrid.xml
osgi> cm put com.ibm.websphere.xs.container-1291179621421-0 deploymentPolicyFile wxs_sample_osgi_root/projects/server/META-INF/protoBufDeployment.xml

```

3. Use the following command to display the configuration:

```

osgi> cm get com.ibm.websphere.xs.container-1291760127968-0
Configuration for service (pid) "com.ibm.websphere.xs.container-1291760127968-0"
(bundle location = null)

key                               value
----                               -
deploymentPolicyFile             /opt/wxs/ObjectGrid/samples/OSGiProto/server/META-INF/protoBufDeployment.xml
objectgridFile                   /opt/wxs/ObjectGrid/samples/OSGiProto/server/META-INF/protoBufObjectgrid.xml
service.factoryPid               com.ibm.websphere.xs.container
service.pid                       com.ibm.websphere.xs.container-1291760127968-0

```

Lesson checkpoint:

In this lesson, you created a configuration service, which you used to create an eXtreme Scale container. Since the ObjectGrid XML files contain the configuration for the grid and its topology, you had to bind the container that you created to those ObjectGrid XML files. With this configuration, the eXtreme Scale container can recognize the OSGi bundles that you will run later in this tutorial.

Lesson 2.4: Install the Google Protocol Buffers and sample plug-in bundles

Complete this tutorial to install the `protobuf-java-2.4.0a-bundle.jar` bundle and the `ProtoBufSamplePlugins-1.0.0.jar` plug-in bundle using the Equinox OSGi console.

Install the Google Protocol Buffers plug-in:

Complete the following steps to install the Google Protocol Buffers plug-in.

In the OSGi console, enter the following command to install the plug-in:

```
osgi> install file:///wxs_sample_osgi_root/lib/com.google.protobuf_2.4.0a.jar
```

The following output is displayed:

```
Bundle ID is 21
```

Sample plug-in bundles overview:

The OSGi sample includes five sample bundles that include eXtreme Scale plug-ins, including a custom `ObjectGridEventListener` and `MapSerializerPlugin` plug-in. The `MapSerializerPlugin` plug-in uses the Google Protocol Buffers sample and messages provided by the `MapSerializerPlugin` sample.

The following bundles are located in `wxs_sample_osgi_root/lib` directory: `ProtoBufSamplePlugins-1.0.0.jar` and the `ProtoBufSamplePlugins-2.0.0.jar`.

The `blueprint.xml` file has the following content with comments removed:

```
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
  <bean id="myShardListener" class="com.ibm.websphere.samples.xs.proto.osgi.MyShardListenerFactory"/>
  <service ref="myShardListener" interface="com.ibm.websphere.objectgrid.plugins.osgi.PluginServiceFactory" ranking="1">
  </service>

  <bean id="myProtoBufSerializer" class="com.ibm.websphere.samples.xs.proto.osgi.ProtoMapSerializerFactory">
    <property name="keyType" value="com.ibm.websphere.samples.xs.serializer.app.proto.DataObjects1$OrderKey" />
    <property name="valueType" value="com.ibm.websphere.samples.xs.serializer.app.proto.DataObjects1$Order" />
  </bean>

  <service ref="myProtoBufSerializer" interface="com.ibm.websphere.objectgrid.plugins.osgi.PluginServiceFactory"
    ranking="1">
  </service>
</blueprint>
```

The Blueprint XML file exports two services, `myShardListener` and `myProtoBufSerializer`. These two services are referenced in the `protoBufObjectgrid.xml` file.

Install the sample plug-in bundle:

Complete the following steps to install the `ProtoBufSamplePlugins-1.0.0.jar` bundle.

Run the following command in the Equinox OSGi console to install the `ProtoBufSamplePlugins-1.0.0.jar` plug-in bundle:

```
osgi> install file:///wxs_sample_osgi_root/lib/ProtoBufSamplePlugins-1.0.0.jar
```

The following output is displayed:

```
Bundle ID is 22
```

Lesson checkpoint:

In this lesson, you installed the `protobuf-java-2.4.0a-bundle.jar` bundle and the `ProtoBufSamplePlugins-1.0.0.jar` plug-in bundle.

Lesson 2.5: Start the OSGi bundles

The WebSphere eXtreme Scale server is packaged as an OSGi server bundle. Complete this lesson to install the eXtreme Scale server bundle as well as other OSGi bundles that you have installed.

1. Run the `ss` command to view the IDs for each bundle.

```
osgi> ss
```

```
Framework is launched.
```

```
id State Bundle
0 ACTIVE org.eclipse.osgi_3.6.1.R36x_v20100806
1 ACTIVE org.eclipse.osgi.services_3.2.100.v20100503
2 ACTIVE org.eclipse.osgi.util_3.2.100.v20100503
3 ACTIVE org.eclipse.equinox.cm_1.0.200.v20100520
4 ACTIVE com.springsource.org.apache.commons.logging_1.1.1
5 ACTIVE com.springsource.org.aopalliance_1.0.0
6 ACTIVE org.springframework.aop_3.0.5.RELEASE
7 ACTIVE org.springframework.asm_3.0.5.RELEASE
8 ACTIVE org.springframework.beans_3.0.5.RELEASE
9 ACTIVE org.springframework.context_3.0.5.RELEASE
10 ACTIVE org.springframework.core_3.0.5.RELEASE
11 ACTIVE org.springframework.expression_3.0.5.RELEASE
12 ACTIVE org.apache.felix.fileinstall_3.0.2
13 ACTIVE net.luminis.cmc_0.2.5
15 ACTIVE org.eclipse.gemini.blueprint.core_1.0.0.RELEASE
16 ACTIVE org.eclipse.gemini.blueprint.extender_1.0.0.RELEASE
17 ACTIVE org.eclipse.gemini.blueprint.io_1.0.0.RELEASE
19 RESOLVED com.ibm.websphere.xs.server_7.1.1
21 RESOLVED Google_Protobuf_2.4.0
22 RESOLVED ProtoBufPlugins_1.0.0
```

2. Start each bundle that you have installed. You must start the bundles in a specific order. See the order of the bundle IDs from the previous example.
 - a. Start the sample plug-in bundle, `ProtoBufPlugins_1.0.0`. Run the following command in the Equinox OSGi console to start the bundle. In this example, the bundle ID of the sample plug-in is 22.

```
osgi> start 22
```

- b. Start the Google Protocol Buffers bundle, `Google_Protobuf_2.4.0`. Run the following command in the Equinox OSGi console to start the bundle. In this example, the bundle ID of the Google Protocol Buffers plug-in is 21.

```
osgi> start 21
```

- c. Start the server bundle, `com.ibm.websphere.xs.server_7.1.1`. Run the following command in the OSGi console to start the server. In this example, the bundle ID of the eXtreme Scale server bundle is 19.

```
osgi> start 19
```

After you start the server, the `MyShardListener` event listener is started and ready to insert or update records. You can see the following output on the OSGi console to confirm that the plug-in bundle has started successfully:

```
SystemOut 0 MyShardListener@1253853884(version=1.0.0) order
com.ibm.websphere.samples.xs.serializer.proto.DataObjects1$Order$Builder
@1ab1aba(22) inserted
```

Lesson checkpoint:

In this lesson, you started two plug-in bundles and the server bundle in the eXtreme Scale container that you configured for the OSGi framework.

Module 3: Running the eXtreme Scale sample client

The WebSphere eXtreme Scale server is now running in an OSGi environment. Complete the steps in this module to run an WebSphere eXtreme Scale client that inserts data into the grid.

Learning objectives

After completing the lessons in this module you will know how to complete the following tasks:

- Run a client application that connects to the grid and inserts and retrieves some data from it.
- Start an order using a non-OSGi client application.

Prerequisites

Complete Module 2: Installing and starting eXtreme Scale bundles in the OSGi framework.

Lesson 3.1: Set up Eclipse to run the client and build the samples

Complete this lesson to import the Eclipse project that you will use to run the client and build the sample plug-ins.

The sample includes a Java SE client program that connects to the grid and inserts and retrieves data from it. It also includes projects that you can use to build and redeploy the OSGi bundles.

The provided project has been tested with Eclipse 3.x and later, and requires only the standard Java development project perspective. Complete the following steps to set up of your WebSphere eXtreme Scale development environment.

1. Open Eclipse to a new or existing workspace.
2. From the File menu, select **Import**.
3. Expand the General folder. Select **Existing Projects into Workspace**, and click **Next**.
4. In the **Select root directory** field, type or browse to the *wxs_sample_osgi_root* directory. Click **Finish**. Several new projects are displayed in your workspace. Build errors will be fixed by defining two user libraries. Complete the next steps to define the user libraries.
5. From the Window menu, select **Preferences**.
6. Expand the **Java > Build Path** branch, and select **User Libraries**.
7. Define the eXtreme Scale user library.
 - a. Click **New**.
 - b. Type `eXtremeScale` in the **User Library Name** field, and click **OK**.
 - c. Select the new user library, and click **Add JARs**.
 - 1) Browse and select the `objectgrid.jar` file from the *wxs_install_root/lib* directory. Click **OK**.
 - 2) To include API documentation for the ObjectGrid APIs, select the API documentation location for the `objectgrid.jar` file that you added in the previous step. Click **Edit**.

- 3) In the location path box for the API documentation, select the Javadoc.zip file that is included in the following directory:
wxs_install_root/docs/javadoc.zip.
8. Define the Google Protocol Buffers user library.
 - a. Click **New**.
 - b. Type `com.google.protobuf` in the **User Library Name** field, and click **OK**.
 - c. Select the new user library, and click **Add JARs**.
 - 1) Browse and select the `com.google.protobuf_2.4.0.a.jar` file from the *wxs_sample_osgi_root/lib* directory. Click **OK**.

Lesson checkpoint:

In this lesson, you imported the sample Eclipse project and defined the user libraries that fixed any build errors.

Lesson 3.2: Start a client and insert data into the grid

Complete this lesson to start a non-OSGi client and run a client application.

The Java client application is `com.ibm.websphere.samples.xs.proto.client.Client`. The Eclipse project, `wxs.sample.osgi.protobuf.client`, contains the Java client application. The main class file is `com.ibm.websphere.samples.xs.proto.client.Client`.

This client uses a client override, ObjectGrid descriptor XML file to override the OSGi configuration, so that the client can run in a non-OSGi environment. See the following content of the file with comments and headers removed. Some lines of code are displayed on multiple lines for formatting purposes.

```
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">

  <objectGrids>
    <objectGrid name="Grid" txTimeout="15">
      <bean id="ObjectGridEventListener" className="" osgiService="" />
      <backingMap name="Map" readOnly="false"
        lockStrategy="PESSIMISTIC" lockTimeout="5"
        copyMode="COPY_TO_BYTES" pluginCollectionRef="serializer"/>
    </objectGrid>
  </objectGrids>

  <backingMapPluginCollections>
    <backingMapPluginCollection id="serializer">

    <bean id="MapSerializer"
      className="com.ibm.websphere.samples.xs.serializer.proto.ProtoMapSerializer"
      osgiService="">
      <property name="keyType" type="java.lang.String"
        value="com.ibm.websphere.samples.xs.serializer.proto.DataObjects2$0OrderKey" />
      <property name="valueType" type="java.lang.String"
        value="com.ibm.websphere.samples.xs.serializer.proto.DataObjects2$0Order" />
    </bean>
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

Click **Run As > Java Application** to run the client application.

When you run the application, the following message is displayed. The message indicates that an order was inserted:

```
order
com.ibm.websphere.samples.xs.serializer.proto.DataObjects1$0Order$Builder@5d165d16(5000000) inserted
```


Lesson checkpoint:

In this lesson, you started the `com.ibm.websphere.samples.xs.proto.client.Client` application, which produced an order.

Module 4: Querying and upgrading the sample bundle

Complete the lessons in this module to use the `xscmd` command to query the service ranking of the sample bundle, upgrade it to a new service ranking, and verify the new service ranking.

Learning objectives

After completing the lessons in this module you will know how to complete the tasks:

- Query the current service ranking for a service.
- Query the current ranking for all services.
- Query all available rankings for a service.
- Query all available service rankings.
- Use the `xscmd` tool to verify whether specific service rankings are available.
- Update service rankings for sample OSGi services.

Prerequisites

Complete Module 3: Running the eXtreme Scale sample client.

Lesson 4.1: Query service rankings

Complete this lesson to query current service rankings as well as those service rankings that are available for upgrade.

- Query the current service ranking for a service. Enter the following command to query the current service ranking being used for service, `myShardListener`, which is used by the ObjectGrid named `Grid` and map set named `MapSet`.

1. Switch to the following directory:

```
cd wxs_home/bin
```

2. Enter the following command to query the current service ranking for the service, `myShardListener`.

```
./xscmd.sh -c osgiCurrent -g Grid -ms MapSet -sn myShardListener
```

The following output is displayed:

```
OSGi Service Name: myShardListener
ObjectGrid Name MapSet Name Server Name      Current Ranking
-----
Grid           MapSet      collocatedServer  1
```

```
CWXS10040I: The command osgiCurrent has completed successfully.
```

- Query the current ranking for all services. Enter the following command to query the current service ranking for all services that are used by the ObjectGrid named `Grid` and map set named `MapSet`.

1. Switch to the following directory:

```
cd wxs_home/bin
```

2. Enter the following command to query the current service ranking for all services.

```
./xscmd.sh -c osgiCurrent -g Grid -ms MapSet
```

The following output is displayed:

OSGi Service Name	Current Ranking	ObjectGrid Name	MapSet Name	Server Name
myProtoBufSerializer	1	Grid	MapSet	collocatedServer
myShardListener	1	Grid	MapSet	collocatedServer

CWXS10040I: The command osgiCurrent has completed successfully.

- Query all available rankings for a service. Enter the following command to query all of the available service rankings for the service named myShardListener.

1. Switch to the following directory:

```
cd wxs_home/bin
```

2. Enter the following command to query all available rankings for a service.

```
./xscmd.sh -c osgiAll -sn myShardListener
```

The following output is displayed:

```
Server: collocatedServer
OSGi Service Name Available Rankings
-----
myShardListener 1
```

Summary - All servers have the same service rankings.

CWXS10040I: The command osgiAll has completed successfully.

The output is grouped by the server. In this example, only the following server exists: collocatedServer.

- Query all available service rankings. Enter the following command to query all of the available service rankings for all services.

1. Switch to the following directory:

```
cd wxs_home/bin
```

2. Enter the following command to query all available service rankings.

```
./xscmd.sh -c osgiAll
```

The following output is displayed:

```
Server: collocatedServer
OSGi Service Name Available Rankings
-----
myProtoBufSerializer 1
myShardListener 1
```

Summary - All servers have the same service rankings.

- Install and start Version 2 of the plug-in bundle. In the server OSGi console, install a new bundle that contains a new version of the Order class and the MapSerializerPlugin plug-in. See Lesson 2.4: Install the Google Protocol Buffers and sample plug-in bundles for details about how to install the ProtoBufSamplePlugins-2.0.0.jar bundle.

1. After the installation, start the new bundle. The services for your new bundle are available, but they are not used by the eXtreme Scale server yet. You must run a service update request to use a service with a specific version.

- Now when you query all the available service rankings again, the service ranking 2 is added in the output.

1. Switch to the following directory:

```
cd wxs_home/bin
```

2. Enter the following command to query all available service rankings.

```
./xscmd.sh -c osgiAll
```

The following output is displayed:

```
Server: collocatedServer
  OSGi Service Name   Available Rankings
  -----
  myProtoBufSerializer 1, 2
  myShardListener      1, 2
```

Summary - All servers have the same service rankings.

Lesson checkpoint:

In this tutorial, you queried currently specified and all available service rankings. You also displayed the service ranking for a new bundle that you installed and started.

Lesson 4.2: Determine whether specific service rankings are available

Complete this lesson to determine whether specific service rankings are available for the service names that you specify.

1. Enter the following command to determine whether the service named `myShardListener`, with service ranking 2 and service named `myProtoBufSerializer`, with service ranking 2 are available. The service ranking list is passed in using `-sr` option.

- a. Switch to the following directory:

```
cd wxs_home/bin
```

- b. Enter the following command to determine whether the services are available:

```
./xscmd.sh -c osgiCheck -sr "myShardListener;2,myProtoBufSerializer;2"
```

The following output is displayed:

```
CWXSIO040I: The command osgiCheck has completed successfully.
```

2. Enter the following command to determine whether the service named `myShardListener`, with service ranking 2 and the service named `myProtoBufSerializer`, with service ranking 3 are available.

- a. Switch to the following directory:

```
cd wxs_home/bin
```

- b. Enter the following command to determine whether the services are available:

```
./xscmd.sh -c osgiCheck -sr "myShardListener;2,myProtoBufSerializer;3"
```

The following output is displayed:

```
Server           OSGi Service           Unavailable Rankings
-----
collocatedServer myProtoBufSerializer           3
```

Lesson checkpoint:

In this lesson, you specified the services `myShardListener` and `myProtoBufSerializer`, along with specific service rankings to determine whether those rankings were available.

Lesson 4.3: Update the service rankings

Complete this lesson to update current service rankings that you queried.

1. Update the service rankings of the services, myShardListener and myProtoBufSerializer, to service ranking 2. The service ranking list is passed in using -sr option.

- a. Switch to the following directory:

```
cd wxs_home/bin
```

- b. Enter the following command to update the service rankings:

```
./xscmd.sh -c osgiUpdate -g Grid -ms MapSet -sr "myShardListener;2,myProtoBufSerializer;2"
```

The following output is displayed:

Update succeeded for the following service rankings:

Service	Ranking
myProtoBufSerializer	2
myShardListener	2

CWXSIO040I: The command osgiUpdate has completed successfully.

The following output is displayed on the OSGi console:

```
SystemOut 0 MyShardListener@326505334(version=2.0.0) order
com.ibm.websphere.samples.xs.serializer.proto.DataObjects2$Order$Builder@
22342234(34) updated
```

Notice that the MyShardListener service is now version 2.0.0, which has service ranking 2.

2. Run the **xscmd** command to query the current service ranking for all services that are used by the ObjectGrid named Grid and the map set named MapSet.

- a. Switch to the following directory:

```
cd wxs_home/bin
```

- b. Enter the following command to query the service rankings for all services that are used by Grid and MapSet:

```
./xscmd.sh -c osgiCurrent -g Grid -ms MapSet
```

The following output is displayed:

OSGi Service Name	Current Ranking	ObjectGrid Name	MapSet Name	Server Name
myProtoBufSerializer	2	Grid	MapSet	collocatedServer
myShardListener	2	Grid	MapSet	collocatedServer

CWXSIO040I: The command osgiCurrent has completed successfully.

Lesson checkpoint:

In this lesson, you updated the service rankings for services myShardListener and myProtoBufSerializer.

Chapter 2. Scenarios



Scenarios include real-world information to build a complete picture. Complete a scenario to understand new concepts or to accomplish common WebSphere eXtreme Scale tasks.

Using an OSGi environment to develop and run eXtreme Scale plug-ins

Use these scenarios to complete common tasks in an OSGi environment. For example, the OSGi framework is ideal for starting servers and clients in an OSGi container, which allows you to dynamically add and update WebSphere eXtreme Scale plug-ins to the runtime environment.

Before you begin

Read the “OSGi framework overview” topic to learn more about OSGi support and the benefits that it can offer.

About this task

The following scenarios are about building and running dynamic plug-ins, which allows you to dynamically install, start, stop, modify, and uninstall plug-ins. You might also complete another likely scenario, which allows you to use the OSGi framework without dynamic capabilities. You can still package your applications as bundles, which are defined by and communicated through services. These service-based bundles offer multiple benefits, which include more efficient development and deployment capabilities.

Scenario goals

After completing this scenario, you will know how to complete the goals:

- Build eXtreme Scale dynamic plug-ins to use in an OSGi environment.
- Run eXtreme Scale containers in an OSGi environment without dynamic capabilities.

OSGi framework overview

OSGi defines a dynamic module system for Java. The OSGi service platform has a layered architecture, and is designed to run on various standard Java profiles. You can start WebSphere eXtreme Scale servers and clients in an OSGi container.

Benefits of running applications in the OSGi container

WebSphere eXtreme Scale OSGi support allows you to deploy the product in the Eclipse Equinox OSGi framework. Previously, if you wanted to update the plug-ins used by eXtreme Scale, you had to restart the Java virtual machine (JVM) to apply the new versions of the plug-ins. With the dynamic update capability that the OSGi framework provides, now you can update the plug-in classes without restarting the JVM. These plug-ins are exported by user bundles as services. WebSphere eXtreme Scale accesses the service or services by looking them up the OSGi registry.

eXtreme Scale containers can be configured to start more easily and dynamically using either the OSGi configuration admin service or with OSGi Blueprint. If you want to deploy a new data grid with its placement strategy, you can do so by creating an OSGi configuration or by deploying a bundle with eXtreme Scale descriptor XML files. With OSGi support, application bundles containing eXtreme Scale configuration data can be installed, started, stopped, updated, and uninstalled without restarting the whole system. With this capability, you can upgrade the application without disrupting the data grid.

Plug-in beans and services can be configured with custom shard scopes, allowing sophisticated integration options with other services running in the data grid. Each plug-in can use OSGi Blueprint rankings to verify that every instance of the plug-in is activated is at the correct version. An OSGi-managed bean (MBean) and **xscmd** utility are provided, which allow you to query the eXtreme Scale plug-in OSGi services and their rankings.

This capability allows administrators to quickly recognize potential configuration and administration errors and upgrade the plug-in service rankings in use by eXtreme Scale .

OSGi bundles

To interact with and deploy plug-ins in the OSGi framework, you must use *bundles*. In the OSGi service platform, a bundle is a Java archive (JAR) file that contains Java code, resources, and a manifest that describes the bundle and its dependencies. The bundle is the unit of deployment for an application. The eXtreme Scale product supports the following bundle types:

Server bundle

The server bundle is the `objectgrid.jar` file and is installed with the eXtreme Scale stand-alone server installation and is required for running eXtreme Scale servers and can also be used for running eXtreme Scale clients, or local, in-memory caches. The bundle ID for the `objectgrid.jar` file is `com.ibm.websphere.xs.server_<version>`, where the version is in the format: `<Version>.<Release>.<Modification>`. For example, the server bundle for eXtreme Scale version 7.1.1 is `com.ibm.websphere.xs.server_7.1.1`.

Client bundle

The client bundle is the `ogclient.jar` file and is installed with the eXtreme Scale stand-alone and client installations and is used to run eXtreme Scale clients or local, in-memory caches. The bundle ID for the `ogclient.jar` file is `com.ibm.websphere.xs.client_<version>`, where the version is in the format: `<Version>.<Release>.<Modification>`. For example, the client bundle for eXtreme Scale version 7.1.1 is `com.ibm.websphere.xs.client_7.1.1`.

Limitations

You cannot restart the eXtreme Scale bundle because you cannot restart the object request broker (ORB). To restart the eXtreme Scale server, you must restart the OSGi framework.

Installing the Eclipse Equinox OSGi framework with Eclipse Gemini for clients and servers

If you want to deploy WebSphere eXtreme Scale in the OSGi framework, then you must set up the Eclipse Equinox Environment.

About this task

The task requires that you download and install the Blueprint framework, which allows you to later configure JavaBeans and expose them as services. The use of services is important because you can expose plug-ins as OSGi services so they can be used by the eXtreme Scale run time environment. The product supports two blueprint containers within the Eclipse Equinox core OSGi framework: Eclipse Gemini and Apache Aries. Use this procedure to set up the Eclipse Gemini container.

Procedure

1. Download Eclipse Equinox SDK Version 3.6.1 or later from the Eclipse website. Create a directory for the Equinox framework, for example: `/opt/equinox`. These instructions refer to this directory as `equinox_root`. Extract the compressed file in the `equinox_root` directory.
2. Download the gemini-blueprint incubation 1.0.0 compressed file from the Eclipse website. Extract the file contents into a temporary directory, and copy the following extracted files to the `equinox_root/plugins` directory:

```
dist/gemini-blueprint-core-1.0.0.jar
dist/gemini-blueprint-extender-1.0.0.jar
dist/gemini-blueprint-io-1.0.0.jar
```

Attention: Depending on the location where you download the compressed Blueprint file, the extracted files might have the extension, `RELEASE.jar`, much like the Spring framework JAR files in the next step. You must verify that the file names match the file references in the `config.ini` file.

3. Download the Spring Framework Version 3.0.5 from the following SpringSource web page: <http://www.springsource.com/download/community>. Extract it into a temporary directory, and copy the following extracted files to the `equinox_root/plugins` directory:

```
org.springframework.aop-3.0.5.RELEASE.jar
org.springframework.asm-3.0.5.RELEASE.jar
org.springframework.beans-3.0.5.RELEASE.jar
org.springframework.context-3.0.5.RELEASE.jar
org.springframework.core-3.0.5.RELEASE.jar
org.springframework.expression-3.0.5.RELEASE.jar
```
4. Download the AOP Alliance Java archive (JAR) file from the SpringSource web page. Copy the `com.springsource.org.aopalliance-1.0.0.jar` to the `equinox_root/plugins` directory.
5. Download the Apache commons logging 1.1.1 JAR file from the SpringSource web page. Copy the `com.springsource.org.apache.commons.logging-1.1.1.jar` file to the `equinox_root/plugins` directory.
6. Download the Luminis OSGi Configuration Admin command-line client. Use this bundle to manage OSGi administrative configurations. You can download the JAR file from the following web page: <https://opensource.luminis.net/wiki/display/SITE/OSGi+Configuration+Admin+command+line+client>. Copy the `net.luminis.cmc-0.2.5.jar` to the `equinox_root/plugins` directory.
7. Download the Apache Felix file installation Version 3.0.2 bundle from the following web page: <http://felix.apache.org/site/index.html>. Copy the `org.apache.felix.fileinstall-3.0.2.jar` file to the `equinox_root/plugins` directory.
8. Create a configuration directory inside `equinox_root/plugins` directory; for example:

```
mkdir equinox_root/plugins/configuration
```

9. Create the following config.ini file in the equinox_root/plugins/ configuration directory, replacing *equinox_root* with the absolute path to your equinox_root directory and removing all trailing spaces after the backslash on each line. You must include a blank line at the end of the file; for example:

```
osgi.noShutdown=true
osgi.java.profile.bootdelegation=none
org.osgi.framework.bootdelegation=none
eclipse.ignoreApp=true
osgi.bundles=\
org.eclipse.osgi.services_3.2.100.v20100503.jar@1:start, \
org.eclipse.osgi.util_3.2.100.v20100503.jar@1:start, \
org.eclipse.equinox.cm_1.0.200.v20100520.jar@1:start, \
com.springsource.org.apache.commons.logging-1.1.1.jar@1:start, \
com.springsource.org.aopalliance-1.0.0.jar@1:start, \
org.springframework.aop-3.0.5.RELEASE.jar@1:start, \
org.springframework.asm-3.0.5.RELEASE.jar@1:start, \
org.springframework.beans-3.0.5.RELEASE.jar@1:start, \
org.springframework.context-3.0.5.RELEASE.jar@1:start, \
org.springframework.core-3.0.5.RELEASE.jar@1:start, \
org.springframework.expression-3.0.5.RELEASE.jar@1:start, \
org.apache.felix.fileinstall-3.0.2.jar@1:start, \
net.luminis.cmc-0.2.5.jar@1:start, \
gemini-blueprint-core-1.0.0.jar@1:start, \
gemini-blueprint-extender-1.0.0.jar@1:start, \
gemini-blueprint-io-1.0.0.jar@1:start
```

If you have already set up the environment, you can clean up the Equinox plug-in repository by removing the following directory: equinox_root\plugins\configuration\org.eclipse.osgi.

10. Run the following commands to start equinox console.

If you are running a different version of Equinox, then your JAR file name is different from the one in the following example:

```
java -jar plugins\org.eclipse.osgi_3.6.1.R36x_v20100806.jar -console
```

Installing eXtreme Scale bundles

WebSphere eXtreme Scale includes bundles that can be installed into an Eclipse Equinox OSGi framework. These bundles are required to start eXtreme Scale servers or use eXtreme Scale clients in OSGi. You can install the eXtreme Scale bundles using the Equinox console or using the config.ini configuration file.

Before you begin

This task assumes that you have installed the following products:

- Eclipse Equinox OSGi framework
- eXtreme Scale stand-alone client or server

About this task

eXtreme Scale includes two bundles. Only one of the following bundles is required in an OSGi framework:

objectgrid.jar

The server bundle is the objectgrid.jar file and is installed with the eXtreme Scale stand-alone server installation and is required for running eXtreme Scale servers and can also be used for running eXtreme Scale clients, or local, in-memory caches. The bundle ID for the objectgrid.jar file is com.ibm.websphere.xs.server_<version>, where the version is in the format: <Version>.<Release>.<Modification>. For example, the server bundle for this release is com.ibm.websphere.xs.server_8.5.0.

ogclient.jar

The ogclient.jar bundle is installed with the eXtreme Scale stand-alone and client installations and is used to run eXtreme Scale clients or local, in-memory caches. The bundle ID for ogclient.jar file is com.ibm.websphere.xs.client_<version>, where the version is in the format:

<Version>_<Release>_<Modification>. For example, the client bundle for this release is `com.ibm.websphere.xs.server_8.5.0`.

For more information about developing eXtreme Scale plug-ins, see the System APIs and Plug-ins topic.

Install the eXtreme Scale client or server bundle into the Eclipse Equinox OSGi framework using the Equinox console:

Procedure

1. Start the Eclipse Equinox framework with the console enabled; for example:

```
java_home/bin/java -jar <equinox_root>/plugins/  
org.eclipse.osgi_3.6.1.R36x_v20100806.jar -console
```

2. Install the eXtreme Scale client or server bundle in the Equinox console:

```
osgi> install file:///<path to bundle>
```

3. Equinox displays the bundle ID for the newly installed bundle:

```
Bundle id is 25
```

4. Start the bundle in the Equinox console, where <id> is the bundle ID assigned when the bundle was installed:

```
osgi> start <id>
```

5. Retrieve the service status in the Equinox console to verify that the bundle has started; for example:

```
osgi> ss
```

When the bundle starts successfully, the bundle displays the ACTIVE state; for example:

```
25      ACTIVE      com.ibm.websphere.xs.server_8.5.0
```

Install the eXtreme Scale client or server bundle into the Eclipse Equinox OSGi framework using the config.ini file:

Procedure

1. Copy the eXtreme Scale client or server (`objectgrid.jar` or `ogclient.jar`) bundle from the `<wxs_install_root>/ObjectGrid/lib` to the Eclipse Equinox plug-ins directory; for example: `<equinox_root>/plugins`

2. Edit the Eclipse Equinox `config.ini` configuration file, and add the bundle to the `osgi.bundles` property; for example:

```
osgi.bundles=\  
org.eclipse.osgi.services_3.2.100.v20100503.jar@1:start, \  
org.eclipse.osgi.util_3.2.100.v20100503.jar@1:start, \  
org.eclipse.equinox.cm_1.0.200.v20100520.jar@1:start, \  
objectgrid.jar@1:start
```

Important: Verify that a blank line exists after the last bundle name. Each bundle is separated by a comma.

3. Start the Eclipse Equinox framework with the console enabled; for example:

```
java_home/bin/java -jar <equinox_root>/plugins/  
org.eclipse.osgi_3.6.1.R36x_v20100806.jar -console
```

4. Retrieve the service status in the Equinox console to verify that the bundle has started:

```
osgi> ss
```

When the bundle starts successfully, the bundle displays the ACTIVE state; for example:

```
25      ACTIVE      com.ibm.websphere.xs.server_8.5.0
```

Results

The eXtreme Scale server or client bundle is installed and started in your Eclipse Equinox OSGi framework.

Running eXtreme Scale containers with non-dynamic plug-ins in an OSGi environment

If you do not need to use the dynamic capability of an OSGi environment, you can still take advantage of tighter coupling, declarative packaging, and service dependencies that the OSGi framework offers.

Before you begin

1. Develop your application using WebSphere eXtreme Scale APIs and plug-ins.
2. Package the application in one or more OSGi bundles with the appropriate import or export dependencies that are declared in one or more bundle manifests. Ensure that all classes or packages that are required for the plug-ins, agents, data objects, and so on, are exported.

About this task

With dynamic plug-ins, you can upgrade your plug-ins without stopping the grid. To use this capability, the original and new plug-ins must be compatible. If you do not need to update plug-ins, or can afford to stop the grid to upgrade them, then you may not need the complexity of dynamic plug-ins. However, there are still good reasons to run your eXtreme Scale application in an OSGi environment. These reasons include the tighter coupling, declarative package, service dependencies, and so on.

One concern with hosting the grid or client in an OSGi environment without using dynamic plug-ins (more specifically, without declaring the plug-ins using OSGi services) is how the eXtreme Scale bundle loads the plug-in classes. The eXtreme Scale bundle relies on OSGi services to load plug-in classes, which allows the bundle to invoke object methods on classes in other bundles without directly importing the packages of those classes.

When the plug-ins are not made available via OSGi services, the eXtreme Scale bundle must be able to load the plug-in classes directly. Rather than modifying the manifest of the eXtreme Scale bundle to import user classes and packages, create a bundle fragment that adds the necessary package imports. The fragment can also import the classes needed for other non-plug-in user classes, such as data objects and agent classes.

Procedure

1. Create an OSGi fragment that uses the eXtreme Scale bundle (client or server, depending on the intended deployment environment) as its host. The fragment declares dependencies (Import-Package) on all of the packages that one or more plug-ins must load. For example, if you are installing a serializer plug-in whose classes reside in the `com.mycompany.myapp.serializers` package and depends on classes in the `com.mycompany.myapp.common` package, then your fragment META-INF/MANIFEST.MF file resembles the following example:

```
Bundle-ManifestVersion: 2
Bundle-Name: Plug-in fragment for XS serializers
Bundle-SymbolicName: com.mycompany.myapp.myfragment; singleton=true
Bundle-Version: 1.0.0
Fragment-Host: com.ibm.websphere.xs.server; bundle-version=7.1.1
```

```
Manifest-Version: 1.0
Import-Package: com.mycompany.myapp.serializers,
               com.mycompany.myapp.common
...
```

This manifest must be packaged in a fragment JAR file, which in this example is `com.mycompany.myapp.myfragment_1.0.0.jar`.

2. Deploy both the newly created fragment, the eXtreme Scale bundle, and application bundles to your OSGi environment. Now, start the bundles.

Results

You can now test and run your application in the OSGi environment without using OSGi services to load user classes, such as plug-ins and agents.

Administering eXtreme Scale servers and applications in an OSGi environment

Use this topic to install the WebSphere eXtreme Scale server bundle, an optional fragment that allows loading of your application bundles and non-dynamic user classes, such as plug-ins, agents, data objects, and so on.

Before you begin

1. Install and start a supported OSGi framework. Currently Equinox is the only supported OSGi implementation. If your application uses Blueprint, make sure to install and start a supported Blueprint implementation. Apache Aries and Eclipse Gemini are both supported.
2. Open the OSGi console.

Procedure

1. Install the eXtreme Scale server bundle. You must know the file URL of the bundle Java archive (JAR) file. For example:

```
osgi> install file:///home/user1/myOsgiEnv/plugins/objectgrid.jar
Bundle id is 41
```

```
osgi>
```

The eXtreme Scale bundle is now installed, but not yet resolved.

2. If the eXtreme Scale server must load user classes directly, rather than using dynamic plug-ins exposed via OSGi services, then you must also install a user-developed fragment that either provides those classes or imports them. If you are using dynamic plug-ins and not using agents, you can skip this step. Here is an example of how to install a custom fragment:

```
osgi> install file:///home/user1/myOsgiEnv/plugins/myFragment.jar
Bundle id is 42
```

```
osgi> ss
```

```
Framework is launched.
```

```
id State      Bundle
...
41 INSTALLED  com.ibm.websphere.xs.server_7.1.1
42 INSTALLED  com.mycompany.myfragment_1.0.0
```

```
osgi>
```

Now the eXtreme Scale server bundle and the custom fragment that attaches to the bundle are both installed.

3. Start the eXtreme Scale server bundle; for example:

```
osgi> start 41
```

```
osgi> ss
```

```
Framework is launched.
```

```
id State      Bundle
...
41 ACTIVE     com.ibm.websphere.xs.server_7.1.1
                Fragments=42
42 RESOLVED   com.mycompany.myfragment_1.0.0
                Master=41
```

```
osgi>
```

4. Now install and start all user application bundles using the same previously mentioned commands. To start a grid on this server, the server and container definition must be declared using Blueprint, or the application must start the server and container programmatically from a bundle activator or some other mechanism.

Results

The eXtreme Scale server bundle and application are deployed, started, and ready to accept work.

Building and running eXtreme Scale dynamic plug-ins for use in an OSGi environment

All eXtreme Scale plug-ins can be configured for an OSGi environment. The primary benefit of dynamic plug-ins is that they allow you to upgrade them without shutting down the grid. This allows you to evolve an application without restarting the grid container processes.

About this task

WebSphere eXtreme Scale OSGi support allows you to deploy the product in an OSGi framework, such as Eclipse Equinox. Previously, if you wanted to update the plug-ins used by eXtreme Scale, you had to restart the Java virtual machine (JVM) to apply the new versions of the plug-ins. With the dynamic plug-in support provided by eXtreme Scale and the ability to update bundles that the OSGi framework provides, you can now update the plug-in classes without restarting the JVM. These plug-ins are exported by *bundles* as services. WebSphere eXtreme Scale accesses the service by looking up the OSGi registry. In the OSGi service platform, a bundle is a Java archive (JAR) file that contains Java code, resources, and a manifest that describes the bundle and its dependencies. The bundle is the unit of deployment for an application.

Procedure

1. Build eXtreme Scale dynamic plug-ins.
2. Configure eXtreme Scale plug-ins with OSGi Blueprint.
3. Install and starting OSGi-enabled plug-ins.

Building eXtreme Scale dynamic plug-ins

WebSphere eXtreme Scale includes ObjectGrid and BackingMap plug-ins. These plug-ins are implemented in Java and are configured using the ObjectGrid descriptor XML file. To create a dynamic plug-in that can be dynamically upgraded, they need to be aware of ObjectGrid and BackingMap life cycle events because they might need to complete some actions during an update. Enhancing a plug-in bundle with life cycle callback methods, event listeners, or both allows the plug-in to complete those actions at the appropriate times.

Before you begin

This topic assumes that you have built the appropriate plug-in. For more information about developing eXtreme Scale plug-ins, see the System APIs and plug-ins topic.

About this task

All eXtreme Scale plug-ins apply to either a BackingMap or ObjectGrid instance. Many plug-ins also interact with other plug-ins. For example, a Loader and TransactionCallback plug-in work together to properly interact with a database transaction and the various database JDBC calls. Some plug-ins might also need to cache configuration data from other plug-ins to improve performance.

The BackingMapLifecycleListener and ObjectGridLifecycleListener plug-ins provide life cycle operations for the respective BackingMap and ObjectGrid instances. This process allows plug-ins to be notified when the parent BackingMap or ObjectGrid and their respective plug-ins might be changed. BackingMap plug-ins implement the BackingMapLifecycleListener interface, and ObjectGrid plug-ins implement the ObjectGridLifecycleListener interface. These plug-ins are automatically invoked when the life cycle of the parent BackingMap or ObjectGrid changes. For more information about life cycle plug-ins, see the “Managing plug-in life cycles” on page 279 topic.

You can expect to enhance bundles using the life cycle methods or event listeners in the following common tasks:

- Starting and stopping resources, such as threads or messaging subscribers.
- Specifying that a notification occur when peer plug-ins have been updated, allowing direct access to the plug-in and detecting any changes.

Whenever you access another plug-in directly, access that plug-in through the OSGi container to ensure that all parts of the system reference the correct plug-in. If, for example, some component in the application directly references, or caches, an instance of a plug-in, it will maintain its reference to that version of the plug-in, even after that plug-in has been dynamically updated. This behavior can cause application-related problems as well as memory leaks. Therefore, write code that depends on dynamic plug-ins that obtain its reference using OSGi, getService() semantics. If the application must cache one or more plug-ins, it listens for life cycle events using ObjectGridLifecycleListener and BackingMapLifecycleListener interfaces. The application must also be able to refresh its cache when necessary, in a thread safe manner.

All eXtreme Scale plug-ins used with OSGi must also implement the respective BackingMapPlugin or ObjectGridPlugin interfaces. New plug-ins such as the MapSerializerPlugin interface enforce this practice. These interfaces provide the

eXtreme Scale runtime environment and OSGi a consistent interface for injecting state into the plug-in and controlling its life cycle.

Use this task to specify that a notification occurs when peer plug-ins are updated, you might create a listener factory that produces a listener instance.

Procedure

- Update the ObjectGrid plug-in class to implement the ObjectGridPlugin interface. This interface includes methods that allow eXtreme Scale to initialize, set the ObjectGrid instance and destroy the plug-in. See the following code example:

```
package com.mycompany;
import com.ibm.websphere.objectgrid.plugins.ObjectGridPlugin;
...

public class MyTranCallback implements TransactionCallback, ObjectGridPlugin {

    private ObjectGrid og = null;

    private enum State {
        NEW, INITIALIZED, DESTROYED
    }

    private State state = State.NEW;

    public void setObjectGrid(ObjectGrid grid) {
        this.og = grid;
    }

    public ObjectGrid getObjectGrid() {
        return this.og;
    }
    void initialize() {
        // Handle any plug-in initialization here. This is called by
        // eXtreme Scale, and not the OSGi bean manager.
        state = State.INITIALIZED;
    }
    boolean isInitialized() {
        return state == State.INITIALIZED;
    }

    public void destroy() {
        // Destroy the plug-in and release any resources. This
        // can be called by the OSGi Bean Manager or by eXtreme Scale.
        state = State.DESTROYED;
    }

    public boolean isDestroyed() {
        return state == State.DESTROYED;
    }
}
```

- Update the ObjectGrid plug-in class to implement the ObjectGridLifecycleListener interface. See the following code example:

```
package com.mycompany;
import com.ibm.websphere.objectgrid.plugins.ObjectGridLifecycleListener;
import com.ibm.websphere.objectgrid.plugins.ObjectGridLifecycleListener.LifecycleEvent;
...

public class MyTranCallback implements TransactionCallback, ObjectGridPlugin, ObjectGridLifecycleListener{
    public void objectGridStateChanged(LifecycleEvent event) {
        switch(event.getState()) {
            case NEW:
            case DESTROYED:
            case DESTROYING:
            case INITIALIZING:
                break;
            case INITIALIZED:
                // Lookup a Loader or MapSerializerPlugin using
                // OSGi or directly from the ObjectGrid instance.
                lookupOtherPlugins()
                break;
            case STARTING:
            case PRELOAD:
                break;
            case ONLINE:
                if (event.isWritable()) {
                    startupProcessingForPrimary();
                } else {
                    startupProcessingForReplica();
                }
                break;
            case QUIESCENCE:
                if (event.isWritable()) {
```

```

        quiesceProcessingForPrimary();
    } else {
        quiesceProcessingForReplica();
    }
    break;
case OFFLINE:
    shutdownShardComponents();
    break;
}
}
...
}

```

- Update a BackingMap plug-in. Update the BackingMap plug-in class to implement the BackingMap plu-in interface. This interface includes methods that allow eXtreme Scale to initialize, set the BackingMap instance, and destroy the plug-in. See the following code example:

```

package com.mycompany;
import com.ibm.websphere.objectgrid.plugins.BackingMapPlugin;
...

public class MyLoader implements Loader, BackingMapPlugin {

    private BackingMap bmap = null;

    private enum State {
        NEW, INITIALIZED, DESTROYED
    }

    private State state = State.NEW;

    public void setBackingMap(BackingMap map) {
        this.bmap = map;
    }

    public BackingMap getBackingMap() {
        return this.bmap;
    }
    void initialize() {
        // Handle any plug-in initialization here. This is called by
        // eXtreme Scale, and not the OSGi bean manager.
        state = State.INITIALIZED;
    }
    boolean isInitialized() {
        return state == State.INITIALIZED;
    }

    public void destroy() {
        // Destroy the plug-in and release any resources. This
        // can be called by the OSGi Bean Manager or by eXtreme Scale.
        state = State.DESTROYED;
    }

    public boolean isDestroyed() {
        return state == State.DESTROYED;
    }
}

```

- Update the BackingMap plug-in class to implement the BackingMapLifecycleListener interface. See the following code example:

```

package com.mycompany;

import com.ibm.websphere.objectgrid.plugins.BackingMapLifecycleListener;
import com.ibm.websphere.objectgrid.plugins.BackingMapLifecycleListener.LifecycleEvent;
...

public class MyLoader implements Loader, ObjectGridPlugin, ObjectGridLifecycleListener{
    ...
    public void backingMapStateChanged(LifecycleEvent event) {
        switch(event.getState()) {
            case NEW:
            case DESTROYED:
            case DESTROYING:
            case INITIALIZING:
                break;
            case INITIALIZED:
                // Lookup a MapSerializerPlugin using
                // OSGi or directly from the ObjectGrid instance.
                lookupOtherPlugins()
                break;
            case STARTING:
            case PRELOAD:
                break;
            case ONLINE:
                if (event.isWritable()) {
                    startupProcessingForPrimary();
                } else {
                    startupProcessingForReplica();
                }
                break;
        }
    }
}

```

```

        case QUIESCE:
            if (event.isWritable()) {
                quiesceProcessingForPrimary();
            } else {
                quiesceProcessingForReplica();
            }
            break;
        case OFFLINE:
            shutdownShardComponents();
            break;
    }
    ...
}

```

Results

By implementing the `ObjectGridPlugin` or `BackingMapPlugin` interface, eXtreme Scale can control the life cycle of your plug-in at the right times.

By implementing the `ObjectGridLifecycleListener` or `BackingMapLifecycleListener` interface, the plug-in is automatically registered as a listener of the associated `ObjectGrid` or `BackingMap` life cycle events. The `INITIALIZING` event is used to signal that all of the `ObjectGrid` and `BackingMap` plug-ins have been initialized and are available for lookup and use. The `ONLINE` event is used to signal that the `ObjectGrid` is online and ready to start processing events.

Configuring eXtreme Scale plug-ins with OSGi Blueprint

All eXtreme Scale `ObjectGrid` and `BackingMap` plug-ins can be defined as OSGi beans and services using the OSGi Blueprint Service available with Eclipse Gemini or Apache Aries.

Before you begin

Before you can configure your plug-ins as OSGi services, you must first package your plug-ins in an OSGi bundle, and understand the fundamental principles of the required plug-ins. The bundle must import the WebSphere eXtreme Scale server or client packages and other dependent packages required by the plug-ins, or create a bundle dependency on the eXtreme Scale server or client bundles. This topic describes how to configure the Blueprint XML to create plug-in beans and expose them as OSGi services for eXtreme Scale to use.

About this task

Beans and services are defined in a Blueprint XML file, and the Blueprint container discovers, creates, and wires the beans together and exposes them as services. The process makes the beans available to other OSGi bundles, including the eXtreme Scale server and client bundles.

When creating custom plug-in services for use with eXtreme Scale, the bundle that is to host the plug-ins, must be configured to use Blueprint. In addition, a Blueprint XML file must be created and stored within the bundle. Read about building OSGi applications with the Blueprint Container specification for a general understanding of the specification.

Procedure

1. Create a Blueprint XML file. You can name the file anything. However, you must include the blueprint namespace:

```

<?xml version="1.0" encoding="UTF-8"?>
<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0">
...
</blueprint>

```


2. Create bean definitions in the Blueprint XML file for each eXtreme Scale plug-in.

Beans are defined using the `<bean>` element and can be wired to other bean references and can include initialization parameters.

Important: When defining a bean, you must use the correct scope. Blueprint supports the singleton and prototype scopes. eXtreme Scale also supports a custom shard scope.

Define most eXtreme Scale plug-ins as prototype or shard-scoped beans, since all of the beans must be unique for each ObjectGrid shard or BackingMap instance it is associated with. Shard-scoped beans can be useful when using the beans in other contexts to allow retrieving the correct instance.

To define a prototype-scoped bean, use the `scope="prototype"` attribute on the bean:

```
<bean id="myPluginBean" class="com.mycompany.MyBean" scope="prototype">
...
</bean>
```

To define a shard-scoped bean, you must add the `objectgrid` namespace to the XML schema, and use the `scope="objectgrid:shard"` attribute on the bean:

```
<?xml version="1.0" encoding="UTF-8"?>

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
           xmlns:objectgrid="http://www.ibm.com/schema/objectgrid"

           xsi:schemaLocation="http://www.ibm.com/schema/objectgrid
                               http://www.ibm.com/schema/objectgrid/objectgrid.xsd">

  <bean id="myPluginBean" class="com.mycompany.MyBean"
        scope="objectgrid:shard">
    ...
  </bean>
...

```

3. Create `PluginServiceFactory` bean definitions for each plug-in bean. All eXtreme Scale beans must have a `PluginServiceFactory` bean defined so that the correct bean scope can be applied. eXtreme Scale includes a `BlueprintServiceFactory` that you can use. It includes two properties that must be set. You must set the `blueprintContainer` property to the `blueprintContainer` reference, and the `beanId` property must be set to the bean identifier name. When eXtreme Scale looks up the service to instantiate the appropriate beans, the server looks up the bean component instance using the Blueprint container.

```
bean id="myPluginBeanFactory"
    class="com.ibm.websphere.objectgrid.plugins.osgi.BluePrintServiceFactory">
  <property name="blueprintContainer" ref="blueprintContainer" />
  <property name="beanId" value="myPluginBean" />
</bean>
```

4. Create a service manager for each `PluginServiceFactory` bean. Each service manager exposes the `PluginServiceFactory` bean, using the `<service>` element. The service element identifies the name to expose to OSGi, the reference to the `PluginServiceFactory` bean, the interface to expose, and the ranking of the service. eXtreme Scale uses the service manager ranking to perform service upgrades when the eXtreme Scale grid is active. If the ranking is not specified, the OSGi framework assumes a ranking of 0. Read about updating service rankings for more information.

Blueprint includes several options for configuring service managers. To define a simple service manager for a `PluginServiceFactory` bean, create a `<service>` element for each `PluginServiceFactory` bean:

```
<service ref="myPluginBeanFactory"
  interface="com.ibm.websphere.objectgrid.plugins.osgi.PluginServiceFactory"
  ranking="1">
</service>
```

5. Store the Blueprint XML file in the plug-ins bundle. The Blueprint XML file must be stored in the OSGI-INF/blueprint directory for the Blueprint container to be discovered.

To store the Blueprint XML file in a different directory, you must specify the following Bundle-Blueprint manifest header:

```
Bundle-Blueprint: OSGI-INF/blueprint.xml
```

Results

The eXtreme Scale plug-ins are now configured to be exposed in an OSGi Blueprint container. In addition, the ObjectGrid descriptor XML file is configured to reference the plug-ins using the OSGi Blueprint service.

Installing and starting OSGi-enabled plug-ins

In this task, you install the dynamic plug-in bundle into the OSGi framework. Then, you start the plug-in.

Before you begin

This topic assumes that the following tasks have been completed:

- The eXtreme Scale server or client bundle has been installed into the Eclipse Equinox OSGi framework. See “Installing eXtreme Scale bundles” on page 38.
- One or more dynamic BackingMap or ObjectGrid plug-ins have been implemented. See “Building eXtreme Scale dynamic plug-ins” on page 43.
- The dynamic plug-ins have been packaged as OSGi services in OSGi bundles.

About this task

This task describes how to install the bundle using the Eclipse Equinox console. The bundle can be installed using several different methods, including modifying the `config.ini` configuration file. Products that embed Eclipse Equinox include alternative methods for managing bundles. For more information on how to add bundles in the `config.ini` file in Eclipse Equinox, see the Eclipse runtime options.

OSGi allows bundles to be started that have duplicate services. WebSphere eXtreme Scale uses the latest service ranking. When starting multiple OSGi frameworks in an eXtreme Scale data grid, you must make sure that the correct service rankings are started on each server. Failure to do so causes the grid to be started with a mixture of different versions.

To see which versions are in-use by the data grid, use the `xscmd` utility to check the current and available rankings. For more information about available service rankings see Updating OSGi services for eXtreme Scale plug-ins with `xscmd`.

Procedure

Install the plug-in bundle into the Eclipse Equinox OSGi framework using the OSGi console.

1. Start the Eclipse Equinox framework with the console enabled; for example:

```
<java_home>/bin/java -jar <equinox_root>/plugins/org.eclipse.osgi_3.6.1.R36x_v20100806.jar -console
```
2. Install the plug-in bundle in the Equinox console.

```
osgi> install file:///<path to bundle>
```

Equinox displays the bundle ID for the newly installed bundle:

```
Bundle id is 17
```

3. Enter the following line to start the bundle in the Equinox console, where <id> is the bundle ID assigned when the bundle was installed:

```
osgi> start <id>
```

4. Retrieve the service status in the Equinox console to verify that the bundle has started:

```
osgi> ss
```

When the bundle has started successfully, the bundle displays the ACTIVE state; for example:

```
17      ACTIVE      com.mycompany.plugin.bundle_VRM
```

Install the plug-in bundle into the Eclipse Equinox OSGi framework using the config.ini file.

5. Copy the plug-in bundle into the Eclipse Equinox plug-ins directory; for example:

```
<equinox_root>/plugins
```

6. Edit the Eclipse Equinox config.ini configuration file, and add the bundle to the osgi.bundles property; for example:

```
osgi.bundles=\
org.eclipse.osgi.services_3.2.100.v20100503.jar@1:start, \
org.eclipse.osgi.util_3.2.100.v20100503.jar@1:start, \
org.eclipse.equinox.cm_1.0.200.v20100520.jar@1:start, \
com.mycompany.plugin.bundle_VRM.jar@1:start
```

Important: Verify there is a blank line after the last bundle name. Each bundle is separated by a comma.

7. Start the Eclipse Equinox framework with the console enabled; for example:

```
<java_home>/bin/java -jar <equinox_root>/plugins/org.eclipse.osgi_3.6.1.R36x_v20100806.jar -console
```

8. Retrieve the service status in the Equinox console to verify that the bundle has started; for example:

```
osgi> ss
```

When the bundle has started successfully, the bundle displays the ACTIVE state; for example:

```
17      ACTIVE      com.mycompany.plugin.bundle_VRM
```

Results

The plug-in bundle is now installed and started. The eXtreme Scale container or client can now be started. For more information on developing eXtreme Scale plug-ins, see the System APIs and Plug-ins topic.

Running eXtreme Scale containers with dynamic plug-ins in an OSGi environment

If your application is hosted in the Eclipse Equinox OSGi framework with Eclipse Gemini or Apache Aries, then you can use this task to help you install and configure your WebSphere eXtreme Scale application in OSGi.

Before you begin

Before you start this task, be sure to complete the following tasks:

- Install the Eclipse Equinox OSGi framework with Eclipse Gemini
- Build and run eXtreme Scale dynamic plug-ins for use in an OSGi environment

About this task

With dynamic plug-ins, you can dynamically upgrade the plug-in while the grid is still active. This allows you to update an application without restarting the grid container processes. For more information about developing eXtreme Scale plug-ins, see System APIs and Plug-ins.

Procedure

1. Configure OSGi-enabled plug-ins using the ObjectGrid descriptor XML file.
2. Start eXtreme Scale container servers using the Eclipse Equinox OSGi framework.
3. Administer OSGi services for eXtreme Scale plug-ins with the xscmd utility.
4. Configure servers with OSGi Blueprint.

Configuring OSGi-enabled plug-ins using the ObjectGrid descriptor XML file

In this task, you add existing OSGi services to a descriptor XML file so that WebSphere eXtreme Scale containers can recognize and load the OSGi-enabled plug-ins correctly.

Before you begin

To configure your plug-ins, be sure to:

- Create your package, and enable dynamic plug-ins for OSGi deployment.
- Have the names of the OSGi services that represent your plug-ins available.

About this task

You have created an OSGi service to wrap your plug-in. Now, these services must be defined in the `objectgrid.xml` file so that eXtreme Scale containers can load and configure the plug-in or plug-ins successfully.

Procedure

1. Any grid-specific plug-ins, such as `TransactionCallback`, must be specified under the `objectGrid` element. See the following example from the `objectgrid.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>

<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">

  <objectGrids>
    <objectGrid name="MyGrid" txTimeout="60">
      <bean id="myTranCallback" osgiService="myTranCallbackFactory"/>
      ...
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

```

    ...
  </objectGrids>
  ...
</objectGridConfig>

```

Important: The `osgiService` attribute value must match the `ref` attribute value that is specified in the blueprint XML file, where the service was defined for `myTranCallback PluginServiceFactory`.

2. Any map-specific plug-ins, such as loaders or serializers, for example, must be specified in the `backingMapPluginCollections` element and referenced from the `backingMap` element. See the following example from the `objectgrid.xml` file:

```

<?xml version="1.0" encoding="UTF-8"?>

objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="MyGrid" txTimeout="60">
      <backingMap name="MyMap1" lockStrategy="PESSIMISTIC"
        copyMode="COPY_TO_BYTES" nullValuesSupported="false"
        pluginCollectionRef="myPluginCollectionRef1"/>
      <backingMap name="MyMap2" lockStrategy="PESSIMISTIC"
        copyMode="COPY_TO_BYTES" nullValuesSupported="false"
        pluginCollectionRef="myPluginCollectionRef2"/>
      ...
    </objectGrid>
    ...
  </objectGrids>
  ...
  <backingMapPluginCollections>
    <backingMapPluginCollection id="myPluginCollectionRef1">
      <bean id="MapSerializerPlugin" osgiService="mySerializerFactory"/>
    </backingMapPluginCollection>
    <backingMapPluginCollection id="myPluginCollectionRef2">
      <bean id="MapSerializerPlugin" osgiService="myOtherSerializerFactory"/>
      <bean id="Loader" osgiService="myLoader"/>
    </backingMapPluginCollection>
    ...
  </backingMapPluginCollections>
  ...
</objectGridConfig>

```

Results

The `objectgrid.xml` file in this example tells eXtreme Scale to create a grid called `MyGrid` with two maps, `MyMap1` and `MyMap2`. The `MyMap1` map uses the serializer wrapped by the OSGi service, `mySerializerFactory`. The `MyMap2` map uses a serializer from the OSGi service, `myOtherSerializerFactory`, and a loader from the OSGi service, `myLoader`.

Starting eXtreme Scale servers using the Eclipse Equinox OSGi framework

WebSphere eXtreme Scale container servers can be started in an Eclipse Equinox OSGi framework using several methods.

Before you begin

Before you can start an eXtreme Scale container, you must have completed the following tasks:

1. The WebSphere eXtreme Scale server bundle must be installed into Eclipse Equinox.
2. Your application must be packaged as an OSGi bundle.
3. Your WebSphere eXtreme Scale plug-ins (if any) must be packaged as an OSGi bundle. They can be bundled in the same bundle as your application or as separate bundles.

About this task

This task describes how to start an eXtreme Scale container server in an Eclipse Equinox OSGi framework. You can use any of the following methods to start container servers using the Eclipse Equinox implementation:

- OSGi Blueprint service

You can include all configuration and metadata in an OSGi bundle. See the following image to understand the Eclipse Equinox process for this method:

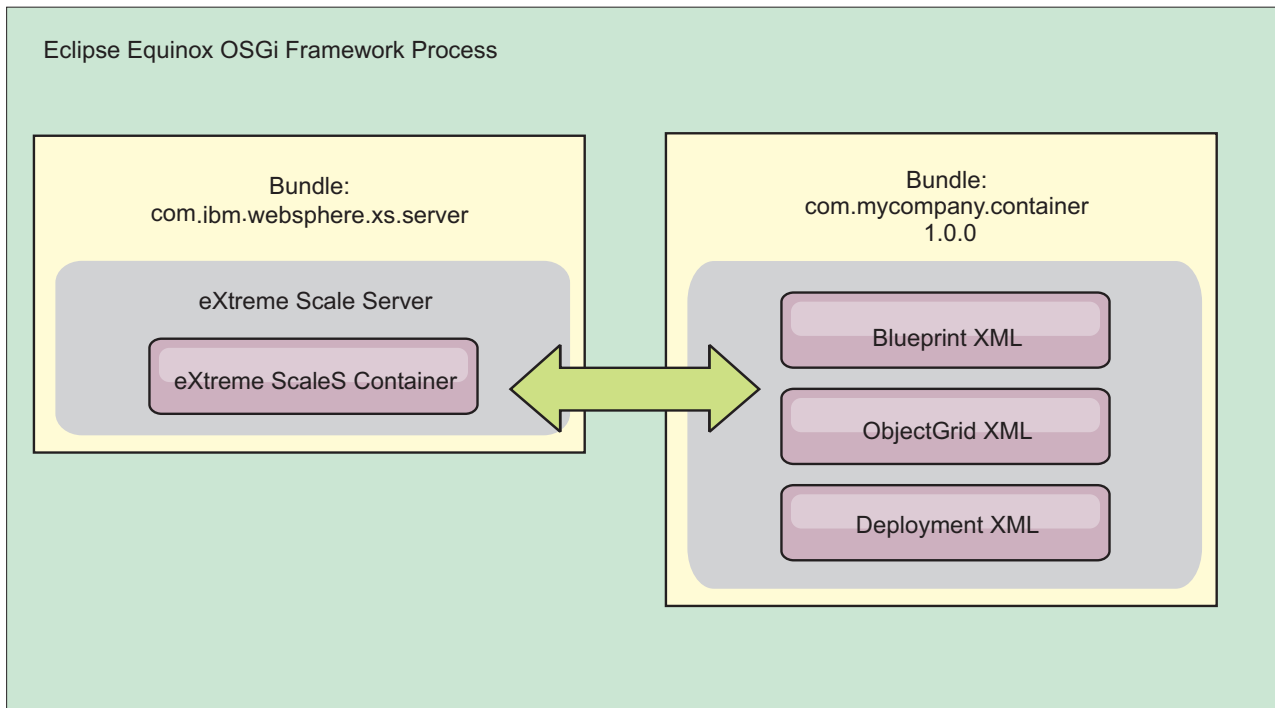


Figure 3. Eclipse Equinox process for including all configuration and metadata in an OSGi bundle

- OSGi Configuration Admin service

You can specify configuration and metadata outside of an OSGi bundle. See the following image to understand the Eclipse Equinox process for this method:

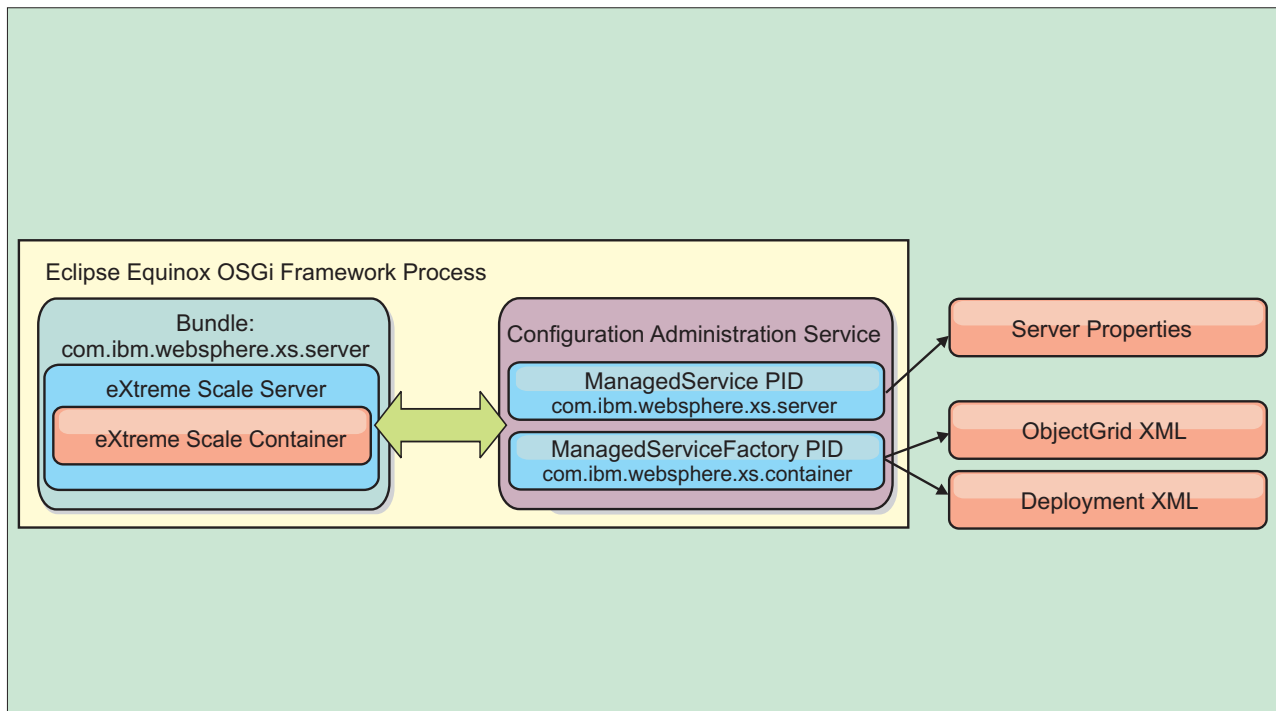


Figure 4. Eclipse Equinox process for specify configuration and metadata outside of an OSGi bundle

- Programmatically
Supports customized configuration solutions.

In each case, an eXtreme Scale server singleton is configured and one or more containers are configured.

The eXtreme Scale server bundle, `objectgrid.jar`, includes all of the required libraries to start and run an eXtreme Scale grid container in an OSGi framework. The server runtime environment communicates with user-supplied plug-ins and data objects using the OSGi service manager.

Important: After an eXtreme Scale server bundle is started and the eXtreme Scale server is initialized, it cannot be restarted. The Eclipse Equinox process must be restarted to restart an eXtreme Scale server.

You can use eXtreme Scale support for Spring namespace to configure eXtreme Scale container servers in a Blueprint XML file. When the server and container XML elements are added to the Blueprint XML file, the eXtreme Scale namespace handler automatically starts a container server using the parameters that are defined in the Blueprint XML file when the bundle is started. The handler stops the container when the bundle is stopped.

To configure eXtreme Scale container servers with Blueprint XML, complete the following steps:

Procedure

- Start an eXtreme Scale container server using OSGi blueprint.
 1. Create a container bundle.
 2. Install the container bundle into the Eclipse Equinox OSGi framework. See "Installing and starting OSGi-enabled plug-ins" on page 48.

3. Start the container bundle.
- Start an eXtreme Scale container server using OSGi configuration admin.
 1. Configure the server and container using config admin.
 2. When the eXtreme Scale server bundle is started, or the persistent identifiers are created with config admin, the server and container automatically start.
 - Start an eXtreme Scale container server using the ServerFactory API. See the server API documentation.
 1. Create an OSGi bundle activator class, and use the eXtreme Scale ServerFactory API to start a server.

Administering OSGi-enabled services using the xscmd utility

You can use the **xscmd** utility to complete administrator tasks, such as viewing services and their rankings that are being used by each container, and updating the runtime environment to use new versions of the bundles.

About this task

With the Eclipse Equinox OSGi framework, you can install multiple versions of the same bundle, and you can update those bundles during run time. WebSphere eXtreme Scale is a distributed environment that runs the container servers in many OSGi framework instances.

Administrators are responsible for manually copying, installing, and starting bundles into the OSGi framework. eXtreme Scale includes an OSGi ServiceTrackerCustomizer to track any services that have been identified as eXtreme Scale plug-ins in the ObjectGrid descriptor XML file. Use the **xscmd** utility to validate which version of the plug-in is used, which versions are available to be used, and to perform bundle upgrades.

eXtreme Scale uses the service ranking number to identify the version of each service. When two or more services are loaded with the same reference, eXtreme Scale automatically uses the service with the highest ranking.

Procedure

- Run the **osgiCurrent** command, and verify that each eXtreme Scale server is using the correct plug-in service ranking.

Since eXtreme Scale automatically chooses the service reference with the highest ranking, it is possible that the data grid may start with multiple rankings of a plug-in service.

If the command detects a mismatch of rankings or if it is unable to find a service, a non-zero error level is set. If the command completed successfully then the error level is set to 0.

The following example shows the output of the **osgiCurrent** command when two plug-ins are installed in the same grid on four servers. The loaderPlugin plug-in is using ranking 1, and the txCallbackPlugin is using ranking 2.

```
OSGi Service Name Current Ranking ObjectGrid Name MapSet Name Server Name
-----
```

loaderPlugin	1	MyGrid	MapSetA	server1
loaderPlugin	1	MyGrid	MapSetA	server2
loaderPlugin	1	MyGrid	MapSetA	server3
loaderPlugin	1	MyGrid	MapSetA	server4
txCallbackPlugin	2	MyGrid	MapSetA	server1
txCallbackPlugin	2	MyGrid	MapSetA	server2
txCallbackPlugin	2	MyGrid	MapSetA	server3
txCallbackPlugin	2	MyGrid	MapSetA	server4

The following example shows the output of the **osgiCurrent** command when server2 was started with a newer ranking of the loaderPlugin:

```
OSGi Service Name Current Ranking ObjectGrid Name MapSet Name Server Name
-----
loaderPlugin      1           MyGrid      MapSetA     server1
loaderPlugin      2           MyGrid      MapSetA     server2
loaderPlugin      1           MyGrid      MapSetA     server3
loaderPlugin      1           MyGrid      MapSetA     server4
txCallbackPlugin  2           MyGrid      MapSetA     server1
txCallbackPlugin  2           MyGrid      MapSetA     server2
txCallbackPlugin  2           MyGrid      MapSetA     server3
txCallbackPlugin  2           MyGrid      MapSetA     server4
```

- Run the **osgiAll** command to verify that the plug-in services have been correctly started on each eXtreme Scale container server.

When bundles start that contain services that an ObjectGrid configuration is referencing, the eXtreme Scale runtime environment automatically tracks the plug-in, but does not immediately use it. The **osgiAll** command shows which plug-ins are available for each server.

When run without any parameters, all services are shown for all grids and servers. Additional filters, including the **-serviceName** <service_name> filter can be specified to limit the output to a single service or a subset of the data grid.

The following example shows the output of the **osgiAll** command when two plug-ins are started on two servers. The loaderPlugin has both rankings 1 and 2 started and the txCallbackPlugin has ranking 1 started. The summary message at the end of the output confirms that both servers see the same service rankings:

```
Server: server1
  OSGi Service Name Available Rankings
  -----
  loaderPlugin      1, 2
  txCallbackPlugin  1

Server: server2
  OSGi Service Name Available Rankings
  -----
  loaderPlugin      1, 2
  txCallbackPlugin  1
```

Summary - All servers have the same service rankings.

The following example shows the output of the **osgiAll** command when the bundle that includes the loaderPlugin with ranking 1 is stopped on server1. The summary message at the bottom of the output confirms that server1 is now missing the loaderPlugin with ranking 1:

```
Server: server1
  OSGi Service Name Available Rankings
  -----
  loaderPlugin      2
  txCallbackPlugin  1

Server: server2
  OSGi Service Name Available Rankings
  -----
  loaderPlugin      1, 2
  txCallbackPlugin  1
```

Summary - The following servers are missing service rankings:

```
Server OSGi Service Name Missing Rankings
-----
server1 loaderPlugin      1
```

The following example shows the output if the service name is specified with the **-sn** argument, but the service does not exist:

```
Server: server2
  OSGi Service Name Available Rankings
  -----
  invalidPlugin      No service found
```

```
Server: server1
  OSGi Service Name Available Rankings
  -----
  invalidPlugin      No service found
```

Summary - All servers have the same service rankings.

- Run the **osgiCheck** command to check sets of plug-in services and rankings to see if they are available.

The **osgiCheck** command accepts one or more sets of service rankings in the form: `-serviceRankings <service name>;<ranking>[,<serviceName>;<ranking>]`

When the rankings are all available, the method returns with an error level of 0. If one or more rankings are not available, a non-zero error level is set. A table of all of the servers that do not include the specified service rankings is displayed. Additional filters can be used to limit the service check to a subset of the available servers in the eXtreme Scale domain.

For example, if the specified ranking or service is absent, the following message is displayed:

```
Server  OSGi Service Unavailable Rankings
-----
server1 loaderPlugin 3
server2 loaderPlugin 3
```

- Run the **osgiUpdate** command to update the ranking of one or more plug-ins for all servers in a single ObjectGrid and MapSet in a single operation.

The command accepts one or more sets of service rankings in the form: `-serviceRankings <service name>;<ranking>[,<serviceName>;<ranking>] -g <grid name> -ms <mapset name>`

With this command, you can complete the following operations:

- Verify that the specified services are available for updating on each of the servers.
- Change the state of the grid to offline using the StateManager interface. See Managing ObjectGrid availability for more information. This process quiesces the grid and waits until any running transactions have completed and prevents any new transactions from starting. This process also signals any ObjectGridLifecycleListener and BackingMapLifecycleListener plug-ins to discontinue any transactional activity. See “Plug-ins for providing event listeners” on page 296 for information about event listener plug-ins.
- Update each eXtreme Scale container running in an OSGi framework to use the new service versions.
- Changes the state of the grid to online, allowing transactions to continue.

The update process is idempotent so that if a client fails to complete any one task, it results in the operation being rolled back. If a client is unable to perform the rollback or is interrupted during the update process, the same command can be issued again, and it continues at the appropriate step.

If the client is unable to continue, and the process is restarted from another client, use the `-force` option to allow the client to perform the update. The **osgiUpdate** command prevents multiple clients from updating the same map set concurrently. For more details about the **osgiUpdate** command, see Updating OSGi services for eXtreme Scale plug-ins with **xscmd**.

Configuring servers with OSGi Blueprint

You can configure WebSphere eXtreme Scale container servers using an OSGi blueprint XML file, allowing simplified packaging and development of self-contained server bundles.

Before you begin

This topic assumes that the following tasks have been completed:

- The Eclipse Equinox OSGi framework has been installed and started with either the Eclipse Gemini or Apache Aries blueprint container.
- The eXtreme Scale server bundle has been installed and started.
- The eXtreme Scale dynamic plug-ins bundle has been created.
- The eXtreme Scale ObjectGrid descriptor XML file and deployment policy XML file have been created.

About this task

This task describes how to configure an eXtreme Scale server with a container using a blueprint XML file. The result of the procedure is a container bundle. When the container bundle is started, the eXtreme Scale server bundle will track the bundle, parse the server XML and start a server and container.

A container bundle can optionally be combined with the application and eXtreme Scale plug-ins when dynamic plug-in updates are not required or the plug-ins do not support dynamic updating.

Procedure

1. Create a Blueprint XML file with the `objectgrid` namespace included. You can name the file anything. However, it must include the blueprint namespace:

```
<?xml version="1.0" encoding="UTF-8"?>

<blueprint xmlns="http://www.osgi.org/xmlns/blueprint/v1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:objectgrid="http://www.ibm.com/schema/objectgrid"
  xsi:schemaLocation="http://www.ibm.com/schema/objectgrid
    http://www.ibm.com/schema/objectgrid/objectgrid.xsd">
  ...
</blueprint>
```

2. Add the XML definition for the eXtreme Scale server with the appropriate server properties. See the Spring descriptor XML file for details on all available configuration properties. See the following example of the XML definition:

```
<objectgrid:server id="xsServer" tracespec="ObjectGridOSGi=all=enabled"
  tracefile="logs/osgi/wxssserver/trace.log" jmxport="1199" listenerPort="2909">
  <objectgrid:catalog host="catserver1.mycompany.com" port="2809" />
  <objectgrid:catalog host="catserver2.mycompany.com" port="2809" />
</objectgrid:server>
```

3. Add the XML definition for the eXtreme Scale container with the reference to the server definition and the ObjectGrid descriptor XML and ObjectGrid deployment XML files embedded in the bundle; for example:

```
<objectgrid:container id="container"
  objectgridxml="/META-INF/objectGrid.xml"
  deploymentxml="/META-INF/objectGridDeployment.xml"
  server="xsServer" />
```

4. Store the Blueprint XML file in the container bundle. The Blueprint XML must be stored in the `OSGI-INF/blueprint` directory for the Blueprint container to be found.

To store the Blueprint XML in a different directory, you must specify the Bundle-Blueprint manifest header; for example:

```
Bundle-Blueprint: OSGI-INF/blueprint.xml
```

5. Package the files into a single bundle JAR file. See the following example of a bundle directory hierarchy:

```
MyBundle.jar
  /META-INF/manifest.mf
  /META-INF/objectGrid.xml
  /META-INF/objectGridDeployment.xml
  /OSGI-INF/blueprint/blueprint.xml
```

Results

An eXtreme Scale container bundle is now created and can be installed in Eclipse Equinox. When the container bundle is started, the eXtreme Scale server runtime environment in the eXtreme Scale server bundle, will automatically start the singleton eXtreme Scale server using the parameters defined in the bundle, and starts a container server. The bundle can be stopped and started, which results in the container stopping and starting. The server is a singleton and does not stop when the bundle is started the first time.

Using JCA to connect transactional applications to eXtreme Scale clients

The following scenario is about connecting clients to applications that participate in transactions.

Before you begin

Read the Transaction processing in the Java EE applications overview topic to learn more about transaction support.

About this task

The Java EE Connector Architecture (JCA) provides support for clients that are using Java Transaction API (JTA). Through JTA, client management is simplified and accomplished using Java Platform, Enterprise Edition (Java EE). The JCA specification also supports resource adapters that you can use to connect applications to eXtreme Scale clients. A resource adapter is a system-level software driver that a Java application uses to connect to an enterprise information system (EIS). A resource adapter plugs into an application server and provides connectivity between the EIS, the application server, and the enterprise application. WebSphere eXtreme Scale provides its own resource adapter, which you can install without any required configuration.

As with previous versions of the product, you can use transactions to process a single unit of work to the data grid. With the support of JCA, when you commit those transactions you can enlist resources for that transaction in one-phase commit, which has the following benefits:

- Simplified eXtreme Scale application development. Previously, developers coordinated eXtreme Scale transactions with resources, such as enterprise beans, servlets, and web containers. Because no rollback mechanism existed, developers had no simple way to recover failures.
- Tighter integration exists with WebSphere Application Server, which includes last participant support to enlist in global transactions if necessary.

Scenario goals

After completing this scenario, you will know how to complete the following goals:

- Use Java Transaction API (JTA) support to develop application components that use transactions.
- Connect your applications with eXtreme Scale clients.

Transaction processing in Java EE applications

WebSphere eXtreme Scale provides its own resource adapter that you can use to connect applications to the data grid and process local transactions.

Through support from the eXtreme Scale resource adapter, Java Platform, Enterprise Edition (Java EE) applications can look up eXtreme Scale client connections and demarcate local transactions using Java EE local transactions or using the eXtreme Scale APIs. When the resource adapter is configured, you can complete the following actions with your Java EE applications:

- Look up or inject eXtreme Scale resource adapter connection factories within a Java EE application component.
- Obtain standard connection handles to the eXtreme Scale client and share them between application components using Java EE conventions.
- Demarcate eXtreme Scale transactions using either the `javax.resource.cci.LocalTransaction` API or the `com.ibm.websphere.objectgrid.Session` interface.
- Use the entire eXtreme Scale client API, such as the `ObjectMap` API and `EntityManager` API.

The following additional capabilities are available with WebSphere Application Server:

- Enlist eXtreme Scale connections with a global transaction as a last participant with other two-phase commit resources. The eXtreme Scale resource adapter provides local transaction support, with a single-phase commit resource. With WebSphere Application Server, your applications can enlist one, single-phase commit resource into a global transaction through last participant support.
- Automatic resource adapter installation when the profile is augmented.
- Automatic security principal propagation.

Administrator responsibilities

The eXtreme Scale resource adapter is installed into the Java EE application server or embedded with the application. After you install the resource adapter, the administrator creates one or more resource adapter connection factories for each catalog service domain and optionally each data grid instance. The connection factory identifies the properties that are required to communicate with the data grid.

Applications reference the connection factory, which establishes the connection to the remote data grid. Each connection factory hosts a single eXtreme Scale client connection that is reused for all application components.

Important: Because the eXtreme Scale client connection might include a near cache, applications must not share a connection. A connection factory must exist for a single application instance to avoid problems sharing objects between applications.

The connection factory hosts an eXtreme Scale client connection, which is shared between all referencing application components. You can use a managed bean (MBean) to access information about the client connection or to reset the connection when it is no longer needed.

Application developer responsibilities

An application developer creates resource references for managed connection factories in the application deployment descriptor or with annotations. Each resource reference includes a local reference for the eXtreme Scale connection factory, as well as the resource-sharing scope.

Important: Enabling resource sharing is important because it allows the local transaction to be shared between application components.

Applications can inject the connection factory into the Java EE application component, or it can look up the connection factory using Java Naming Directory Interface (JNDI). The connection factory is used to obtain connection handles to the eXtreme Scale client connection. The eXtreme Scale client connection is managed independently from the resource adapter connection and is established on first use, and reused for all subsequent connections.

After finding the connection, the application retrieves an eXtreme Scale session reference. With the eXtreme Scale session reference, the application can use the entire eXtreme Scale client APIs and features.

You can demarcate transactions in one of the following ways:

- Use the `com.ibm.websphere.objectgrid.Session` transaction demarcation methods.
- Use the `javax.resource.cci.LocalTransaction` local transaction.
- Use a global transaction, when you use WebSphere Application Server with last participant support enabled. When you select this approach for demarcation, you must:
 - Use an application-managed global transaction with the `javax.transaction.UserTransaction`.
 - Use a container-managed transaction.

Application deployer responsibilities

The application deployer binds the local reference to the resource adapter connection factory that the application developer defines to the resource adapter connection factories that the administrator defines. The application deployer must assign the correct connection factory type and scope to the application and ensure that the connection factory is not shared between applications to avoid Java object sharing. The application deployer is also responsible for configuring and mapping other appropriate configuration information that is common to all connection factories.

Installing an eXtreme Scale resource adapter

The WebSphere eXtreme Scale resource adapter is Java Connector Architecture (JCA) 1.5 compatible and can be installed on a Java 2 Platform, Enterprise Edition (J2EE) 1.4 or later application server such as WebSphere Application Server.

Before you begin

The resource adapter is in the `wxsra.rar` resource adapter archive (RAR) file, which is available in all installations of eXtreme Scale. The RAR file is in the following directories:

- For WebSphere Application Server installations: `wxs_install_root/optionalLibraries/ObjectGrid`
- For stand-alone installations: `wxs_install_root/ObjectGrid/lib` directory

The resource adapter is coupled with the eXtreme Scale runtime environment. It requires the eXtreme Scale runtime JAR files in the correct classpath. In general, you can upgrade the eXtreme Scale runtime environment without updating the resource adapter. Upgrading the eXtreme Scale runtime environment also upgrades the resource adapter runtime environment. The resource adapter supports version 8.5 and up to two versions later of the eXtreme Scale runtime environment. Later versions of the resource adapter might require later versions of the eXtreme Scale runtime environment as they become available.

The `wxsra.rar` file requires one of the eXtreme Scale client runtime JAR files to operate. For details about which client runtime JAR file is appropriate, see *Runtime files for WebSphere eXtreme Scale stand-alone installation* and *Runtime files for WebSphere eXtreme Scale integrated with WebSphere Application Server*, which include details about the available runtime JAR files.

About this task

You can install the eXtreme Scale resource adapter using several options that allow for flexible deployment scenarios. The resource adapter can be embedded with the Java Platform, Enterprise Edition (Java EE) application, or it can be installed as a stand-alone RAR file that is shared between applications.

Embedding the resource adapter with the application simplifies deployment because connection factories are only created within the scope of the application and cannot be shared between applications. With the resource adapter embedded in the application, you can also embed the cache objects and ObjectGrid client plug-in classes within the application. Embedding the resource adapter also protects the application from inadvertently sharing cache objects between applications, which can result in `java.lang.ClassCastException` exceptions.

By installing the `wxsra.rar` file as a stand-alone resource adapter, you can create resource manager connection factories at the node scope. This option is useful in the following situations:

- When it is not practical to embed the `wxsra.rar` file inside the application
- When the version of eXtreme Scale is not known at build time
- When you want to share an eXtreme Scale client connection with multiple applications

Important: In multiple versions of WebSphere Application Server, up to Version 8.0.2, you cannot install the eXtreme Scale resource adapter in an application EAR

file and in the stand-alone server simultaneously. The result, when you use the enterprise archive (EAR) file that also has the RAR file installed, is that the application experiences an exception, such as `ClassCastException`: `com.ibm.websphere.xs.ra.XSConnectionFactory` incompatible with `com.ibm.websphere.xs.ra.XSConnectionFactory`. The following example WebSphere Application Server message and call stack for this error are displayed when a servlet encounters this exception:

```
SRVE0068E: An exception was thrown by one of the service methods of the servlet [ClientServlet]
in application [JTASampleClientEAR]. Exception created : [java.lang.ClassCastException:
com.ibm.websphere.xs.ra.XSConnectionFactory incompatible with com.ibm.websphere.xs.ra.XSConnectionFactory
at com.ibm.websphere.xs.sample.jtasample.WXSClientServlet.connectClient(WXSClientServlet.java:484)
at com.ibm.websphere.xs.sample.jtasample.WXSClientServlet.doGet(WXSClientServlet.java:200)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:575)
at javax.servlet.http.HttpServlet.service(HttpServlet.java:668)
at com.ibm.ws.webcontainer.servlet.ServletWrapper.service(ServletWrapper.java:1214)
at com.ibm.ws.webcontainer.servlet.ServletWrapper.handleRequest(ServletWrapper.java:774)
at com.ibm.ws.webcontainer.servlet.ServletWrapper.handleRequest(ServletWrapper.java:456)
```

Procedure

- **Install an embedded eXtreme Scale resource adapter.** When the `wxsra.rar` file is embedded in the application EAR file, the resource adapter must have access to the eXtreme Scale runtime libraries.

For applications that run in WebSphere Application Server, the following choices and subsequent actions are available:

Option	Description
If eXtreme Scale is integrated with the WebSphere Application Server node	The runtime library files are already available in the system classpath, and no other action is required.
If eXtreme Scale is not integrated with the WebSphere Application Server node	You must include the <code>wsogclient.jar</code> file in the <code>wxsra.rar</code> classpath.

For applications that do not run in WebSphere Application Server, the client runtime library file, `ogclient.jar`, or the server runtime library file, `objectgrid.jar`, must be in the classpath of the RAR file.

- **Install a stand-alone eXtreme Scale resource adapter.** When you install the `wxsra.rar` file as a stand-alone resource adapter, it must have access to the eXtreme Scale runtime libraries.

For applications that run in WebSphere Application Server, the following choices and subsequent actions are available:

Option	Description
If eXtreme Scale is integrated with the WebSphere Application Server node	The runtime library files are already available in the system classpath, and no other action is required.
If eXtreme Scale is not integrated with the WebSphere Application Server node	You must include the <code>wsogclient.jar</code> file in the <code>wxsra.rar</code> classpath.

For applications that do not run in WebSphere Application Server, the client runtime library file, `ogclient.jar`, or the server runtime library file, `objectgrid.jar`, must be in the classpath of the RAR file.

1. Give the resource adapter access to any shared classes. All ObjectGrid plug-in classes and the applications that use them must share a class loader. Since the resource adapter is shared by multiple applications, all classes must be accessible by the same class loader. You can create this access by using a shared library between all applications that interact with the resource adapter.

What to do next

Now that you have installed the eXtreme Scale resource adapter, you can configure connection factories so that your Java EE applications can connect to a remote eXtreme Scale data grid.

Configuring eXtreme Scale connection factories

An eXtreme Scale connection factory allows Java EE applications to connect to a remote WebSphere eXtreme Scale data grid. Use custom properties to configure resource adapters.

Before you begin

Before you create the connection factories, you must install the resource adapter.

About this task

After you install the resource adapter, you can create one or more resource adapter connection factories that represent eXtreme Scale client connections to remote data grids. Complete the following steps to configure a resource adapter connection factory and use it within an application.

You can create an eXtreme Scale connection factory at the node scope for stand-alone resource adapters or within the application for embedded resource adapters. See the related topics for information about how to create connection factories in WebSphere Application Server.

Procedure

1. Using the WebSphere Application Server administrative console to create an eXtreme Scale connection factory that represents an eXtreme Scale client connection. See *Configuring Java EE Connector connection factories* in the administrative console. After you specify properties for the connection factory in the General Properties panel, you must click **Apply** for the Custom properties link to become active.
2. Click **Custom properties** in the administrative console. Set the following custom properties to configure the client connection to the remote data grid.

Table 1. Custom properties for configuring connection factories

Property Name	Type	Description
ConnectionName	String	The name of the eXtreme Scale client connection. The ConnectionName helps identify the connection when exposed as a managed bean. This property is optional. If not specified, the ConnectionName is undefined.
CatalogServiceEndpoints	String	The catalog service domain end points in the format: <host>:<port>[,<host><port>]. For more information, see Catalog service domain settings. This property is required if the catalog service domain is not set.
CatalogServiceDomain	String	The catalog service domain name that is defined in WebSphere Application Server. For more information, see <i>Configuring catalog servers and catalog service domains</i> . This property is required if the CatalogServiceEndpoints property is not set.
ObjectGridName	String	(Optional) The name of the data grid that this connection factory connects to. If not specified, then the application must supply the name when obtaining the connection from the connection factory.
ObjectGridURL	String	(Optional) The URL of the client data grid, override XML file. This property is not valid if the ObjectGridResource is also specified. For more information, see <i>Configuring clients</i> .

Table 1. Custom properties for configuring connection factories (continued)

Property Name	Type	Description
ObjectGridResource	String	The resource path of the client data grid, override XML file. This property is optional and invalid if ObjectGridURL is also specified. For more information, see Configuring clients.
ClientPropertiesURL	String	(Optional) The URL of the client properties file. This property is not valid if the ClientPropertiesResource is also specified. For more information, see Client properties file for more information.
ClientPropertiesResource	String	(Optional) The resource path of the client properties file. This property is not valid if the ClientPropertiesURL is also specified. For more information, see Client properties file for more information.

WebSphere Application Server also allows other configuration options for adjusting connection pools and managing security. See the related information for links to WebSphere Application Server Information Center topics.

What to do next

Create an eXtreme Scale connection factory reference in the application. See “Configuring applications to connect with eXtreme Scale” on page 65 for more information.

Configuring Eclipse environments to use eXtreme Scale connection factories

The eXtreme Scale resource adapter includes custom connection factories. To use these interfaces in your eXtreme Scale Java Platform, Enterprise Edition (Java EE) applications, you must import the `wxsra.rar` file into your workspace and link it to your application project.

Before you begin

- You must install Rational® Application Developer Version 7 or later or Eclipse Java EE IDE for Web Developers Version 1.4 or later.
- A server runtime environment must be configured.

Procedure

1. Import the `wxsra.rar` file into your project by selecting **File > Import**. The Import window is displayed.
2. Select **Java EE > RAR file**. The Connector Import window is displayed.
3. To specify the connector file, click **Browse** to locate the `wxsra.rar` file. The `wxsra.rar` file is installed when you install a resource adapter. You can find the resource adapter archive (RAR) file in the following location:
 - For WebSphere Application Server installations: `wxs_install_root/optionalLibraries/ObjectGrid`
 - For stand-alone installations: `wxs_install_root/ObjectGrid/lib` directory
4. Create a name for the new connector project in the **Connector project** field. You can use `wxsra`, which is the default name.
5. Choose a Target runtime, which references a Java EE server runtime environment.
6. Optionally select **Add project to EAR** to embed the RAR into an existing EAR project.

Results

The RAR file is now imported into your Eclipse workspace.

What to do next

You can reference the RAR project from your other Java EE projects using the following steps:

1. Right click on the project and click **Properties**.
2. Select **Java Build Path**.
3. Select the Projects tab.
4. Click **Add**.
5. Select the **wxsra** connector project, and click **OK**.
6. Click **OK** again to close the Properties window.

The eXtreme Scale resource adapter classes are now in the classpath. To install product runtime JAR files using the Eclipse console, see “Setting up a stand-alone development environment” on page 125 for more information.

Configuring applications to connect with eXtreme Scale

Applications use an eXtreme Scale connection factory to create connection handles to an eXtreme Scale client connection. You can configure resource adapter connection factory references using this task.

Before you begin

Create a Java Platform, Enterprise Edition (Java EE) application component, such as an Enterprise JavaBeans (EJB) container or servlet.

Procedure

Create a `javax.resource.cci.ConnectionFactory` resource reference in the application component. Resource references are declared in the deployment descriptor by the application provider. The connection factory represents an eXtreme Scale client connection that can be used to communicate with one or more named data grids that are available in the catalog service domain.

Securing J2C client connections

Use the Java 2 Connector (J2C) architecture to secure connections between WebSphere eXtreme Scale clients and your applications.

About this task

Applications reference the connection factory, which establishes the connection to the remote data grid. Each connection factory hosts a single eXtreme Scale client connection that is reused for all application components.

Important: Since the eXtreme Scale client connection might include a near cache, it is important that applications do not share a connection. A connection factory must exist for a single application instance to avoid problems sharing objects between applications.

You can set the credential generator with the API or in the client properties file. In the client properties file, the `securityEnabled` and `credentialGenerator` properties are used. The following code example is displayed on multiple lines for publication purposes:

```
securityEnabled=true
credentialGeneratorClass=com.ibm.websphere.objectgrid.security.plugins.builtins.
    UserPasswordCredentialGenerator
credentialGeneratorProps=operator XXXXXX
```

The credential generator and credential in the client properties file are used for the eXtreme Scale connect operation and the default J2C credentials. Therefore, the credentials that are specified with the API are used at J2C connect time for the J2C connection. However, if no credentials are specified at J2C connect time, then the credential generator in the client properties file is used.

Procedure

1. Set up secure access where the J2C connection represents the eXtreme Scale client. Use the ClientPropertiesResource connection factory property or the ClientPropertiesURL connection factory property to configure client authentication.
2. Configure the client security properties to use the connection factory that references the appropriate credential generator object for eXtreme Scale. These properties are also compatible with eXtreme Scale server security. For example, use the WSTokenCredentialGenerator credential generator for WebSphere credentials when eXtreme Scale is installed with WebSphere Application Server. Alternatively, use the UserPasswordCredentialGenerator credential generator when you run the eXtreme Scale in a stand-alone environment. In the following example, credentials are passed programmatically using the API call instead of using the configuration in the client properties:

```
XSConnectionSpec spec = new XSConnectionSpec();
spec.setCredentialGenerator(new UserPasswordCredentialGenerator("operator", "xxxxxx"));
Connection conn = connectionFactory.getConnection(spec);
```

3. (Optional) Disable the near cache, if required.

All J2C connections from a single connection factory share a single near cache. Grid entry permissions and map permissions are validated on the server, but not on the near cache. When an application uses multiple credentials to create J2C connections, and the configuration uses specific permissions for grid entries and maps for those credentials, then disable the near cache. Disable the near cache using the connection factory property, ObjectGridResource or ObjectGridURL. Set the numberOfBuckets property to 0.

4. (Optional) Set security policy settings, if required.

If the J2EE application contains the embedded eXtreme Scale resource adapter archive (RAR) file configuration, you might be required to set additional security policy settings in the security policy file for the application. For example, these policies are required:

```
permission com.ibm.websphere.security.WebSphereRuntimePermission "accessRuntimeClasses";
permission java.lang.RuntimePermission "accessDeclaredMembers";
permission javax.management.MBeanTrustPermission "register";
permission java.lang.RuntimePermission "getClassLoader";
```

Additionally, any property or resource files used by connection factories require file or other permissions, such as permission java.io.FilePermission "filePath";. For WebSphere Application Server, the policy file is META-INF/was.policy, and it is located in the J2EE EAR file.

Results

The client security properties that you configured on the catalog service domain are used as default values. The values that you specify override any properties that are defined in the client.properties files.

What to do next

Use eXtreme Scale data access APIs to develop client components that you want to use transactions.

Developing eXtreme Scale client components to use transactions

The WebSphere eXtreme Scale resource adapter provides client connection management and local transaction support. With this support, Java Platform, Enterprise Edition (Java EE) applications can look up eXtreme Scale client connections and demarcate local transactions with Java EE local transactions or the eXtreme Scale APIs.

Before you begin

Create an eXtreme Scale connection factory resource reference.

About this task

There are several options for working with eXtreme Scale data access APIs. In all cases, the eXtreme Scale connection factory must be injected into the application component, or looked up in Java Naming Directory Interface (JNDI). After the connection factory is looked up, you can demarcate transactions and create connections to access the eXtreme Scale APIs.

You can optionally cast the `javax.resource.cci.ConnectionFactory` instance to a `com.ibm.websphere.xs.ra.XSConnectionFactory` that provides additional options for retrieving connection handles. The resulting connection handles must be cast to the `com.ibm.websphere.xs.ra.XSConnection` interface, which provides the `getSession` method. The `getSession` method returns a `com.ibm.websphere.objectgrid.Session` object handle that allows applications to use any of the eXtreme Scale data access APIs, such as the `ObjectMap` API and `EntityManager` API.

The Session handle and any derived objects are valid for the life of the `XSConnection` handle.

The following procedures can be used to demarcate eXtreme Scale transactions. You cannot mix each of the procedures. For example, you cannot mix global transaction demarcation and local transaction demarcation in the same application component context.

Procedure

- Use autocommit, local transactions. Use the following steps to automatically commit data access operations or operations that do not support an active transaction:
 1. Retrieve a `com.ibm.websphere.xs.ra.XSConnection` connection outside of the context of a global transaction.
 2. Retrieve and use the `com.ibm.websphere.objectgrid.Session` session to interact with the data grid.
 3. Invoke any data access operation that supports autocommit transactions.
 4. Close the connection.
- Use an `ObjectGrid` session to demarcate a local transaction. Use the following steps to demarcate an `ObjectGrid` transaction using the `Session` object:

1. Retrieve a `com.ibm.websphere.xs.ra.XSConnection` connection.
 2. Retrieve the `com.ibm.websphere.objectgrid.Session` session.
 3. Use the `Session.begin()` method to start the transaction.
 4. Use the session to interact with the data grid.
 5. Use the `Session.commit()` or `rollback()` methods to end the transaction.
 6. Close the connection.
- Use a `javax.resource.cci.LocalTransaction` transaction to demarcate a local transaction. Use the following steps to demarcate an `ObjectGrid` transaction using the `javax.resource.cci.LocalTransaction` interface:
 1. Retrieve a `com.ibm.websphere.xs.ra.XSConnection` connection.
 2. Retrieve the `javax.resource.cci.LocalTransaction` transaction using the `XSConnection.getLocalTransaction()` method.
 3. Use the `LocalTransaction.begin()` method to start the transaction.
 4. Retrieve and use the `com.ibm.websphere.objectgrid.Session` session to interact with the data grid.
 5. Use the `LocalTransaction.commit()` or `rollback()` methods to end the transaction.
 6. Close the connection.
 - Enlist the connection in a global transaction. This procedure also applies to container-managed transactions:
 1. Begin the global transaction through the `javax.transaction.UserTransaction` interface or with a container-managed transaction.
 2. Retrieve a `com.ibm.websphere.xs.ra.XSConnection` connection.
 3. Retrieve and use the `com.ibm.websphere.objectgrid.Session` session.
 4. Close the connection.
 5. Commit or roll back the global transaction.

Example

See the following code example, which demonstrates the previous steps for demarcating eXtreme Scale transactions.

```
// (C) Copyright IBM Corp. 2001, 2012.
// All Rights Reserved. Licensed Materials - Property of IBM.
package com.ibm.ws.xs.ra.test.ee;

import javax.naming.InitialContext;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.LocalTransaction;
import javax.transaction.Status;
import javax.transaction.UserTransaction;

import junit.framework.TestCase;

import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.xs.ra.XSConnection;

/**
 * This sample requires that it runs in a J2EE context in your
 * application server. For example, using the JUnitEE framework servlet.
 *
 * The code in these test methods would typically reside in your own servlet,
 * EJB, or other web component.
 *
 * The sample depends on a configured WebSphere eXtreme Scale connection
 * factory registered at of JNDI Name of "eis/embedded/wxscf" that defines
 * a connection to a grid containing a Map with the name "Map1".
 */
```

```

* The sample does a direct lookup of the JNDI name and does not require
* resource injection.
*/
public class DocSampleTests extends TestCase {
    public final static String CF_JNDI_NAME = "eis/embedded/wxscf";
    public final static String MAP_NAME = "Map1";

    Long            key = null;
    Long            value = null;
    InitialContext  ctx = null;
    ConnectionFactory cf = null;

    public DocSampleTests() {
    }
    public DocSampleTests(String name) {
        super(name);
    }
    protected void setUp() throws Exception {
        ctx = new InitialContext();
        cf = (ConnectionFactory)ctx.lookup(CF_JNDI_NAME);
        key = System.nanoTime();
        value = System.nanoTime();
    }
    /**
     * This example runs when not in the context of a global transaction
     * and uses autocommit.
     */
    public void testLocalAutocommit() throws Exception {
        Connection conn = cf.getConnection();
        try {
            Session session = ((XSConnection)conn).getSession();
            ObjectMap map = session.getMap(MAP_NAME);
            map.insert(key, value); // Or various data access operations
        }
        finally {
            conn.close();
        }
    }
    /**
     * This example runs when not in the context of a global transaction
     * and demarcates the transaction using session.begin()/session.commit()
     */
    public void testLocalSessionTransaction() throws Exception {
        Session session = null;
        Connection conn = cf.getConnection();
        try {
            session = ((XSConnection)conn).getSession();
            session.begin();
            ObjectMap map = session.getMap(MAP_NAME);
            map.insert(key, value); // Or various data access operations
            session.commit();
        }
        finally {
            if (session != null && session.isTransactionActive()) {
                try { session.rollback(); }
                catch (Exception e) { e.printStackTrace(); }
            }
            conn.close();
        }
    }
    /**
     * This example uses the LocalTransaction interface to demarcate
     * transactions.
     */
    public void testLocalTranTransaction() throws Exception {
        LocalTransaction tx = null;
        Connection conn = cf.getConnection();
        try {
            tx = conn.getLocalTransaction();
            tx.begin();
            Session session = ((XSConnection)conn).getSession();
            ObjectMap map = session.getMap(MAP_NAME);
            map.insert(key, value); // Or various data access operations
            tx.commit(); tx = null;
        }
        finally {
            if (tx != null) {

```

```

        try { tx.rollback(); }
        catch (Exception e) { e.printStackTrace(); }
    }
    conn.close();
}

/**
 * This example depends on an externally managed transaction,
 * the externally managed transaction might typically be present in
 * an EJB with its transaction attributes set to REQUIRED or REQUIRES_NEW.
 * NOTE: If there is NO global transaction active, this example runs in auto-commit
 * mode because it doesn't verify a transaction exists.
 */
public void testGlobalTransactionContainerManaged() throws Exception {
    Connection conn = cf.getConnection();
    try {
        Session session = ((XSCConnection)conn).getSession();
        ObjectMap map = session.getMap(MAP_NAME);
        map.insert(key, value); // Or various data access operations
    }
    catch (Throwable t) {
        t.printStackTrace();
        UserTransaction tx = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
        if (tx.getStatus() != Status.STATUS_NO_TRANSACTION) {
            tx.setRollbackOnly();
        }
    }
    finally {
        conn.close();
    }
}

/**
 * This example demonstrates starting a new global transaction using the
 * UserTransaction interface. Typically the container starts the global
 * transaction (for example in an EJB with a transaction attribute of
 * REQUIRES_NEW), but this sample will also start the global transaction
 * using the UserTransaction API if it is not currently active.
 */
public void testGlobalTransactionTestManaged() throws Exception {
    boolean started = false;
    UserTransaction tx = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
    if (tx.getStatus() == Status.STATUS_NO_TRANSACTION) {
        tx.begin();
        started = true;
    }
    // else { called with an externally/container managed transaction }
    Connection conn = null;
    try {
        conn = cf.getConnection(); // Get connection after the global tran starts
        Session session = ((XSCConnection)conn).getSession();
        ObjectMap map = session.getMap(MAP_NAME);
        map.insert(key, value); // Or various data access operations
        if (started) {
            tx.commit(); started = false; tx = null;
        }
    }
    finally {
        if (started) {
            try { tx.rollback(); }
            catch (Exception e) { e.printStackTrace(); }
        }
        if (conn != null) { conn.close(); }
    }
}
}

```

Administering J2C client connections

The WebSphere eXtreme Scale connection factory includes an eXtreme Scale client connection that can be shared between applications and persisted through application restarts.

About this task

The client connection includes a management bean that provides connection status information and lifecycle management operations.

Procedure

Maintain client connections. When the first connection is obtained from the XSConnectionFactory connection factory object, an eXtreme Scale client connection is established to the remote data grid and the ObjectGridJ2CConnection MBean is created. The client connection is maintained for the life of the process. To end a client connection, invoke one of the following events::

- Stop the resource adapter. A resource adapter can be stopped, for example, when it is embedded in an application and the application is stopped.
- Invoke the resetConnection MBean operation on the ObjectGridJ2CConnection MBean. When the connection is reset, all connections are invalidated, transactions completed, and the ObjectGrid client connection is destroyed. Subsequent calls to the getConnection methods on the connection factory result in a new client connection.

WebSphere Application Server also provides additional management beans for managing J2C connections, monitoring connection pools, and performance.

Chapter 3. Getting started



After you install the product, you can use the getting started sample to test the installation and use the product for the first time.

Tutorial: Getting started with WebSphere eXtreme Scale

After you install WebSphere eXtreme Scale in a stand-alone environment, you can use the getting started sample application as a simple introduction to its capability as an in-memory data grid. The getting started sample application is only included in full (client and server) installations of WebSphere eXtreme Scale.

Learning objectives

- Learn about the ObjectGrid descriptor XML file and deployment policy descriptor XML files that you use to configure your environment
- Start catalog and container servers using the configuration files
- Learn about developing a client application
- Run the client application to insert data into the data grid
- Monitor your data grids with the Web console

Time required

60 minutes

Getting started tutorial lesson 1: Defining data grids with configuration files

The `objectgrid.xml` and `deployment.xml` files are required to start container servers.

The sample uses the `objectgrid.xml` and `deployment.xml` files that are in the `wxs_install_root/ObjectGrid/gettingstarted/xml` directory. These files are passed to the start commands to start container servers and a catalog server. The `objectgrid.xml` file is the ObjectGrid descriptor XML file. The `deployment.xml` file is the ObjectGrid deployment policy descriptor XML file. These files together define a distributed topology.

ObjectGrid descriptor XML file

An ObjectGrid descriptor XML file is used to define the structure of the ObjectGrid that is used by the application. It includes a list of backing map configurations. These backing maps store the cache data. The following example is a sample `objectgrid.xml` file. The first few lines of the file include the required header for each ObjectGrid XML file. This example file defines the Grid ObjectGrid with Map1 and Map2 backing maps.

```
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="Grid">
```

```

        <backingMap name="Map1" />
        <backingMap name="Map2" />
    </objectGrid>
</objectGrids>

</objectGridConfig>

```

Deployment policy descriptor XML file

The deployment policy descriptor XML file is intended to be paired with the corresponding ObjectGrid XML, the `objectgrid.xml` file. In the following example, the first few lines of the `deployment.xml` file include the required header for each deployment policy XML file. The file defines the `objectgridDeployment` element for the Grid ObjectGrid that is defined in the `objectgrid.xml` file. Both the `Map1` and `Map2` BackingMaps that are defined within the Grid ObjectGrid are included in the `mapSet` `mapSet` that has the `numberOfPartitions`, `minSyncReplicas`, and `maxSyncReplicas` attributes configured.

```

<deploymentPolicy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/deploymentPolicy
  ../deploymentPolicy.xsd"
  xmlns="http://ibm.com/ws/objectgrid/deploymentPolicy">

    <objectgridDeployment objectgridName="Grid">
        <mapSet name="mapSet" numberOfPartitions="13" minSyncReplicas="0"
          maxSyncReplicas="1" >
            <map ref="Map1"/>
            <map ref="Map2"/>
        </mapSet>
    </objectgridDeployment>

</deploymentPolicy>

```

The `numberOfPartitions` attribute of the `mapSet` element specifies the number of partitions for the `mapSet`. It is an optional attribute and the default is 1. The number should be appropriate for the anticipated capacity of the data grid.

The `minSyncReplicas` attribute of `mapSet` is to specify the minimum number of synchronous replicas for each partition in the `mapSet`. It is an optional attribute and the default is 0. Primary and replica are not placed until the domain can support the minimum number of synchronous replicas. To support the `minSyncReplicas` value, you need one more container than the value of `minSyncReplicas`. If the number of synchronous replicas falls below the value of `minSyncReplicas`, write transactions are no longer allowed for that partition.

The `maxSyncReplicas` attribute of `mapSet` is to specify the maximum number of synchronous replicas for each partition in the `mapSet`. It is an optional attribute and the default is 0. No other synchronous replicas are placed for a partition after a domain reaches this number of synchronous replicas for that specific partition. Adding containers that can support this ObjectGrid can result in an increased number of synchronous replicas if your `maxSyncReplicas` value has not already been met. The sample set the `maxSyncReplicas` to 1 means the domain will at most place one synchronous replica. If you start more than one container server instance, there will be only one synchronous replica placed in one of the container server instances.

Lesson checkpoint

In this lesson, you learned:

- How to define maps that store the data in the ObjectGrid descriptor XML file.

- How to use the deployment descriptor XML file to define the number of partitions and replicas for the data grid.

Getting started tutorial lesson 2: Creating a client application

To insert, delete, update, and retrieve data from your data grid, you must write a client application. The getting started sample includes a client application that you can use to learn about creating your own client application.

The `Client.java` file in the `wxs_install_root/ObjectGrid/gettingstarted/client/src/` directory is the client program that demonstrates how to connect to a catalog server, obtain the `ObjectGrid` instance, and use the `ObjectMap` API. The `ObjectMap` API stores data as key-value pairs and is ideal for caching objects that have no relationships involved. The following steps discuss the contents of the `Client.java` file.

If you need to cache objects that have relationships, use the `EntityManager` API.

1. Connect to the catalog service by obtaining a `ClientClusterContext` instance.

To connect to the catalog server, use the `connect` method of `ObjectGridManager` API. The `connect` method that is used requires only the catalog server endpoint in the format of `hostname:port`. You can indicate multiple catalog server endpoints by separating the list of `hostname:port` values with commas. The following code snippet demonstrates how to connect to a catalog server and obtain a `ClientClusterContext` instance:

```
ClientClusterContext ccc = ObjectGridManagerFactory.getObjectGridManager().connect("localhost:2809", null, null);
```

If the connections to the catalog servers succeed, the `connect` method returns a `ClientClusterContext` instance. The `ClientClusterContext` instance is required to obtain the `ObjectGrid` from `ObjectGridManager` API.

2. Obtain an `ObjectGrid` instance.

To obtain `ObjectGrid` instance, use the `getObjectGrid` method of the `ObjectGridManager` API. The `getObjectGrid` method requires both the `ClientClusterContext` instance and the name of the data grid instance. The `ClientClusterContext` instance is obtained during the connection to catalog server. The name of `ObjectGrid` instance is `Grid` that is specified in the `objectgrid.xml` file. The following code snippet demonstrates how to obtain the data grid by calling the `getObjectGrid` method of the `ObjectGridManager` API.

```
ObjectGrid grid = ObjectGridManagerFactory.getObjectGridManager().getObjectGrid(ccc, "Grid");
```

3. Get a `Session` instance.

You can get a `Session` from the obtained `ObjectGrid` instance. A `Session` instance is required to get the `ObjectMap` instance, and perform transaction demarcation. The following code snippet demonstrates how to get a `Session` instance by calling the `getSession` method of the `ObjectGrid` API.

```
Session sess = grid.getSession();
```

4. Get an `ObjectMap` instance.

After getting a `Session`, you can get an `ObjectMap` instance from a `Session` instance by calling `getMap` method of the `Session` API. You must pass the name of map as parameter to `getMap` method to get the `ObjectMap` instance. The following code snippet demonstrates how to obtain `ObjectMap` by calling the `getMap` method of the `Session` API.

```
ObjectMap map1 = sess.getMap("Map1");
```

5. Use the `ObjectMap` methods.

After an ObjectMap instance is obtained, you can use the ObjectMap API. Remember that the ObjectMap interface is a transactional map and requires transaction demarcation by using the begin and commit methods of the Session API. If there is no explicit transaction demarcation in the application, the ObjectMap operations run with auto-commit transactions.

The following code snippet demonstrates how to use the ObjectMap API with an auto-commit transaction.

```
map1.insert(key1, value1);
```

The following code snippet demonstrates how to use the ObjectMap API with explicit transaction demarcation.

```
sess.begin();
map1.insert(key1, value1);
sess.commit();
```

6. Optional: Close the Session. After all of the Session and ObjectMap operations are complete, close the session with the Session.close() method. Running this method returns the resources that were being used by the session.

```
sess.close();
```

As a result, subsequent getSession() method calls return faster, and fewer Session objects are in the heap.

Lesson checkpoint

In this lesson, you learned how to create a simple client application for performing data grid operations.

Getting started tutorial lesson 3: Running the getting started sample client application

Use the following steps to start your first data grid and run a client to interact with the data grid. The catalog server, container server, and client all run on a single server in this example.

The env.sh|bat script is called by the other scripts to set needed environment variables. Normally you do not need to change this script.

- **UNIX** **Linux** ./env.sh
- **Windows** env.bat

To run the application, you must first start the catalog service process. The catalog service is the control center of the data grid. The catalog service tracks the locations of container servers, and controls the placement of data to host container servers. After the catalog service starts, you can start the container servers, which store the application data for the data grid. To store multiple copies of the data, you can start multiple container servers. When all the servers are started, you can run the client application to insert, update, remove, and get data from the data grid.

1. Open a terminal session or command-line window.
2. Use the following command to navigate to the gettingstarted directory:

```
cd wxs_install_root/ObjectGrid/gettingstarted
```
3. Run the following script to start a catalog service process on localhost:

- **UNIX** **Linux** ./runcat.sh
- **Windows** runcat.bat

The catalog service process runs in the current terminal window.

You can also start the catalog service with the **startOgServer** command. Run the **startOgServer** from the `wxs_install_root/ObjectGrid/bin` directory:

- **UNIX** **Linux** `./startOgServer.sh cs0 -catalogServiceEndPoints cs0:localhost:6600:6601 -listenerPort 2809`
- **Windows** `startOgServer.bat cs0 -catalogServiceEndPoints cs0:localhost:6600:6601 -listenerPort 2809`

4. Open another terminal session or command-line window, and run the following command to start a container server instance:

- **UNIX** **Linux** `./runcontainer.sh server0`
- **Windows** `runcontainer.bat server0`

The container server runs in the current terminal window. If you want to start more container server instances to support replication, you can repeat this step with a different server name.

You can also start container servers with the **startOgServer** command. Run the **startOgServer** from the `wxs_install_root/ObjectGrid/bin` directory:

- **UNIX** **Linux** `./startOgServer.sh c0 -catalogServiceEndPoints localhost:2809 -objectgridFile gettingstarted\xml\objectgrid.xml -deploymentPolicyFile gettingstarted/xml/deployment.xml`
- **Windows** `startOgServer.bat c0 -catalogServiceEndPoints localhost:2809 -objectgridFile gettingstarted\xml\objectgrid.xml -deploymentPolicyFile gettingstarted\xml\deployment.xml`

5. Open another terminal session or command-line window to run client commands.

The `runclient.sh|bat` script runs the simple create, retrieve, update, and delete (CRUD) client and starts the specified operation. The `runclient.sh|bat` script is run with the following parameters:

- **UNIX** **Linux** `./runclient.sh command value1 value2`
- **Windows** `runclient.bat command value1 value2`

For *command*, use one of the following options:

- Specify as `i` to insert *value2* into data grid with key *value1*
 - Specify as `u` to update object keyed by *value1* to *value2*
 - Specify as `d` to delete object keyed by *value1*
 - Specify as `g` to retrieve and display object keyed by *value1*
- a. Add data to the data grid.

Important: If your system is using double byte character sets (DBCS), you might see garbled or corrupted text when you insert data into the data grid with the **runClient** script. This text can display in the output or in the cache. To work around this issue, update the Java call in the **runClient** script to include the **-Xargencoding** argument, and then specify the DBCS as a Unicode character set. For example, use the command: `\u runClient i key\u2e81 Hello\u2e84World`

- **UNIX** **Linux** `./runclient.sh i key1 helloWorld`
- **Windows** `runclient.bat i key1 helloWorld`

- b. Search and display the value:

- **UNIX** **Linux** `./runclient.sh g key1`

- `Windows` `runclient.bat g key1`
- c. Update the value:
 - `UNIX` `Linux` `./runclient.sh u key1 goodbyeWorld`
 - `Windows` `runclient.bat u key1 goodbyeWorld`
- d. Delete the value:
 - `UNIX` `Linux` `./runclient.sh d key1`
 - `Windows` `runclient.bat d key1`

Lesson checkpoint

In this lesson, you learned:

- How to start catalog servers and container servers
- How to run the sample client application

Getting started tutorial lesson 4: Monitor your environment

You can use the `xscmd` utility and web console tools to monitor your data grid environment.

Monitoring with the web console


With the web console, you can chart current and historical statistics. This console provides some preconfigured charts for high-level overviews, and has a custom reports page that you can use to build charts from the available statistics. You can use the charting capabilities in the monitoring console of WebSphere eXtreme Scale to view the overall performance of the data grids in your environment.

Install the web console as an optional feature when you run the installation wizard.


1. Start the console server. The `startConsoleServer.bat|sh` script for starting the console server is in the `wxs_install_root/ObjectGrid/bin` directory of your installation.
2. Log on to the console.
 - a. From your web browser, go to `https://your.console.host:7443`, replacing `your.console.host` with the host name of the server onto which you installed the console.
 - b. Log on to the console.
 - **User ID:** admin
 - **Password:** admin

The console welcome page is displayed.
3. Edit the console configuration. Click **Settings > Configuration** to review the console configuration. The console configuration includes information such as:
 - Trace string for the WebSphere eXtreme Scale client, such as `*=all=disabled`
 - The Administrator name and password
 - The Administrator email address
4. Establish and maintain connections to catalog servers that you want to monitor. Repeat the following steps to add each catalog server to the configuration.
 - a. Click **Settings > eXtreme Scale Catalog Servers**.
 - b. Add a new catalog server.



- 1) Click the add icon () to register an existing catalog server.
 - 2) Provide information, such as the host name and listener port. See Planning for network ports for more information about port configuration and defaults.
 - 3) Click **OK**.
 - 4) Verify that the catalog server has been added to the navigation tree.
5. Group the catalog servers that you created into a catalog service domain. You must create a catalog service domain when security is enabled in your catalog servers because security settings are configured in the catalog service domain.
- a. Click **Settings > eXtreme Scale Domains** page.
 - b. Add a new catalog service domain.



- 1) Click the add icon () to register a catalog service domain. Enter a name for the catalog service domain.
- 2) After you create the catalog service domain, you can edit the properties. The catalog service domain properties follow:

Name Indicates the host name of the domain, as assigned by the administrator.

Catalog servers

Lists one or more catalog servers that belong to the selected domain. You can add the catalog servers that you created in the previous step.

Generator class

Specifies the name of the class that implements the CredentialGenerator interface. This class is used to get credentials for clients. If you specify a value in this field, the value overrides the **credentialGeneratorClass** property in the `client.properties` file.

Generator properties

Specifies the properties for the CredentialGenerator implementation class. The properties are set to the object with the `setProperties(String)` method. The `credentialGeneratorProps` value is used only if the value of the `credentialGeneratorClass` property is not null. If you specify a value in this field, the value overrides the **credentialGeneratorProps** property in the `client.properties` file.

eXtreme Scale client properties path

Specifies the path to the client properties file that you edited to include security properties in a previous step. For example, you might indicate the `c:\ObjectGridProperties\sampleclient.properties` file. If you want to stop the console from trying to use secure connections, you can delete the value in this field. After you set the path, the console uses an unsecured connection.

- 3) Click **OK**.
- 4) Verify that the domain has been added to the navigation tree.

To view information about an existing catalog service domain, click the name of the catalog service domain in the navigation tree on the **Settings > eXtreme Scale Domains** page.

6. View the connection status. The **Current domain** field indicates the name of the catalog service domain that is currently being used to display information in the web console. The connection status displays next to the name of the catalog service domain.
7. View statistics for the data grids and servers, or create a custom report.

Monitoring with the xscmd utility

1. Optional: If client authentication is enabled: Open a command-line window. On the command line, set appropriate environment variables.
 - a. Set the `CLIENT_AUTH_LIB` environment variable:
 - **Windows** `set CLIENT_AUTH_LIB=<path_to_security_JAR_or_classes>`
 - **UNIX** `set CLIENT_AUTH_LIB=<path_to_security_JAR_or_classes>`
`export CLIENT_AUTH_LIB`
2. Go to the `wxs_home/bin` directory.
`cd wxs_home/bin`
3. Run various commands to display information about your environment.
 - Show all the online container servers for the Grid data grid and the mapSet map set:
`xscmd -c showPlacement -g Grid -ms mapSet`
 - Display the routing information for the data grid.
`xscmd -c routetable -g Grid`
 - Display the number of map entries in the data grid.
`xscmd -c showMapSizes -g Grid -ms mapSet`

Stopping the servers

After you are done using the client application and monitoring the getting started sample environment, you can stop the servers.

- If you used the script files to start the servers, use `<ctrl+c>` to stop the catalog service process and container servers in the respective windows.
- If you used the **startOgServer** command to start your servers, use the **stopOgServer** command to stop the servers.

Stop the container server:

- **UNIX** **Linux** `stopOgServer.sh c0 -catalogServiceEndpoints localhost:2809`
- **Windows** `stopOgServer.bat c0 -catalogServiceEndpoints localhost:2809`

Stop the catalog server:

- **UNIX** **Linux** `stopOgServer.sh cs1 -catalogServiceEndpoints localhost:2809`
- **Windows** `stopOgServer.bat cs1 -catalogServiceEndpoints localhost:2809`

Lesson checkpoint

In this lesson, you learned:

- How to start the web console and connect it to the catalog server

- How to monitor data grid and server statistics
- How to stop the servers

Getting started with developing applications

To begin developing WebSphere eXtreme Scale applications, set up a development environment in Eclipse.

About this task

When you are developing WebSphere eXtreme Scale applications, you can use the embedded server APIs to create and start servers, ObjectGrid instances, and to insert data into the data grid. You can unit test your application and the associated configuration directly in the Eclipse environment.

When you are ready to move your application to a broader environment, you can create configuration XML files that you import to create your deployment.

Procedure

1. Set up a development environment in Eclipse.

By adding the WebSphere eXtreme Scale Java archive (JAR) files to the development environment you can begin to use the APIs to develop your applications.

More information: “Setting up a stand-alone development environment” on page 125

2. Create a simple application that starts servers, creates an ObjectGrid instance, and inserts data into the data grid.

- a. Use the ServerFactory API to start and stop servers.

More information: Using the embedded server API to start and stop servers

- b. Use the ObjectGridManager API to retrieve the ObjectGrid instance that you created.

More information: “Interacting with an ObjectGrid using the ObjectGridManager interface” on page 135

- c. Use the ObjectMap API to insert data into the data grid.

More information: “Caching objects with no relationships involved (ObjectMap API)” on page 152 The ObjectMap API is the simplest way to access and write data to the data grid. If your objects have complex relationships, you can use the following APIs to read, write, and update data:

- “Accessing data with indexes (Index API)” on page 142
- “Caching objects and their relationships (EntityManager API)” on page 164
- “Retrieving entities and objects (Query API)” on page 190
- “Accessing data with the REST data service” on page 255

For more information about choosing between the different APIs, see Chapter 5, “Developing applications,” on page 125.

3. Unit test your application.

You can also use the **xscmd** utility to display information about the running servers, replicas, and so on. See Administering with the **xscmd** utility for more information.

4. When you are satisfied with your application within the development environment, create XML configuration files and update your application to use the configuration. The Getting Started sample application provides examples of these configuration files and a simple Java application that uses the configuration files.

More information:“Tutorial: Getting started with WebSphere eXtreme Scale” on page 73

5. Run your application using the XML configuration files. How you start your servers depends on the environment that you are using.

You can run your application in one of the following containers:

- Stand-alone Java virtual machine (JVM)
- Tomcat
- WebSphere Application Server
- OSGi

Chapter 4. Planning



Before you install WebSphere eXtreme Scale and deploy your data grid applications, you must decide on your caching topology, complete capacity planning, review the hardware and software requirements, networking and tuning settings, and so on. You can also use the operational checklist to ensure that your environment is ready to have an application deployed.

For a discussion of the best practices that you can use when you are designing your WebSphere eXtreme Scale applications, read the following article on developerWorks®: Principles and best practices for building high performing and highly resilient WebSphere eXtreme Scale applications.

Planning the topology

With WebSphere eXtreme Scale, your architecture can use local in-memory data caching or distributed client-server data caching. The architecture can have varied relationships with your databases. You can also configure the topology to span multiple data centers.

WebSphere eXtreme Scale requires minimal additional infrastructure to operate. The infrastructure consists of scripts to install, start, and stop a Java Platform, Enterprise Edition application on a server. Cached data is stored in the container servers, and clients remotely connect to the server.

In-memory environments

When you deploy in a local, in-memory environment, WebSphere eXtreme Scale runs within a single Java virtual machine and is not replicated. To configure a local environment you can use an ObjectGrid XML file or the ObjectGrid APIs.

Distributed environments

When you deploy in a distributed environment, WebSphere eXtreme Scale runs across a set of Java virtual machines, increasing the performance, availability and scalability. With this configuration, you can use data replication and partitioning. You can also add additional servers without restarting your existing eXtreme Scale servers. As with a local environment, an ObjectGrid XML file, or an equivalent programmatic configuration, is needed in a distributed environment. You must also provide a deployment policy XML file with configuration details

You can create either simple deployments or large, terabyte-sized deployments in which thousands of servers are needed.

Local in-memory cache

In the simplest case, WebSphere eXtreme Scale can be used as a local (non-distributed) in-memory data grid cache. The local case can especially benefit high-concurrency applications where multiple threads need to access and modify transient data. The data kept in a local data grid can be indexed and retrieved using queries. Queries help you to work with large in memory data sets. The support provided with the Java virtual machine (JVM), although it is ready to use, has a limited data structure.

The local in-memory cache topology for WebSphere eXtreme Scale is used to provide consistent, transactional access to temporary data within a single Java virtual machine.

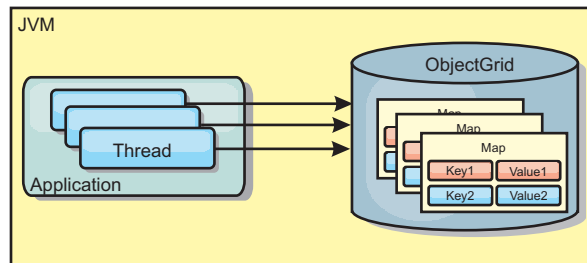


Figure 5. Local in-memory cache scenario

Advantages

- Simple setup: An ObjectGrid can be created programmatically or declaratively with the ObjectGrid deployment descriptor XML file or with other frameworks such as Spring.
- Fast: Each BackingMap can be independently tuned for optimal memory utilization and concurrency.
- Ideal for single-Java virtual machine topologies with small dataset or for caching frequently accessed data.
- Transactional. BackingMap updates can be grouped into a single unit of work and can be integrated as a last participant in 2-phase transactions such as Java Transaction Architecture (JTA) transactions.

Disadvantages

- Not fault tolerant.
- The data is not replicated. In-memory caches are best for read-only reference data.
- Not scalable. The amount of memory required by the database might overwhelm the Java virtual machine.
- Problems occur when adding Java virtual machines:
 - Data cannot easily be partitioned
 - Must manually replicate state between Java virtual machines or each cache instance could have different versions of the same data.
 - Invalidation is expensive.
 - Each cache must be warmed up independently. The warm-up is the period of loading a set of data so that the cache gets populated with valid data.

When to use

The local, in-memory cache deployment topology should only be used when the amount of data to be cached is small (can fit into a single Java virtual machine) and is relatively stable. Stale data must be tolerated with this approach. Using evictors to keep most frequently or recently used data in the cache can help keep the cache size low and increase relevance of the data.

Peer-replicated local cache

You must ensure the cache is synchronized if multiple processes with independent cache instances exist. To ensure that the cache instances are synchronized, enable a peer-replicated cache with Java Message Service (JMS).

WebSphere eXtreme Scale includes two plug-ins that automatically propagate transaction changes between peer ObjectGrid instances. The JMSObjectGridEventListener plug-in automatically propagates eXtreme Scale changes using JMS.

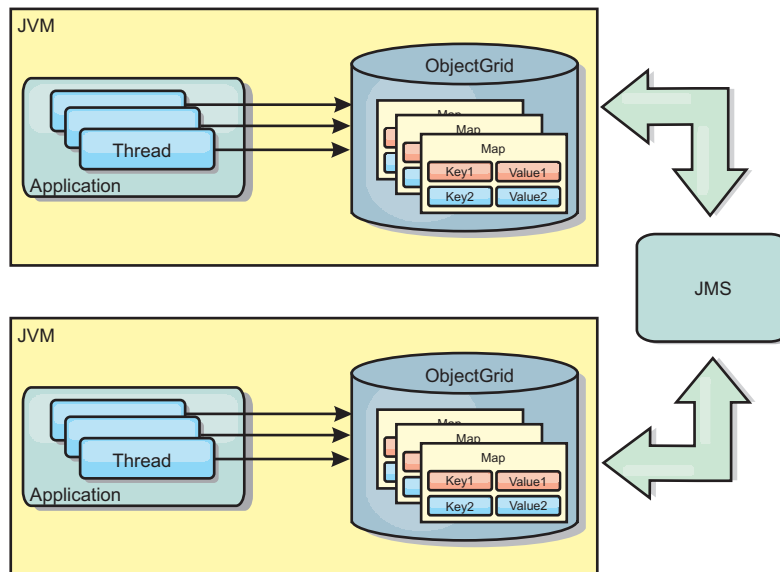


Figure 6. Peer-replicated cache with changes that are propagated with JMS

If you are running a WebSphere Application Server environment, the TranPropListener plug-in is also available. The TranPropListener plug-in uses the high availability (HA) manager to propagate the changes to each peer cache instance.

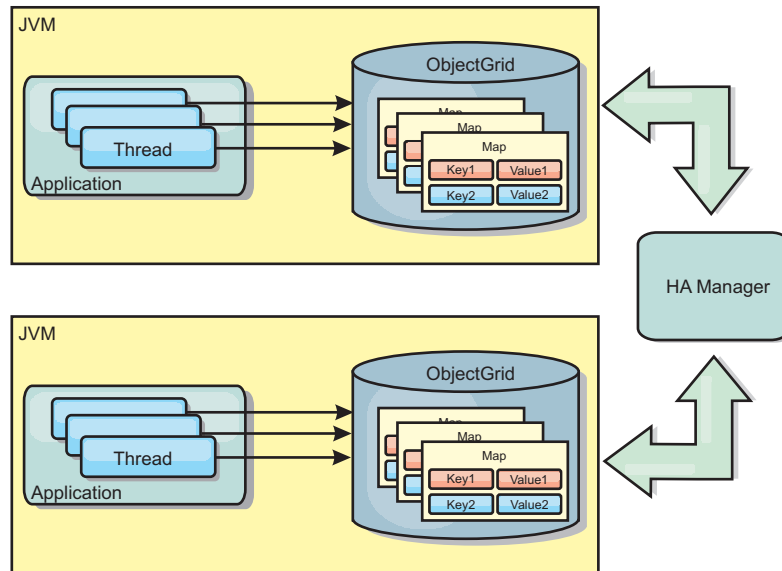


Figure 7. Peer-replicated cache with changes that are propagated with the high availability manager

Advantages

- The data is more valid because the data is updated more often.
- With the TranPropListener plug-in, like the local environment, the eXtreme Scale can be created programmatically or declaratively with the eXtreme Scale deployment descriptor XML file or with other frameworks such as Spring. Integration with the high availability manager is done automatically.
- Each BackingMap can be independently tuned for optimal memory utilization and concurrency.
- BackingMap updates can be grouped into a single unit of work and can be integrated as a last participant in 2-phase transactions such as Java Transaction Architecture (JTA) transactions.
- Ideal for few-JVM topologies with a reasonably small dataset or for caching frequently accessed data.
- Changes to the eXtreme Scale are replicated to all peer eXtreme Scale instances. The changes are consistent as long as a durable subscription is used.

Disadvantages

- Configuration and maintenance for the JMSObjectGridEventListener can be complex. eXtreme Scale can be created programmatically or declaratively with the eXtreme Scale deployment descriptor XML file or with other frameworks such as Spring.
- Not scalable: The amount of memory required by the database may overwhelm the JVM.
- Functions improperly when adding Java virtual machines:
 - Data cannot easily be partitioned
 - Invalidation is expensive.
 - Each cache must be warmed-up independently

When to use

Use deployment topology only when the amount of data to be cached is small, can fit into a single JVM, and is relatively stable.

Embedded cache

WebSphere eXtreme Scale grids can run within existing processes as embedded eXtreme Scale servers or you can manage them as external processes.

Embedded grids are useful when you are running in an application server, such as WebSphere Application Server. You can start eXtreme Scale servers that are not embedded by using command line scripts and run in a Java process.

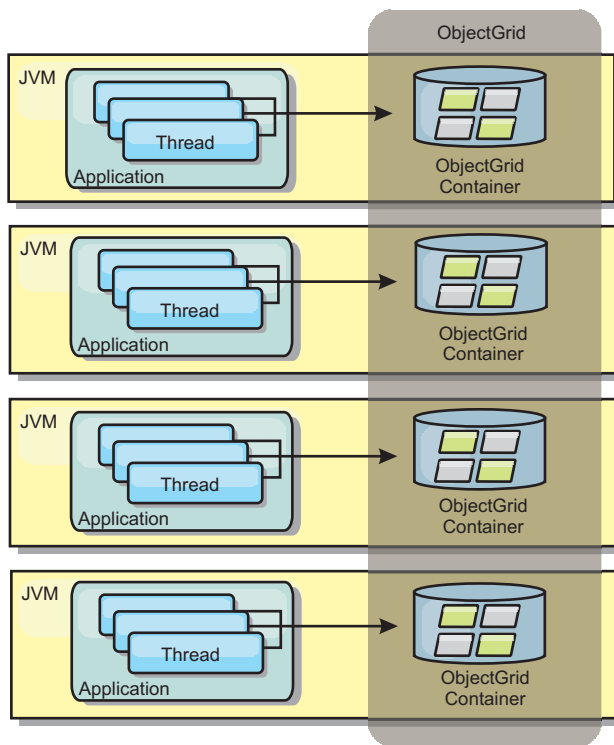


Figure 8. Embedded cache

Advantages

- Simplified administration since there are less processes to manage.
- Simplified application deployment since the grid is using the client application classloader.
- Supports partitioning and high availability.

Disadvantages

- Increased the memory footprint in client process since all of the data is collocated in the process.
- Increase CPU utilization for servicing client requests.
- More difficult to handle application upgrades since clients are using the same application Java archive files as the servers.
- Less flexible. Scaling of clients and grid servers cannot increase at the same rate. When servers are externally defined, you can have more flexibility in managing the number of processes.

When to use

Use embedded grids when there is plenty of memory free in the client process for grid data and potential failover data.

For more information, see the topic on enabling the client invalidation mechanism in the *Administration Guide*.

Distributed cache

WebSphere eXtreme Scale is most often used as a shared cache, to provide transactional access to data to multiple components where a traditional database would otherwise be used. The shared cache eliminates the need to configure a database.

Coherency of the cache

The cache is coherent because all of the clients see the same data in the cache. Each piece of data is stored on exactly one server in the cache, preventing wasteful copies of records that could potentially contain different versions of the data. A coherent cache can also hold more data as more servers are added to the data grid, and scales linearly as the grid grows in size. Because clients access data from this data grid with remote procedural calls, it can also be known as a remote cache, or far cache. Through data partitioning, each process holds a unique subset of the total data set. Larger data grids can both hold more data and service more requests for that data. Coherency also eliminates the need to push invalidation data around the data grid because no stale data exists. The coherent cache only holds the latest copy of each piece of data.

If you are running a WebSphere Application Server environment, the TranPropListener plug-in is also available. The TranPropListener plug-in uses the high availability component (HA Manager) of WebSphere Application Server to propagate the changes to each peer ObjectGrid cache instance.

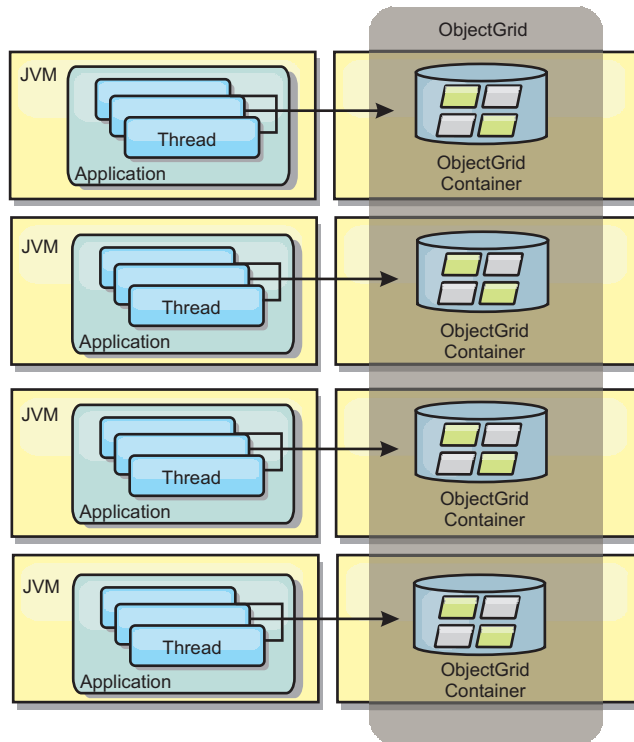


Figure 9. Distributed cache

Near cache

Clients can optionally have a local, in-line cache when eXtreme Scale is used in a distributed topology. This optional cache is called a near cache, an independent ObjectGrid on each client, serving as a cache for the remote, server-side cache. The near cache is enabled by default when locking is configured as optimistic or none and cannot be used when configured as pessimistic.

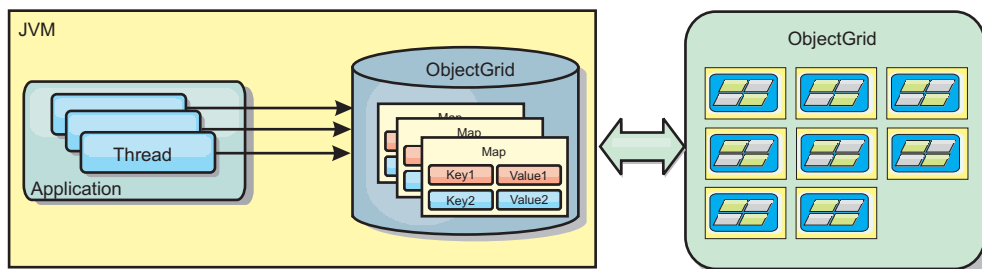


Figure 10. Near cache

A near cache is very fast because it provides in-memory access to a subset of the entire cached data set that is stored remotely in the eXtreme Scale servers. The near cache is not partitioned and contains data from any of the remote eXtreme Scale partitions. WebSphere eXtreme Scale can have up to three cache tiers as follows.

1. The transaction tier cache contains all changes for a single transaction. The transaction cache contains a working copy of the data until the transaction is committed. When a client transaction requests data from an ObjectMap, the transaction is checked first

2. The near cache in the client tier contains a subset of the data from the server tier. When the transaction tier does not have the data, the data is fetched from the client tier, if available and inserted into the transaction cache
3. The data grid in the server tier contains the majority of the data and is shared among all clients. The server tier can be partitioned, which allows a large amount of data to be cached. When the client near cache does not have the data, it is fetched from the server tier and inserted into the client cache. The server tier can also have a Loader plug-in. When the grid does not have the requested data, the Loader is invoked and the resulting data is inserted from the backend data store into the grid.

To disable the near cache, set the `numberOfBuckets` attribute to 0 in the client override eXtreme Scale descriptor configuration. See the topic on map entry locking for details on eXtreme Scale lock strategies. The near cache can also be configured to have a separate eviction policy and different plug-ins using a client override eXtreme Scale descriptor configuration.

Advantage

- Fast response time because all access to the data is local. Looking for the data in the near cache first saves a trip to the grid of servers, thus making even the remote data locally accessible.

Disadvantages

- Increases duration of stale data because the near cache at each tier may be out of synch with the current data in the data grid.
- Relies on an evictor to invalidate data to avoid running out of memory.

When to use

Use when response time is important and stale data can be tolerated.

Database integration: Write-behind, in-line, and side caching

WebSphere eXtreme Scale is used to front a traditional database and eliminate read activity that is normally pushed to the database. A coherent cache can be used with an application directly or indirectly using an object relational mapper. The coherent cache can then offload the database or backend from reads. In a slightly more complex scenario, such as transactional access to a data set where only some of the data requires traditional persistence guarantees, filtering can be used to offload even write transactions.

You can configure WebSphere eXtreme Scale to function as a highly flexible in-memory database processing space. However, WebSphere eXtreme Scale is not an object relational mapper (ORM). It does not know where the data in the data grid came from. An application or an ORM can place data in an eXtreme Scale server. It is the responsibility of the source of the data to make sure that it stays consistent with the database where data originated. This means eXtreme Scale cannot invalidate data that is pulled from a database automatically. The application or mapper must provide this function and manage the data stored in eXtreme Scale.

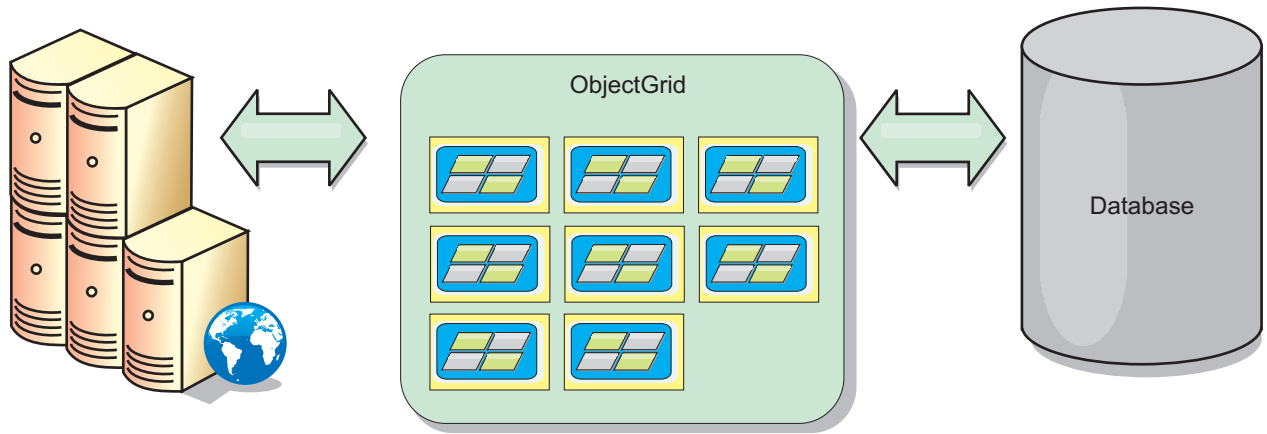


Figure 11. ObjectGrid as a database buffer

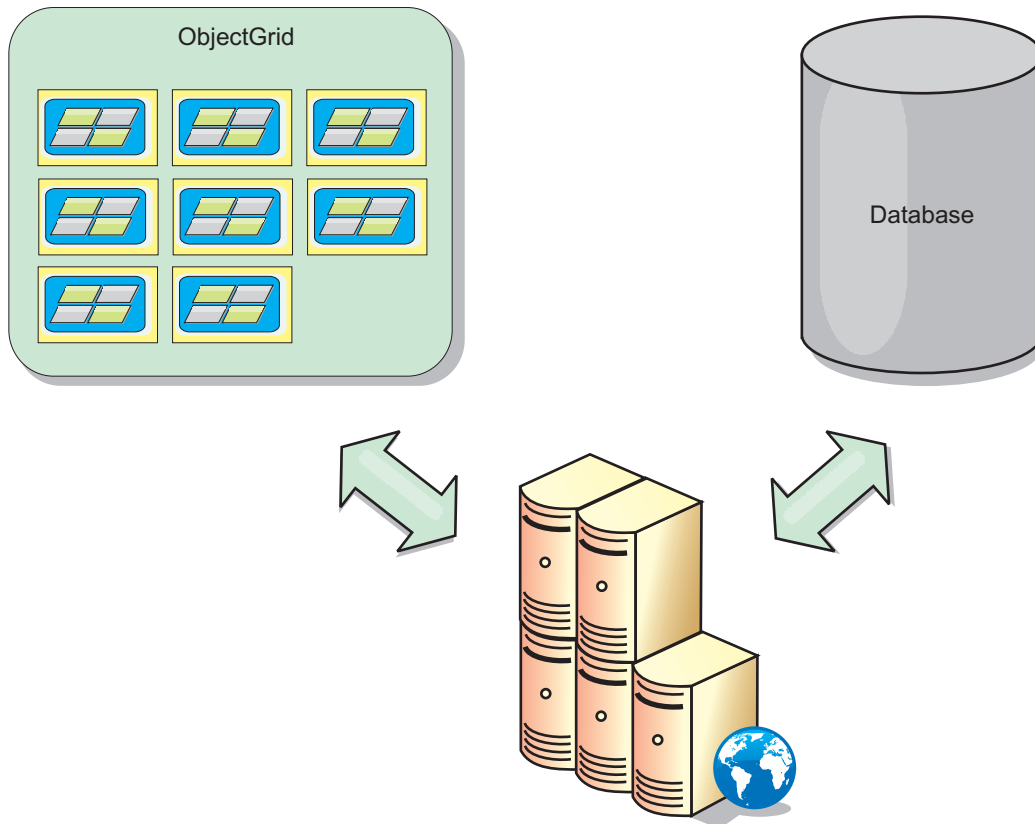


Figure 12. ObjectGrid as a side cache

Sparse and complete cache

WebSphere eXtreme Scale can be used as a sparse cache or a complete cache. A sparse cache only keeps a subset of the total data, while a complete cache keeps all of the data, and can be populated lazily, as the data is needed. Sparse caches are normally accessed using keys (instead of indexes or queries) because the data is only partially available.

Sparse cache

When a key is not present in a sparse cache, or the data is not available and a cache miss occurs, the next tier is invoked. The data is fetched, from a database for example, and is inserted into the data grid cache tier. If you are using a query or index, only the currently loaded values are accessed and the requests are not forwarded to the other tiers.

Complete cache

A complete cache contains all of the required data and can be accessed using non-key attributes with indexes or queries. A complete cache is preloaded with data from the database before the application tries to access the data. A complete cache can function as a database replacement after data is loaded. Because all of the data is available, queries and indexes can be used to find and aggregate data.

Side cache

When WebSphere eXtreme Scale is used as a side cache, the back end is used with the data grid.

Side cache

You can configure the product as a side cache for the data access layer of an application. In this scenario, WebSphere eXtreme Scale is used to temporarily store objects that would normally be retrieved from a back-end database. Applications check to see if the data grid contains the data. If the data is in the data grid, the data is returned to the caller. If the data does not exist, the data is retrieved from the back-end database. The data is then inserted into the data grid so that the next request can use the cached copy. The following diagram illustrates how WebSphere eXtreme Scale can be used as a side-cache with an arbitrary data access layer such as OpenJPA or Hibernate.

Side cache plug-ins for Hibernate and OpenJPA

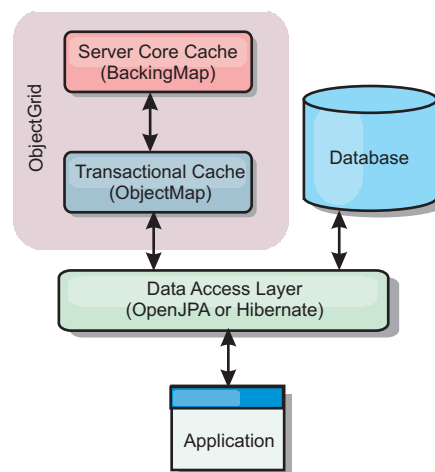


Figure 13. Side cache

Cache plug-ins for both OpenJPA and Hibernate are included in WebSphere eXtreme Scale, so you can use the product as an automatic side-cache. Using WebSphere eXtreme Scale as a cache provider increases performance when reading and querying data and reduces load to the database. There are advantages

that WebSphere eXtreme Scale has over built-in cache implementations because the cache is automatically replicated between all processes. When one client caches a value, all other clients can use the cached value.

In-line cache

You can configure in-line caching for a database back end or as a side cache for a database. In-line caching uses eXtreme Scale as the primary means for interacting with the data. When eXtreme Scale is used as an in-line cache, the application interacts with the back end using a Loader plug-in.

In-line cache

When used as an in-line cache, WebSphere eXtreme Scale interacts with the back end using a Loader plug-in. This scenario can simplify data access because applications can access the eXtreme Scale APIs directly. Several different caching scenarios are supported in eXtreme Scale to make sure the data in the cache and the data in the back end are synchronized. The following diagram illustrates how an in-line cache interacts with the application and back end.

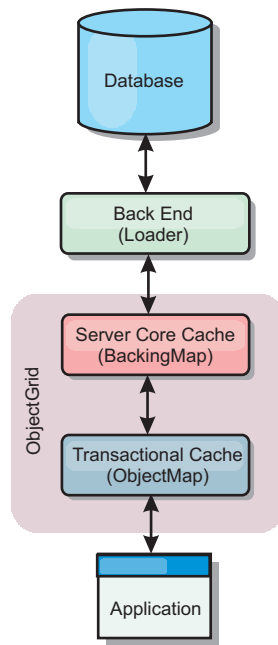


Figure 14. In-line cache

The in-line caching option simplifies data access because it allows applications to access the eXtreme Scale APIs directly. WebSphere eXtreme Scale supports several in-line caching scenarios, as follows.

- Read-through
- Write-through
- Write-behind

Read-through caching scenario

A read-through cache is a sparse cache that lazily loads data entries by key as they are requested. This is done without requiring the caller to know how the entries are populated. If the data cannot be found in the eXtreme Scale cache, eXtreme Scale will retrieve the missing data from the Loader plug-in, which loads the data

from the back-end database and inserts the data into the cache. Subsequent requests for the same data key will be found in the cache until it is removed, invalidated or evicted.

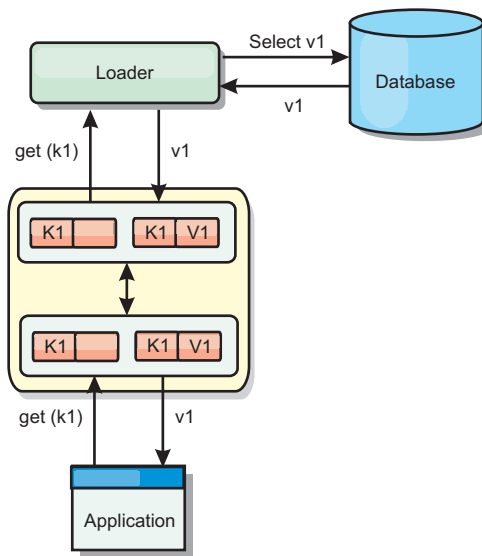


Figure 15. Read-through caching

Write-through caching scenario

In a write-through cache, every write to the cache synchronously writes to the database using the Loader. This method provides consistency with the back end, but decreases write performance since the database operation is synchronous. Since the cache and database are both updated, subsequent reads for the same data will be found in the cache, avoiding the database call. A write-through cache is often used in conjunction with a read-through cache.

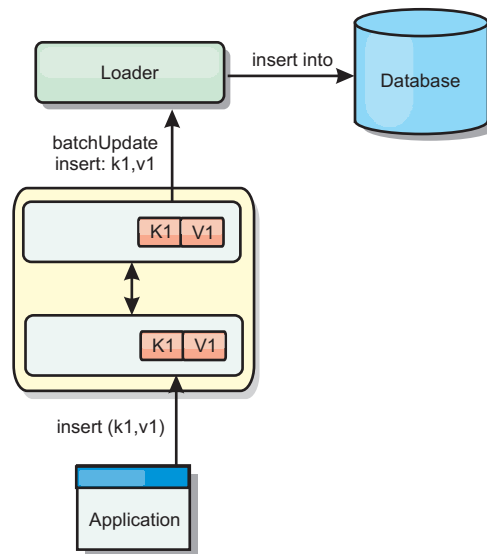


Figure 16. Write-through caching

Write-behind caching scenario

Database synchronization can be improved by writing changes asynchronously. This is known as a write-behind or write-back cache. Changes that would normally be written synchronously to the loader are instead buffered in eXtreme Scale and written to the database using a background thread. Write performance is significantly improved because the database operation is removed from the client transaction and the database writes can be compressed.

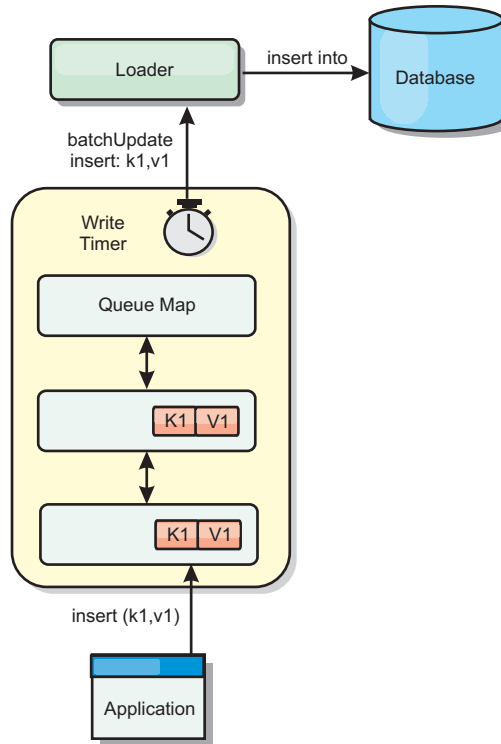


Figure 17. Write-behind caching

Write-behind caching

You can use write-behind caching to reduce the overhead that occurs when updating a database you are using as a back end.

Write-behind caching overview

Write-behind caching asynchronously queues updates to the Loader plug-in. You can improve performance by disconnecting updates, inserts, and removes for a map, the overhead of updating the back-end database. The asynchronous update is performed after a time-based delay (for example, five minutes) or an entry-based delay (1000 entries).

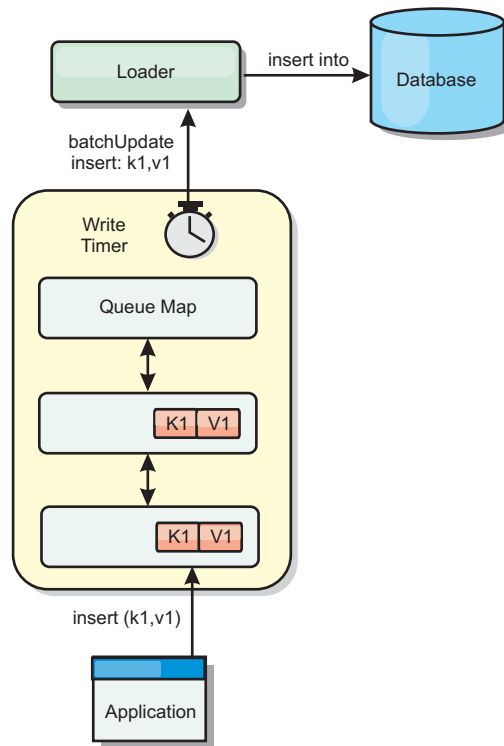


Figure 18. Write-behind caching

The write-behind configuration on a `BackingMap` creates a thread between the loader and the map. The loader then delegates data requests through the thread according to the configuration settings in the `BackingMap.setWriteBehind` method. When an eXtreme Scale transaction inserts, updates, or removes an entry from a map, a `LogElement` object is created for each of these records. These elements are sent to the write-behind loader and queued in a special `ObjectMap` called a queue map. Each backing map with the write-behind setting enabled has its own queue maps. A write-behind thread periodically removes the queued data from the queue maps and pushes them to the real back-end loader.

The write-behind loader only sends insert, update, and delete types of `LogElement` objects to the real loader. All other types of `LogElement` objects, for example, `EVICT` type, are ignored.

Write-behind support is an extension of the Loader plug-in, which you use to integrate eXtreme Scale with the database. For example, consult the [Configuring JPA loaders](#) information about configuring a JPA loader.

Benefits

Enabling write-behind support has the following benefits:

- **Back end failure isolation:** Write-behind caching provides an isolation layer from back end failures. When the back-end database fails, updates are queued in the queue map. The applications can continue driving transactions to eXtreme Scale. When the back end recovers, the data in the queue map is pushed to the back-end.
- **Reduced back end load:** The write-behind loader merges the updates on a key basis so only one merged update per key exists in the queue map. This merge decreases the number of updates to the back-end database.

- **Improved transaction performance:** Individual eXtreme Scale transaction times are reduced because the transaction does not need to wait for the data to be synchronized with the back-end.

Loaders

With a Loader plug-in, a data grid map can behave as a memory cache for data that is typically kept in a persistent store on either the same system or another system. Typically, a database or file system is used as the persistent store. A remote Java virtual machine (JVM) can also be used as the source of data, allowing hub-based caches to be built using eXtreme Scale. A loader has the logic for reading and writing data to and from a persistent store.

Overview

Loaders are backing map plug-ins that are invoked when changes are made to the backing map or when the backing map is unable to satisfy a data request (a cache miss). The Loader is invoked when the cache is unable to satisfy a request for a key, providing read-through capability and lazy-population of the cache. A loader also allows updates to the database when cache values change. All changes within a transaction are grouped together to allow the number of database interactions to be minimized. A TransactionCallback plug-in is used in conjunction with the loader to trigger the demarcation of the backend transaction. Using this plug-in is important when multiple maps are included in a single transaction or when transaction data is flushed to the cache without committing.

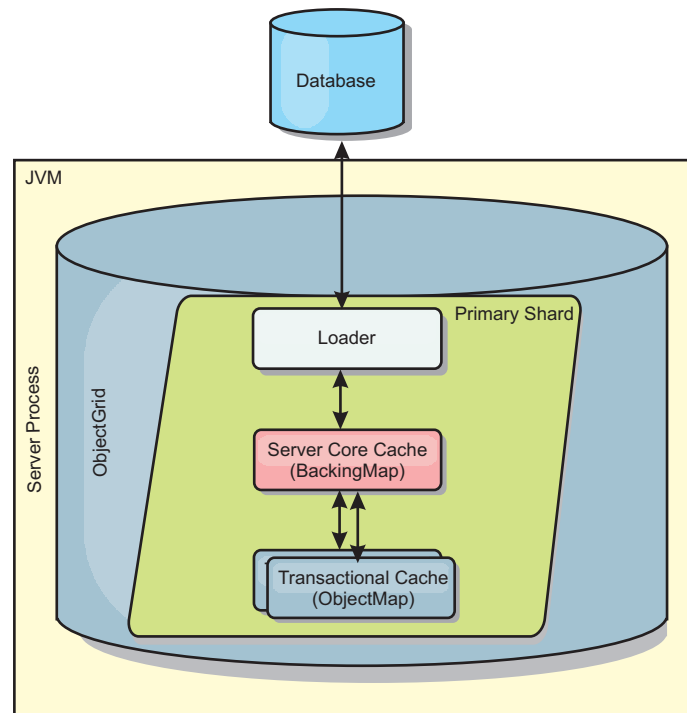


Figure 19. Loader

The loader can also use overqualified updates to avoid keeping database locks. By storing a version attribute in the cache value, the loader can see the before and after image of the value as it is updated in the cache. This value can then be used when updating the database or back end to verify that the data has not been updated. A Loader can also be configured to preload the data grid when it is

started. When partitioned, a Loader instance is associated with each partition. If the "Company" Map has ten partitions, there are ten Loader instances, one per primary partition. When the primary shard for the Map is activated, the preloadMap method for the loader is invoked synchronously or asynchronously which allows loading the map partition with data from the back-end to occur automatically. When invoked synchronously, all client transactions are blocked, preventing inconsistent access to the data grid. Alternatively, a client preloader can be used to load the entire data grid.

Two built-in loaders can greatly simplify integration with relational database back ends. The JPA loaders utilize the Object-Relational Mapping (ORM) capabilities of both the OpenJPA and Hibernate implementations of the Java Persistence API (JPA) specification. See "JPA Loaders" on page 365 for more information.

If you are using loaders in a multiple data center configuration, you must consider how revision data and cache consistency is maintained between the data grids. For more information, see "Loader considerations in a multi-master topology" on page 109.

Loader configuration

To add a Loader into the BackingMap configuration, you can use programmatic configuration or XML configuration. A loader has the following relationship with a backing map.

- A backing map can have only one loader.
- A client backing map (near cache) cannot have a loader.
- A loader definition can be applied to multiple backing maps, but each backing map has its own loader instance.

Data pre-loading and warm-up

In many scenarios that incorporate the use of a loader, you can prepare your data grid by pre-loading it with data.

When used as a complete cache, the data grid must hold all of the data and must be loaded before any clients can connect to it. When you are using a sparse cache, you can warm up the cache with data so that clients can have immediate access to data when they connect.

Two approaches exist for pre-loading data into the data grid: Using a Loader plug-in or using a client loader, as described in the following sections.

Loader plug-in

The loader plug-in is associated with each map and is responsible for synchronizing a single primary partition shard with the database. The preloadMap method of the loader plug-in is invoked automatically when a shard is activated. For example, if you have 100 partitions, 100 loader instances exist, each loading the data for its partition. When run synchronously, all clients are blocked until the preload has completed.

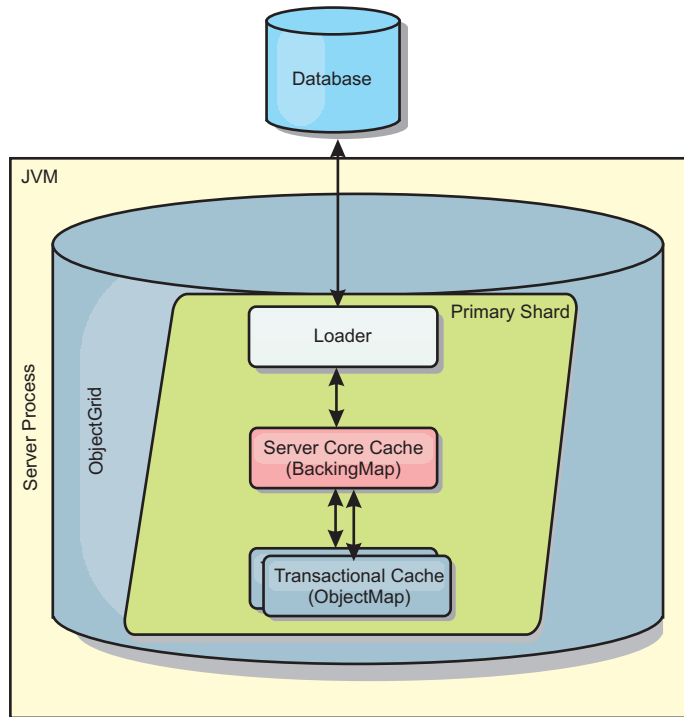


Figure 20. Loader plug-in

Client loader

A client loader is a pattern for using one or more clients to load the grid with data. Using multiple clients to load grid data can be effective when the partition scheme is not stored in the database. You can invoke client loaders manually or automatically when the data grid starts. Client loaders can optionally use the StateManager to set the state of the data grid to pre-load mode, so that clients are not able to access the grid while it is pre-loading the data. WebSphere eXtreme Scale includes a Java Persistence API (JPA)-based loader that you can use to automatically load the data grid with either the OpenJPA or Hibernate JPA providers. For more information about cache providers, see JPA level 2 (L2) cache plug-in.

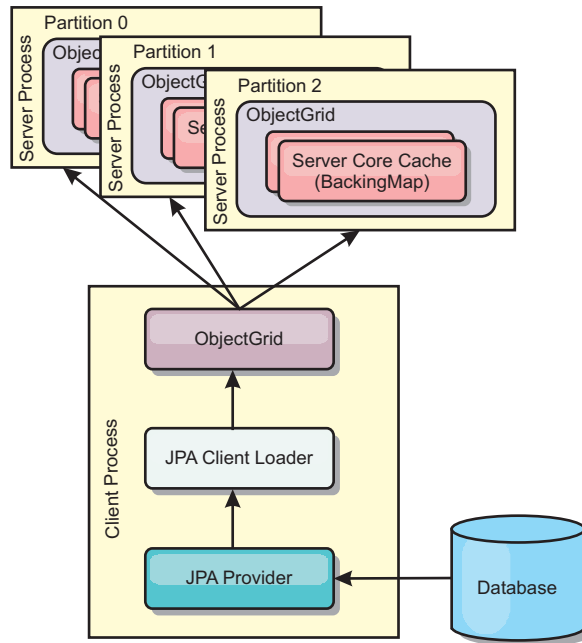


Figure 21. Client loader

Database synchronization techniques

When WebSphere eXtreme Scale is used as a cache, applications must be written to tolerate stale data if the database can be updated independently from an eXtreme Scale transaction. To serve as a synchronized in-memory database processing space, eXtreme Scale provides several ways of keeping the cache updated.

Database synchronization techniques

Periodic refresh

The cache can be automatically invalidated or updated periodically using the Java Persistence API (JPA) time-based database updater. The updater periodically queries the database using a JPA provider for any updates or inserts that have occurred since the previous update. Any changes identified are automatically invalidated or updated when used with a sparse cache. If used with a complete cache, the entries can be discovered and inserted into the cache. Entries are never removed from the cache.

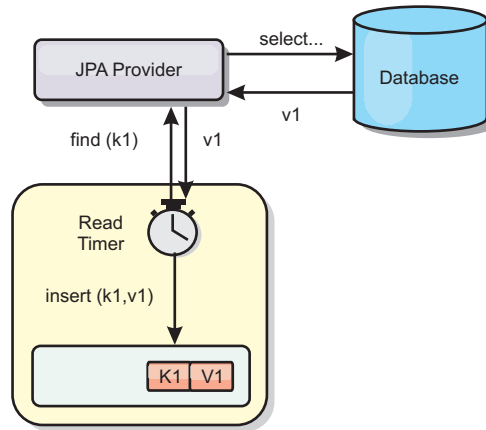


Figure 22. Periodic refresh

Eviction

Sparse caches can utilize eviction policies to automatically remove data from the cache without affecting the database. There are three built-in policies included in eXtreme Scale: time-to-live, least-recently-used, and least-frequently-used. All three policies can optionally evict data more aggressively as memory becomes constrained by enabling the memory-based eviction option.

Event-based invalidation

Sparse and complete caches can be invalidated or updated using an event generator such as Java Message Service (JMS). Invalidation using JMS can be manually tied to any process that updates the back-end using a database trigger. A JMS ObjectGridEventListener plug-in is provided in eXtreme Scale that can notify clients when the server cache has any changes. This can decrease the amount of time the client can see stale data.

Programmatic invalidation

The eXtreme Scale APIs allow manual interaction of the near and server cache using the `Session.beginNoWriteThrough()`, `ObjectMap.invalidate()` and `EntityManager.invalidate()` API methods. If a client or server process no longer needs a portion of the data, the invalidate methods can be used to remove data from the near or server cache. The `beginNoWriteThrough` method applies any `ObjectMap` or `EntityManager` operation to the local cache without calling the loader. If invoked from a client, the operation applies only to the near cache (the remote loader is not invoked). If invoked on the server, the operation applies only to the server core cache without invoking the loader.

Data invalidation

To remove stale cache data, you can use an administrative, event-based or programmatic invalidation mechanism.

8.5+

Administrative invalidation

You can use the web console or the `xscmd` utility to invalidate data based on the key. You can filter the cache data with a regular expression and then invalidate the data based on the regular expression.

Event-based invalidation

Sparse and complete caches can be invalidated or updated using an event generator such as Java Message Service (JMS). Invalidation using JMS can be manually tied to any process that updates the back-end using a database trigger. A JMS ObjectGridEventListener plug-in is provided in eXtreme Scale that can notify clients when the server cache changes. This type of notification decreases the amount of time the client can see stale data.

Event-based invalidation normally consists of the following three components.

- **Event queue:** An event queue stores the data change events. It could be a JMS queue, a database, an in-memory FIFO queue, or any kind of manifest as long as it can manage the data change events.
- **Event publisher:** An event publisher publishes the data change events to the event queue. An event publisher is usually an application you create or an eXtreme Scale plug-in implementation. The event publisher knows when the data is changed or it changes the data itself. When a transaction commits, events are generated for the changed data and the event publisher publishes these events to the event queue.
- **Event consumer:** An event consumer consumes data change events. The event consumer is usually an application to ensure the target grid data is updated with the latest change from other grids. This event consumer interacts with the event queue to get the latest data change and applies the data changes in the target grid. The event consumers can use eXtreme Scale APIs to invalidate stale data or update the grid with the latest data.

For example, JMSObjectGridEventListener has an option for a client-server model, in which the event queue is a designated JMS destination. All server processes are event publishers. When a transaction commits, the server gets the data changes and publishes them to the designated JMS destination. All the client processes are event consumers. They receive the data changes from the designated JMS destination and apply the changes to the client's near cache.

See the topic on enabling the client invalidation mechanism in the *Administration Guide* for more information.

Programmatic invalidation

The WebSphere eXtreme Scale APIs allow manual interaction of the near and server cache using the `Session.beginNoWriteThrough()`, `ObjectMap.invalidate()` and `EntityManager.invalidate()` API methods. If a client or server process no longer needs a portion of the data, the invalidate methods can be used to remove data from the near or server cache. The `beginNoWriteThrough` method applies any `ObjectMap` or `EntityManager` operation to the local cache without calling the loader. If invoked from a client, the operation applies only to the near cache (the remote loader is not invoked). If invoked on the server, the operation applies only to the server core cache without invoking the loader.

You can use programmatic invalidation with other techniques to determine when to invalidate the data. For example, this invalidation method uses event-based invalidation mechanisms to receive the data change events, and then uses APIs to invalidate the stale data.

Indexing

Use the `MapIndexPlugin` plug-in to build an index or several indexes on a `BackingMap` to support non-key data access.

Index types and configuration

The indexing feature is represented by the `MapIndexPlugin` plug-in or `Index` for short. The `Index` is a `BackingMap` plug-in. A `BackingMap` can have multiple `Index` plug-ins configured, as long as each one follows the `Index` configuration rules.

You can use the indexing feature to build one or more indexes on a `BackingMap`. An index is built from an attribute or a list of attributes of an object in the `BackingMap`. This feature provides a way for applications to find certain objects more quickly. With the indexing feature, applications can find objects with a specific value or within a range of values of indexed attributes.

Two types of indexing are possible: static and dynamic. With static indexing, you must configure the index plug-in on the `BackingMap` before initializing the `ObjectGrid` instance. You can do this configuration with XML or programmatic configuration of the `BackingMap`. Static indexing starts building an index during `ObjectGrid` initialization. The index is always synchronized with the `BackingMap` and ready for use. After the static indexing process starts, the maintenance of the index is part of the eXtreme Scale transaction management process. When transactions commit changes, these changes also update the static index, and index changes are rolled back if the transaction is rolled back.

With dynamic indexing, you can create an index on a `BackingMap` before or after the initialization of the containing `ObjectGrid` instance. Applications have life cycle control over the dynamic indexing process so that you can remove a dynamic index when it is no longer needed. When an application creates a dynamic index, the index might not be ready for immediate use because of the time it takes to complete the index building process. Because the amount of time depends upon the amount of data indexed, the `DynamicIndexCallback` interface is provided for applications that want to receive notifications when certain indexing events occur. These events include `ready`, `error`, and `destroy`. Applications can implement this callback interface and register with the dynamic indexing process.

If a `BackingMap` has an index plug-in configured, you can obtain the application index proxy object from the corresponding `ObjectMap`. Calling the `getIndex` method on the `ObjectMap` and passing in the name of the index plug-in returns the index proxy object. You must cast the index proxy object to an appropriate application index interface, such as `MapIndex`, `MapRangeIndex`, or a customized index interface. After obtaining the index proxy object, you can use methods defined in the application index interface to find cached objects.

The steps to use indexing are summarized in the following list:

- Add either static or dynamic index plug-ins into the `BackingMap`.
- Obtain an application index proxy object by issuing the `getIndex` method of the `ObjectMap`.
- Cast the index proxy object to an appropriate application index interface, such as `MapIndex`, `MapRangeIndex`, or a customized index interface.
- Use methods that are defined in application index interface to find cached objects.

The `HashIndex` class is the built-in index plug-in implementation that can support both of the built-in application index interfaces: `MapIndex` and `MapRangeIndex`. You also can create your own indexes. You can add `HashIndex` as either a static or dynamic index into the `BackingMap`, obtain either `MapIndex` or `MapRangeIndex` index proxy object, and use the index proxy object to find cached objects.

Default index

If you want to iterate through the keys in a local map, you can use the default index. This index does not require any configuration, but it must be used against the shard, using an agent or an `ObjectGrid` instance retrieved from the `ShardEvents.shardActivated(ObjectGrid shard)` method.

Data quality consideration

The results of index query methods only represent a snapshot of data at a point of time. No locks against data entries are obtained after the results return to the application. Application has to be aware that data updates may occur on a returned data set. For example, the application obtains the key of a cached object by running the `findAll` method of `MapIndex`. This returned key object is associated with a data entry in the cache. The application should be able to run the `get` method on `ObjectMap` to find an object by providing the key object. If another transaction removes the data object from the cache just before the `get` method is called, the returned result will be null.

Indexing performance considerations

One of the main objectives of the indexing feature is to improve overall `BackingMap` performance. If indexing is not used properly, the performance of the application might be compromised. Consider the following factors before using this feature.

- **The number of concurrent write transactions:** Index processing can occur every time a transaction writes data into a `BackingMap`. Performance degrades if many transactions are writing data into the map concurrently when an application attempts index query operations.
- **The size of the result set that is returned by a query operation:** As the size of the resultset increases, the query performance declines. Performance tends to degrade when the size of the result set is 15% or more of the `BackingMap`.
- **The number of indexes built over the same `BackingMap`:** Each index consumes system resources. As the number of the indexes built over the `BackingMap` increases, performance decreases.

The indexing function can improve `BackingMap` performance drastically. Ideal cases are when the `BackingMap` has mostly read operations, the query result set is of a small percentage of the `BackingMap` entries, and only few indexes are built over the `BackingMap`.

Planning multiple data center topologies

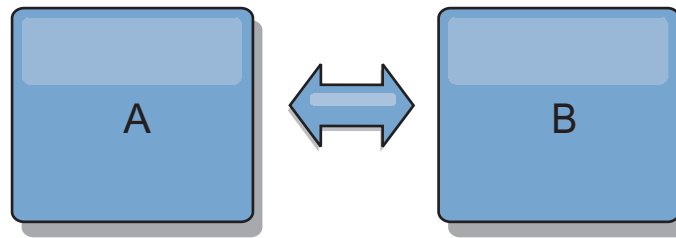
Using multi-master asynchronous replication, two or more data grids can become exact copies of each other. Each data grid is hosted in an independent catalog service domain, with its own catalog service, container servers, and a unique name. With multi-master asynchronous replication, you can use links to connect a collection of catalog service domains. The catalog service domains are then synchronized using replication over the links. You can construct almost any topology through the definition of links between the catalog service domains.

Topologies for multimaster replication

You have several different options when choosing the topology for your deployment that incorporates multimaster replication.

Links connecting catalog service domains

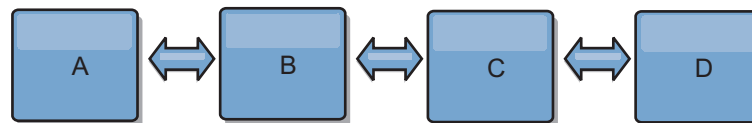
A replication data grid infrastructure is a connected graph of catalog service domains with bidirectional links among them. With a link, two catalog service domains can communicate data changes. For example, the simplest topology is a pair of catalog service domains with a single link between them. The catalog service domains are named alphabetically: A, B, C, and so on, from the left. A link can cross a wide area network (WAN), spanning large distances. Even if the link is interrupted, you can still change data in either catalog service domain. The topology reconciles changes when the link reconnects the catalog service domains. Links automatically try to reconnect if the network connection is interrupted.



After you set up the links, the product first tries to make every catalog service domain identical. Then, eXtreme Scale tries to maintain the identical conditions as changes occur in any catalog service domain. The goal is for each catalog service domain to be an exact mirror of every other catalog service domain connected by the links. The replication links between the catalog service domains help ensure that any changes made in one catalog service domain are copied to the other catalog service domains.

Line topologies

Although it is such a simple deployment, a line topology demonstrates some qualities of the links. First, it is not necessary for a catalog service domain to be connected directly to every other catalog service domain to receive changes. The catalog service domain B pulls changes from catalog service domain A. The catalog service domain C receives changes from catalog service domain A through catalog service domain B, which connects catalog service domains A and C. Similarly, catalog service domain D receives changes from the other catalog service domains through catalog service domain C. This ability spreads the load of distributing changes away from the source of the changes.



Notice that if catalog service domain C fails, the following actions would occur:

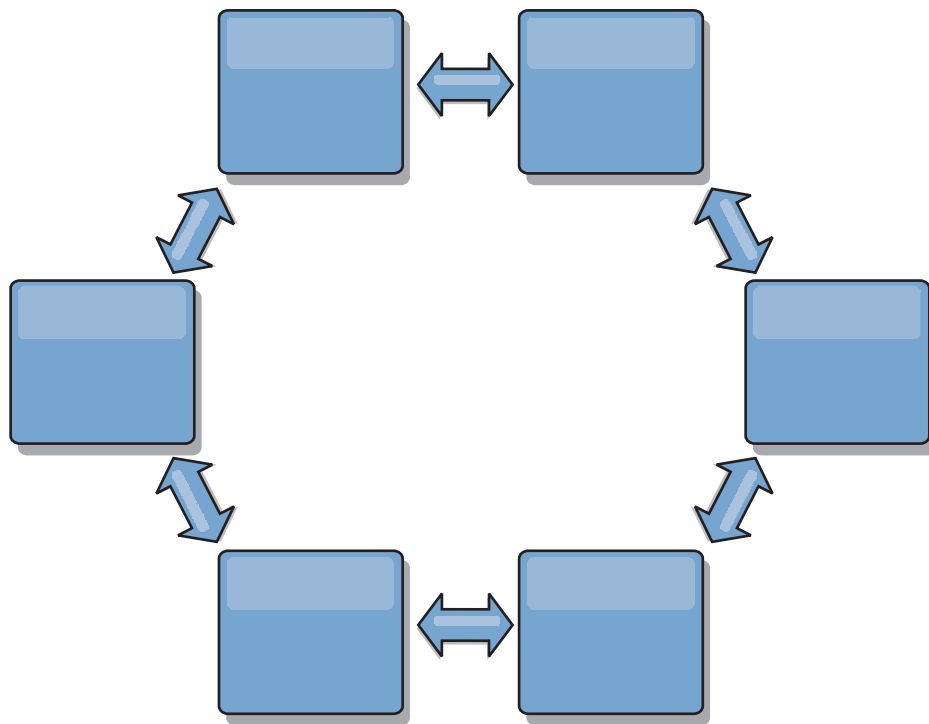
1. catalog service domain D would be orphaned until catalog service domain C was restarted
2. catalog service domain C would synchronize itself with catalog service domain B, which is a copy of catalog service domain A

- catalog service domain D would use catalog service domain C to synchronize itself with changes on catalog service domain A and B. These changes initially occurred while catalog service domain D was orphaned (while catalog service domain C was down).

Ultimately, catalog service domains A, B, C, and D would all become identical to one other again.

Ring topologies

Ring topologies are an example of a more resilient topology. When a catalog service domain or a single link fails, the surviving catalog service domains can still obtain changes. The catalog service domains travel around the ring, away from the failure. Each catalog service domain has at most two links to other catalog service domains, no matter how large the ring topology. The latency to propagate changes can be large. Changes from a particular catalog service domain might need to travel through several links before all the catalog service domains have the changes. A line topology has the same characteristic.

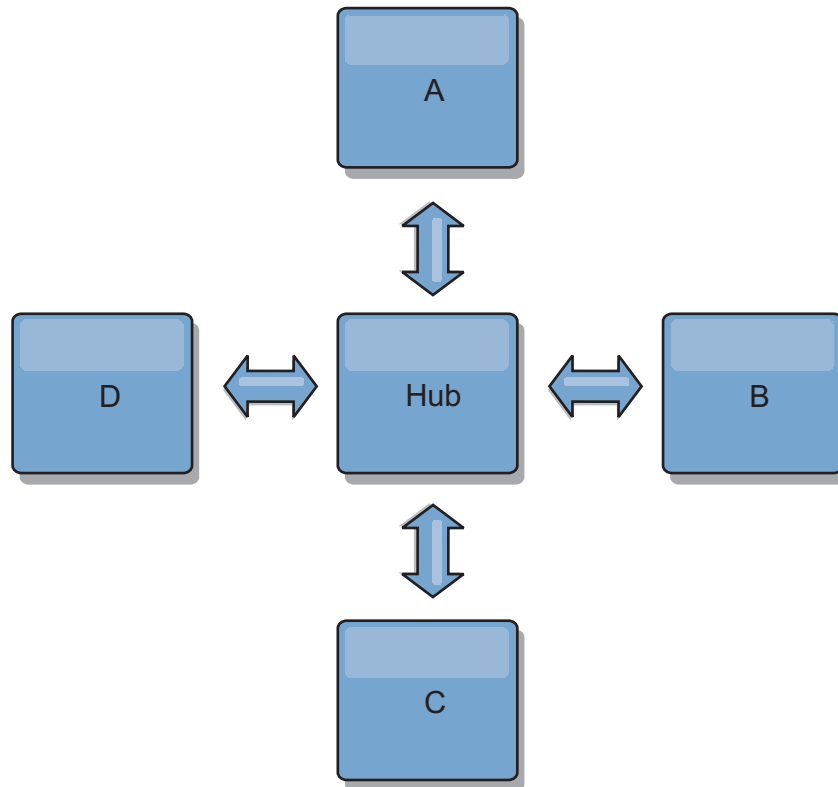


You can also deploy a more sophisticated ring topology, with a root catalog service domain at the center of the ring. The root catalog service domain functions as the central point of reconciliation. The other catalog service domains act as remote points of reconciliation for changes occurring in the root catalog service domain. The root catalog service domain can arbitrate changes among the catalog service domains. If a ring topology contains more than one ring around a root catalog service domain, the catalog service domain can only arbitrate changes among the innermost ring. However, the results of the arbitration spread throughout the catalog service domains in the other rings.

Hub-and-spoke topologies

With a hub-and-spoke topology, changes travel through a hub catalog service domain. Because the hub is the only intermediate catalog service domain that is

specified, hub-and-spoke topologies have lower latency. The hub catalog service domain is connected to every spoke catalog service domain through a link. The hub distributes changes among the catalog service domains. The hub acts as a point of reconciliation for collisions. In an environment with a high update rate, the hub might require run on more hardware than the spokes to remain synchronized. WebSphere eXtreme Scale is designed to scale linearly, meaning you can make the hub larger, as needed, without difficulty. However, if the hub fails, then changes are not distributed until the hub restarts. Any changes on the spoke catalog service domains will be distributed after the hub is reconnected.



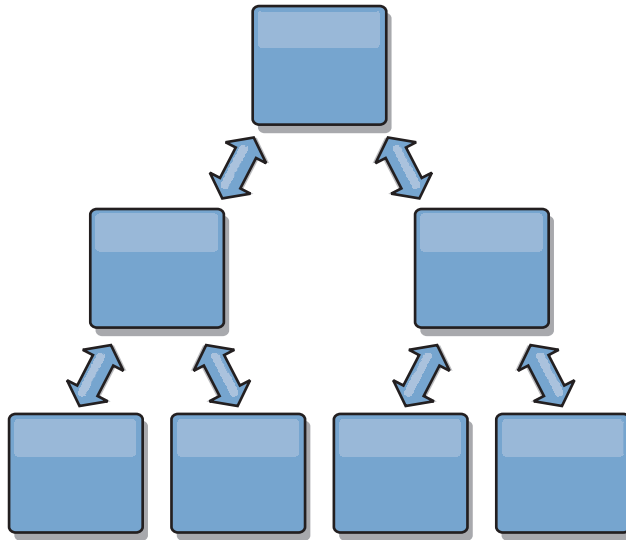
You can also use a strategy with fully replicated clients, a topology variation which uses a pair of servers that are running as a hub. Every client creates a self-contained single container data grid with a catalog in the client JVM. A client uses its data grid to connect to the hub catalog. This connection causes the client to synchronize with the hub as soon as the client obtains a connection to the hub.

Any changes made by the client are local to the client, and are replicated asynchronously to the hub. The hub acts as an arbitration catalog service domain, distributing changes to all connected clients. The fully replicated clients topology provides a reliable L2 cache for an object relational mapper, such as OpenJPA. Changes are distributed quickly among client JVMs through the hub. If the cache size can be contained within the available heap space, the topology is a reliable architecture for this style of L2.

Use multiple partitions to scale the hub catalog service domain on multiple JVMs, if necessary. Because all of the data still must fit in a single client JVM, multiple partitions increase the capacity of the hub to distribute and arbitrate changes. However, having multiple partitions does not change the capacity of a single catalog service domain.

Tree topologies

You can also use an acyclic directed tree. An acyclic tree has no cycles or loops, and a directed setup limits links to existing only between parents and children. This configuration is useful for topologies that have many catalog service domains. In these topologies, it is not practical to have a central hub that is connected to every possible spoke. This type of topology can also be useful when you must add child catalog service domains without updating the root catalog service domain.



A tree topology can still have a central point of reconciliation in the root catalog service domain. The second level can still function as a remote point of reconciliation for changes occurring in the catalog service domain beneath them. The root catalog service domain can arbitrate changes between the catalog service domains on the second level only. You can also use N-ary trees, each of which have N children at each level. Each catalog service domain connects out to n links.

Fully replicated clients

This topology variation involves a pair of servers that are running as a hub. Every client creates a self-contained single container data grid with a catalog in the client JVM. A client uses its data grid to connect to the hub catalog, causing the client to synchronize with the hub as soon as the client obtains a connection to the hub.

Any changes made by the client are local to the client, and are replicated asynchronously to the hub. The hub acts as an arbitration catalog service domain, distributing changes to all connected clients. The fully replicated clients topology provides a good L2 cache for an object relational mapper, such as OpenJPA. Changes are distributed quickly among client JVMs through the hub. As long as the cache size can be contained within the available heap space of the clients, this topology is a good architecture for this style of L2.

Use multiple partitions to scale the hub catalog service domain on multiple JVMs, if necessary. Because all of the data still must fit in a single client JVM, using multiple partitions increases the capacity of the hub to distribute and arbitrate changes, but it does not change the capacity of a single catalog service domain.

Configuration considerations for multi-master topologies

Consider the following issues when you are deciding whether and how to use multi-master replication topologies.

- **Map set requirements**

Map sets must have the following characteristics to replicate changes across catalog service domain links:

- The ObjectGrid name and map set name within a catalog service domain must match the ObjectGrid name and map set name of other catalog service domains. For example, ObjectGrid "og1" and map set "ms1" must be configured in catalog service domain A and catalog service domain B to replicate the data in the map set between the catalog service domains.
- Is a FIXED_PARTITION data grid. PER_CONTAINER data grids cannot be replicated.
- Has the same number of partitions in each catalog service domain. The map set might or might not have the same number and types of replicas.
- Has the same data types being replicated in each catalog service domain.
- Contains the same maps and dynamic map templates in each catalog service domain.
- Does not use entity manager. A map set containing an entity map is not replicated across catalog service domains.
- Does not use write-behind caching support. A map set containing a map that is configured with write-behind support is not replicated across catalog service domains.

Any map sets with the preceding characteristics begin to replicate after the catalog service domains in the topology have been started.

- **Class loaders with multiple catalog service domains**

Catalog service domains must have access to all classes that are used as keys and values. Any dependencies must be reflected in all class paths for data grid container Java virtual machines (JVM) for all domains. If a CollisionArbiter plug-in retrieves the value for a cache entry, then the classes for the values must be present for the domain that is starting the arbiter.

Loader considerations in a multi-master topology

When you are using loaders in a multi-master topology, you must consider the possible collision and revision information maintenance challenges. The data grid maintains revision information about the items in the data grid so that collisions can be detected when other primary shards in the configuration write entries to the data grid. When entries are added from a loader, this revision information is not included and the entry takes on a new revision. Because the revision of the entry seems to be a new insert, a false collision could occur if another primary shard also changes this state or pulls the same information in from a loader.

Replication changes invoke the get method on the loader with a list of the keys that are not already in the data grid but are going to be changed during the replication transaction. When the replication occurs, these entries are collision entries. When the collisions are arbitrated and the revision is applied then a batch update is called on the loader to apply the changes to the database. All of the maps that were changed in the revision window are updated in the same transaction.

Preload conundrum

Consider a two data center topology with data center A and data center B. Both data centers have independent databases, but only data center A has a data grid that is running. When you establish a link between the data centers for a multi-master configuration, the data grids in data center A begin pushing data to the new data grids in data center B, causing a collision with every entry. Another major issue that occurs is with any data that is in the database in data center B but not in the database in data center A. These rows are not populated and arbitrated, resulting in inconsistencies that are not resolved.

Solution to the preload conundrum

Because data that resides only in the database cannot have revisions, you must always fully preload the data grid from the local database before establishing the multi-master link. Then, both data grids can revision and arbitrate the data, eventually reaching a consistent state.

Sparse cache conundrum

With a sparse cache, the application first attempts to find data in the data grid. If the data is not in the data grid, the data is searched for in the database using the loader. Entries are evicted from the data grid periodically to maintain a small cache size.

This cache type can be problematic in a multi-master configuration scenario because the entries within the data grid have revisioning metadata that help detect when collisions occur and which side has made changes. When links between the data centers are not working, one data center can update an entry and then eventually update the database and invalidate the entry in the data grid. When the link recovers, the data centers attempt to synchronize revisions with each other. However, because the database was updated and the data grid entry was invalidated, the change is lost from the perspective of the data center that went down. As a result, the two sides of the data grid are out of synch and are not consistent.

Solution to the sparse cache conundrum

Hub and spoke topology:

You can run the loader only in the hub of a hub and spoke topology, maintaining consistency of the data while scaling out the data grid. However, if you are considering this deployment, note that the loaders can allow the data grid to be partially loaded, meaning that an evictor has been configured. If the spokes of your configuration are sparse caches but have no loader, then any cache misses have no way to retrieve data from the database. Because of this restriction, you should use a fully populated cache topology with a hub and spoke configuration.

Invalidations and eviction

Invalidation creates inconsistency between the data grid and the database. Data can be removed from the data grid either programmatically or with eviction. When you develop your application, you must be aware that revision handling does not replicate changes that are invalidated, resulting in inconsistencies between primary shards.

Invalidation events are not cache state changes and do not result in replication. Any configured evictors run independently from other evictors in the configuration. For example, you might have one evictor configured for a memory threshold in one catalog service domain, but a different type of less aggressive evictor in your other linked catalog service domain. When data grid entries are removed due to the memory threshold policy, the entries in the other catalog service domain are not affected.

Database updates and data grid invalidation

Problems occur when you update the database directly in the background while calling the invalidation on the data grid for the updated entries in a multi-master configuration. This problem occurs because the data grid cannot replicate the change to the other primary shards until some type of cache access moves the entry into the data grid.

Multiple writers to a single logical database

When you are using a single database with multiple primary shards that are connected through a loader, transactional conflicts result. Your loader implementation must specially handle these types of scenarios.

Mirroring data using multi-master replication

You can configure independent databases that are connected to independent catalog service domains. In this configuration, the loader can push changes from one data center to the other data center.

Design considerations for multi-master replication

When implementing multi-master replication, you must consider aspects in your design such as: arbitration, linking, and performance.

Arbitration considerations in topology design

Change collisions might occur if the same records can be changed simultaneously in two places. Set up each catalog service domain to have about the same amount of processor, memory, network resources. You might observe that catalog service domains performing change collision handling (arbitration) use more resources than other catalog service domains. Collisions are detected automatically. They are handled with one of two mechanisms:

- **Default collision arbiter:** The default protocol is to use the changes from the lexically lowest named catalog service domain. For example, if catalog service domain A and B generate a conflict for a record, then the change from catalog service domain B is ignored. Catalog service domain A keeps its version and the record in catalog service domain B is changed to match the record from catalog service domain A. This behavior applies as well for applications where users or sessions are normally bound or have affinity with one of the data grids.
- **Custom collision arbiter:** Applications can provide a custom arbiter. When a catalog service domain detects a collision, it starts the arbiter. For information about developing a useful custom arbiter, see “Developing custom arbiters for multi-master replication” on page 283.

For topologies in which collisions are possible, consider implementing a hub-and-spoke topology or a tree topology. These two topologies are conducive to avoiding constant collisions, which can happen in the following scenarios:

1. Multiple catalog service domains experience a collision
2. Each catalog service domain handles the collision locally, producing revisions
3. The revisions collide, resulting in revisions of revisions

To avoid collisions, choose a specific catalog service domain, called an *arbitration catalog service domain* as the collision arbiter for a subset of catalog service domains. For example, a hub-and-spoke topology might use the hub as the collision handler. The spoke collision handler ignores any collisions that are detected by the spoke catalog service domains. The hub catalog service domain creates revisions, preventing unexpected collision revisions. The catalog service domain that is assigned to handle collisions must link to all of the domains for which it is responsible for handling collisions. In a tree topology, any internal parent domains handle collisions for their immediate children. In contrast, if you use a ring topology, you cannot designate one catalog service domain in the ring as the arbiter.

The following table summarizes the arbitration approaches that are most compatible with various topologies.

Table 2. Arbitration approaches. This table states whether application arbitration is compatible with various technologies.

Topology	Application Arbitration?	Notes
A line of two catalog service domains	Yes	Choose one catalog service domain as the arbiter.
A line of three catalog service domains	Yes	The middle catalog service domain must be the arbiter. Think of the middle catalog service domain as the hub in a simple hub-and-spoke topology.
A line of more than three catalog service domains	No	Application arbitration is not supported.
A hub with N spokes	Yes	Hub with links to all spokes must be the arbitration catalog service domain.
A ring of N catalog service domains	No	Application arbitration is not supported.
An acyclic, directed tree (n-ary tree)	Yes	All root nodes must rate their direct descendants only.

Linking considerations in topology design

Ideally, a topology includes the minimum number of links while optimizing trade-offs among change latency, fault tolerance, and performance characteristics.

- **Change latency**

Change latency is determined by the number of intermediate catalog service domains a change must go through before arriving at a specific catalog service domain.

A topology has the best change latency when it eliminates intermediate catalog service domains by linking every catalog service domain to every other catalog service domain. However, a catalog service domain must perform replication work in proportion to its number of links. For large topologies, the sheer number of links to be defined can cause an administrative burden.

The speed at which a change is copied to other catalog service domains depends on additional factors, such as:

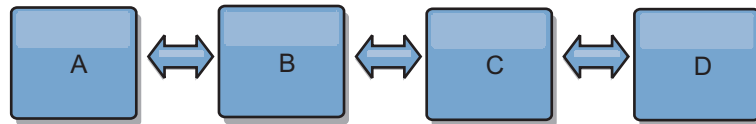
- Processor and network bandwidth on the source catalog service domain
- The number of intermediate catalog service domains and links between the source and target catalog service domain
- The processor and network resources available to the source, target, and intermediate catalog service domains

- **Fault tolerance**

Fault tolerance is determined by how many paths exist between two catalog service domains for change replication.

If you have only one link between a given pair of catalog service domains, a link failure disallows propagation of changes. Similarly, changes are not propagated between catalog service domains if any of the intermediate domains experiences link failure. Your topology could have a single link from one catalog service domain to another such that the link passes through intermediate domains. If so, then changes are not propagated if any of the intermediate catalog service domains is down.

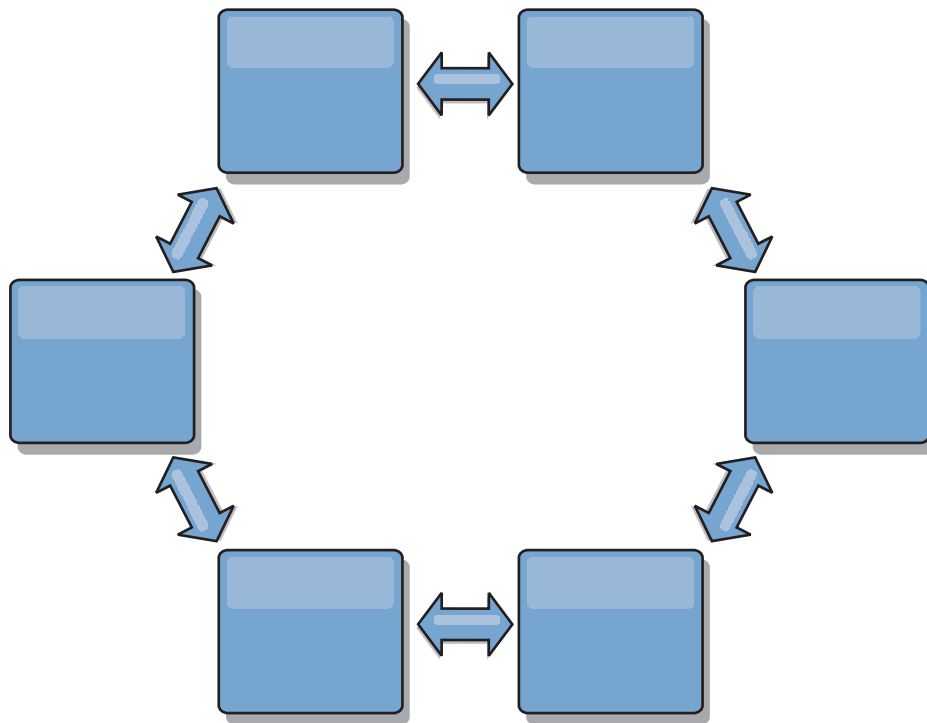
Consider the line topology with four catalog service domains A, B, C, and D:



If any of these conditions hold, Domain D does not see any changes from A:

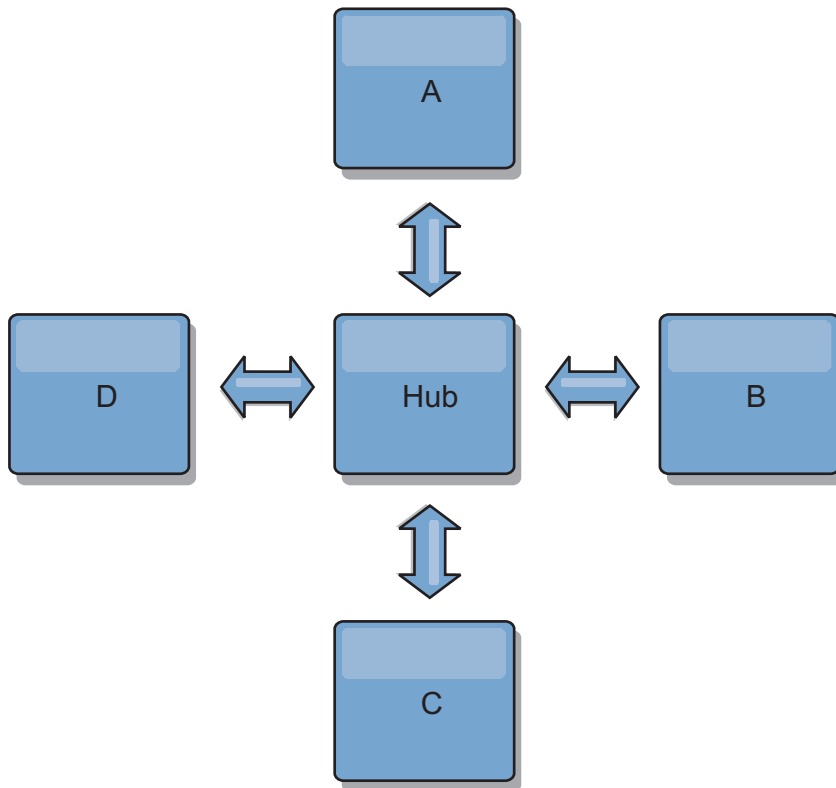
- Domain A is up and B is down
- Domains A and B are up and C is down
- The link between A and B is down
- The link between B and C is down
- The link between C and D is down

In contrast, with a ring topology, each catalog service domain can receive changes from either direction.



For example, if a given catalog service in your ring topology is down, then the two adjacent domains can still pull changes directly from each other.

All changes are propagated through the hub. Thus, as opposed to the line and ring topologies, the hub-and-spoke design is susceptible to break down if the hub fails.



A single catalog service domain is resilient to a certain amount of service loss. However, larger failures such as wide network outages or loss of links between physical data centers can disrupt any of your catalog service domains.

- **Linking and performance**

The number of links defined on a catalog service domain affects performance. More links use more resources and replication performance can drop as a result. The ability to retrieve changes for a domain A through other domains effectively offloads domain A from replicating its transactions everywhere. The change distribution load on a domain is limited by the number of links it uses, not how many domains are in the topology. This load property provides scalability, so the domains in the topology can share the burden of change distribution.

A catalog service domain can retrieve changes indirectly through other catalog service domains. Consider a line topology with five catalog service domains.

A <=> B <=> C <=> D <=> E

- A pulls changes from B, C, D, and E through B
- B pulls changes from A and C directly, and changes from D and E through C
- C pulls changes from B and D directly, and changes from A through B and E through D
- D pulls changes from C and E directly, and changes from A and B through C
- E pulls changes from D directly, and changes from A, B, and C through D

The distribution load on catalog service domains A and E is lowest, because they each have a link only to a single catalog service domain. Domains B, C, and D each have a link to two domains. Thus, the distribution load on domains B, C, and D is double the load on domains A and E. The workload depends on the number of links in each domain, not on the overall number of domains in the topology. Thus, the described distribution of loads would remain constant, even if the line contained 1000 domains.

Multi-master replication performance considerations

Take the following limitations into account when using multi-master replication topologies:

- **Change distribution tuning**, as discussed in the previous section.
- **Replication link performance** WebSphere eXtreme Scale creates a single TCP/IP socket between any pair of JVMs. All traffic between the JVMs occurs through the single socket, including traffic from multi-master replication. Catalog service domains are hosted on at least n container JVMs, providing at least n TCP links to peer catalog service domains. Thus, the catalog service domains with larger numbers of containers have higher replication performance levels. More containers require more processor and network resources.
- **TCP sliding window tuning and RFC 1323** RFC 1323 support on both ends of a link yields more data for a round trip. This support results in higher throughput, expanding the capacity of the window by a factor of about 16,000.

Recall that TCP sockets use a sliding window mechanism to control the flow of bulk data. This mechanism typically limits the socket to 64 KB for a round-trip interval. If the round-trip interval is 100 ms, then the bandwidth is limited to 640 KB/second without additional tuning. Fully using the bandwidth available on a link might require tuning that is specific to an operating system. Most operating systems include tuning parameters, including RFC 1323 options, to enhance throughput over high-latency links.

Several factors can affect replication performance:

- The speed at which eXtreme Scale retrieves changes.

- The speed at which eXtreme Scale can service retrieve replication requests.
- The sliding window capacity.
- With network buffer tuning on both sides of a link, eXtreme Scale retrieves changes over the socket efficiently.
- **Object Serialization** All data must be serializable. If a catalog service domain is not using COPY_TO_BYTES, then the catalog service domain must use Java serialization or ObjectTransformers to optimize serialization performance.
- **Compression** WebSphere eXtreme Scale compresses all data sent between catalog service domains by default. Disabling compression is not currently available.
- **Memory tuning** The memory usage for a multi-master replication topology is largely independent of the number of catalog service domains in the topology. Multi-master replication adds a fixed amount of processing per Map entry to handle versioning. Each container also tracks a fixed amount of data for each catalog service domain in the topology. A topology with two catalog service domains uses approximately the same memory as a topology with 50 catalog service domains. WebSphere eXtreme Scale does not use replay logs or similar queues in its implementation. Thus, there is no recovery structure ready in the case that a replication link is unavailable for a substantial period and later restarts.

Planning to develop WebSphere eXtreme Scale applications

Set up your development environment and learn where to find details about available programming interfaces.

API overview

WebSphere eXtreme Scale provides several features that are accessed programmatically using the Java programming language through application programming interfaces (APIs) and system programming interfaces.

WebSphere eXtreme Scale APIs

When you are using eXtreme Scale APIs, you must distinguish between transactional and non-transactional operations. A transactional operation is an operation that is performed within a transaction. ObjectMap, EntityManager, Query, and DataGrid API are transactional APIs that are contained inside the Session that is a transactional container. Non-transactional operations have nothing to do with a transaction, such as configuration operations.

The ObjectGrid, BackingMap, and plug-in APIs are non-transactional. The ObjectGrid, BackingMap, and other configuration APIs are categorized as ObjectGrid Core API. Plug-ins are for customizing the cache to achieve the functions that you want, and are categorized as the System Programming API. A plug-in in eXtreme Scale is a component that provides a certain type of function to the pluggable eXtreme Scale components that include ObjectGrid and BackingMap. A feature represents a specific function or characteristic of an eXtreme Scale component, including ObjectGrid, Session, BackingMap, ObjectMap, and so on. Typically, features are configurable with configuration APIs. Plug-ins can be built-in, but might require that you develop your own plug-ins in some situations.

You can normally configure the ObjectGrid and BackingMap to meet your application requirements. When the application has special requirements, consider using specialized plug-ins. WebSphere eXtreme Scale might have built-in plug-ins

that meet your requirements. For example, if you need a peer-to-peer replication model between two local ObjectGrid instances or two distributed eXtreme Scale grids, the built-in JMSObjectGridEventListener is available. If none of the built-in plug-ins can solve your business problems, refer to the System Programming API to provide your own plug-ins.

ObjectMap is a simple map-based API. If the cached objects are simple and no relationship is involved, the ObjectMap API is ideal for your application. If object relationships are involved, use the EntityManager API, which supports graph-like relationships.

Query is a powerful mechanism for finding data in the ObjectGrid. Both Session and EntityManager provide the traditional query capability.

The DataGrid API is a powerful computing capability in a distributed eXtreme Scale environment that involves many machines, replicas, and partitions. Applications can run business logic in parallel in all of the nodes in the distributed eXtreme Scale environment. The application can obtain the DataGrid API through the ObjectMap API.

The WebSphere eXtreme Scale REST data service is a Java HTTP service that is compatible with Microsoft WCF Data Services (formally ADO.NET Data Services) and implements the Open Data Protocol (OData). The REST data service allows any HTTP client to access an eXtreme Scale grid. It is compatible with the WCF Data Services support that is supplied with the Microsoft .NET Framework 3.5 SP1. RESTful applications can be developed with the rich tooling provided by Microsoft Visual Studio 2008 SP1. For more details, refer to the eXtreme Scale REST data service user guide.

Plug-ins overview

A WebSphere eXtreme Scale plug-in is a component that provides a certain type of function to the pluggable components that include ObjectGrid and BackingMap. WebSphere eXtreme Scale provides several plug points to allow applications and cache providers to integrate with various data stores, alternative client APIs and to improve overall performance of the cache. The product ships with several default, prebuilt plug-ins, but you can also build custom plug-ins with the application.

All plug-ins are concrete classes that implement one or more eXtreme Scale plug-in interfaces. These classes are then instantiated and invoked by the ObjectGrid at appropriate times. The ObjectGrid and BackingMaps each allow custom plug-ins to be registered.

ObjectGrid plug-ins

The following plug-ins are available for an ObjectGrid instance. If the plug-in is server side only, the plug-ins are removed on the client ObjectGrid and BackingMap instances. The ObjectGrid and BackingMap instances are only on the server.

- **TransactionCallback:** A TransactionCallback plug-in provides transaction life cycle events. If the TransactionCallback plug-in is the built-in JPATxCallback (com.ibm.websphere.objectgrid.jpa.JPATxCallback) class implementation, then the plug-in is server side only. However, the subclasses of the JPATxCallback class are not server side only.
- **ObjectGridEventListener:** An ObjectGridEventListener plug-in provides ObjectGrid life cycle events for the ObjectGrid, shards, and transactions.

- **ObjectGridLifecycleListener:** An ObjectGridLifecycleListener plug-in provides ObjectGrid life cycle events for the ObjectGrid instance. The ObjectGridLifecycleListener plug-in can be used as an optional mixin interface for all other ObjectGrid plug-ins.
- **ObjectGridPlugin:** An ObjectGridPlugin is an optional mix-in interface that provides extended life cycle management events for all other ObjectGrid plug-ins.
- **SubjectSource, ObjectGridAuthorization, SubjectValidation:** eXtreme Scale provides several security endpoints to allow custom authentication mechanisms to be integrated with eXtreme Scale. (Server side only)



Common ObjectGrid plug-in requirements

The ObjectGrid instantiates and initializes plug-in instances using JavaBeans conventions. All of the previous plug-in implementations have the following requirements:

- The plug-in class must be a top-level public class.
- The plug-in class must provide a public, no-argument constructor.
- The plug-in class must be available in the class path for both servers and clients (as appropriate).
- Attributes must be set using the JavaBeans style property methods.
- Plug-ins, unless specifically noted, are registered before ObjectGrid initializes and cannot be changed after the ObjectGrid is initialized.

BackingMap plug-ins

The following plug-ins are available for a BackingMap:

- **Evictor:** An evictor plug-in is a default mechanism is provided for evicting cache entries and a plug-in for creating custom evictors.
-  **ObjectTransformer:** An ObjectTransformer plug-in allows you to serialize, deserialize, and copy objects in the cache. The ObjectTransformer interface has been replaced by the DataSerializer plug-ins, which you can use to efficiently store arbitrary data in WebSphere eXtreme Scale so that existing product APIs can efficiently interact with your data.
-  **OptimisticCallback:** An OptimisticCallback plug-in allows you to customize versioning and comparison operations of cache objects when you are using the optimistic lock strategy. The OptimisticCallback plug-in has been replaced by the ValueDataSerializer.Versionable interface, which you can implement when you use the DataSerializer plug-in with the COPY_TO_BYTES copy mode or when you use the @Version annotation with the EntityManager API.
- **MapEventListener:** A MapEventListener plug-in provides callback notifications and significant cache state changes that occur for a BackingMap.
- **BackingMapLifecycleListener:** A BackingMapLifecycleListener plug-in provides BackingMap life cycle events for the BackingMap instance. The BackingMapLifecycleListener plug-in can be used as an optional mix-in interface for all other BackingMap plug-ins.
- **BackingMapPlugin:** A BackingMapPlugin is an optional mix-in interface that provides extended life cycle management events for all other BackingMap plug-ins.

- **Indexing:** Use the indexing feature, which is represented by the MapIndexplug-in plug-in, to build an index or several indexes on a BackingMap map to support non-key data access.
- **Loader:** A Loader plug-in on an ObjectGrid map acts as a memory cache for data that is typically kept in a persistent store on either the same system or some other system. (Server side only)
- **MapSerializerPlugin:** A MapSerializerPlugin allows you to serialize and inflate Java objects and non-Java data in the cache. It is used with the DataSerializer mix-in interfaces, allowing robust and flexible options for high-performance applications.

REST data services overview

The WebSphere eXtreme Scale REST data service is a Java HTTP service that is compatible with Microsoft WCF Data Services (formally ADO.NET Data Services) and implements the Open Data Protocol (OData). Microsoft WCF Data Services is compatible with this specification when using Visual Studio 2008 SP1 and the .NET Framework 3.5 SP1.

Compatibility requirements

The REST data service allows any HTTP client to access a data grid. The REST data service is compatible with the WCF Data Services support supplied with the Microsoft .NET Framework 3.5 SP1. RESTful applications can be developed with the rich tooling provided by Microsoft Visual Studio 2008 SP1. The figure provides an overview of how WCF Data Services interacts with clients and databases.

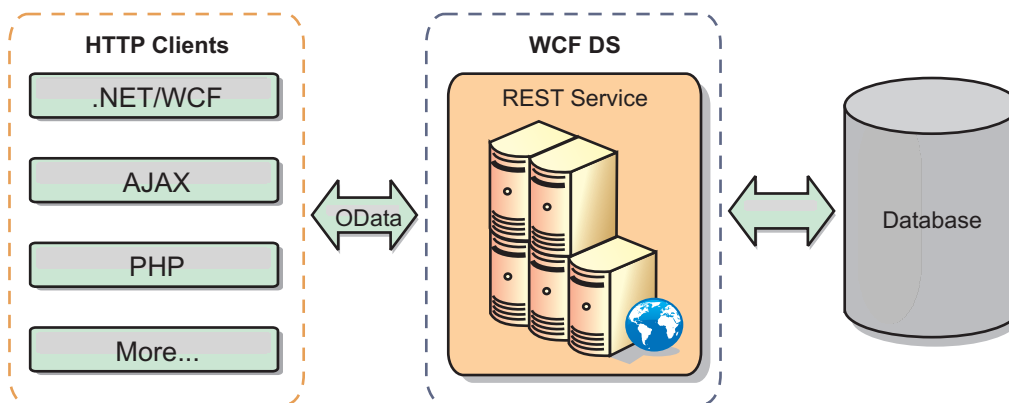


Figure 23. Microsoft WCF Data Services

WebSphere eXtreme Scale includes a function-rich API set for Java clients. As shown in the following figure, the REST data service is a gateway between HTTP clients and the WebSphere eXtreme Scale data grid, communicating with the grid through an WebSphere eXtreme Scale client. The REST data service is a Java servlet, which allows flexible deployments for common Java Platform, Enterprise Edition (JEE) platforms, such as WebSphere Application Server. The REST data service communicates with the WebSphere eXtreme Scale data grid using the WebSphere eXtreme Scale Java APIs. It allows WCF Data Services clients or any other client that can communicate with HTTP and XML.

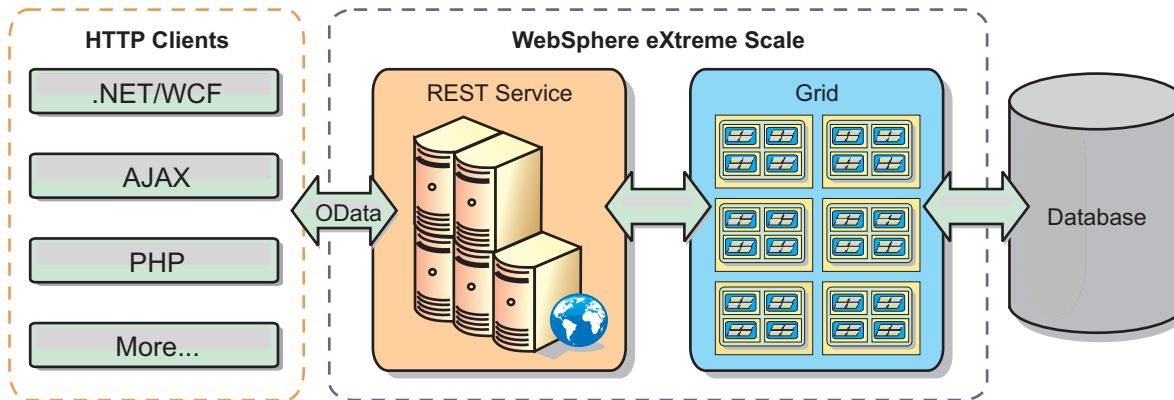


Figure 24. WebSphere eXtreme Scale REST data service

Refer to the [Configuring REST data services](#), or use the following links to learn more about WCF Data Services.

- [Microsoft WCF Data Services Developer Center](#)
- [ADO.NET Data Services overview on MSDN](#)
- [Whitepaper: Using ADO.NET Data Services](#)
- [Atom Publish Protocol: Data Services URI and Payload Extensions](#)
- [Conceptual Schema Definition File Format](#)
- [Entity Data Model for Data Services Packaging Format](#)
- [Open Data Protocol](#)
- [Open Data Protocol FAQ](#)

Features

This version of the eXtreme Scale REST data service supports the following features:

- Automatic modeling of eXtreme Scale EntityManager API entities as WCF Data Services entities, which includes the following support:
 - Java data type to Entity Data Model type conversion
 - Entity association support
 - Schema root and key association support, which is required for partitioned data grids

See [Entity model](#) for more information.

- Atom Publish Protocol (AtomPub or APP) XML and JavaScript Object Notation (JSON) data payload format.
- Create, Read, Update and Delete (CRUD) operations using the respective HTTP request methods: POST, GET, PUT and DELETE. In addition, the Microsoft extension: MERGE is supported.
- Simple queries, using filters
- Batch retrieval and change set requests
- Partitioned data grid support for high availability
- Interoperability with eXtreme Scale EntityManager API clients
- Support for standard JEE Web servers
- Optimistic concurrency
- User authorization and authentication between the REST data service and the eXtreme Scale data grid

Known problems and limitations

- Tunneled requests are not supported.

Spring framework overview

Spring is a framework for developing Java applications. WebSphere eXtreme Scale provides support to allow Spring to manage transactions and configure the clients and servers comprising your deployed in-memory data grid.

8.5+

Spring cache provider

Spring Framework Version 3.1 introduced a new cache abstraction. With this new abstraction, you can transparently add caching to an existing Spring application. You can use WebSphere eXtreme Scale as the cache provider for the cache abstraction. For more information, see [Configuring a Spring cache provider](#).

Spring managed native transactions

Spring provides container-managed transactions that are similar to a Java Platform, Enterprise Edition application server. However, the Spring mechanism can use different implementations. WebSphere eXtreme Scale provides transaction manager integration which allows Spring to manage the ObjectGrid transaction life cycles. For more information, see [“Managing transactions with Spring”](#) on page 381.

Spring managed extension beans and namespace support

Also, eXtreme Scale integrates with Spring to allow Spring-style beans defined for extension points or plug-ins. This feature provides more sophisticated configurations and more flexibility for configuring the extension points.

In addition to Spring managed extension beans, eXtreme Scale provides a Spring namespace called "objectgrid". Beans and built-in implementations are pre-defined in this namespace, which makes it easier for users to configure eXtreme Scale.

Shard scope support

With the traditional style Spring configuration, an ObjectGrid bean can either be a singleton type or prototype type. ObjectGrid also supports a new scope called the "shard" scope. If a bean is defined as shard scope, then only one bean is created per shard. All requests for beans with an ID or IDs matching that bean definition in the same shard results in that one specific bean instance being returned by the Spring container.

The following example shows that a `com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl` bean is defined with scope set to shard. Therefore, only one instance of the `JPAPropFactoryImpl` class is created per shard.

```
<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl" scope="shard" />
```

Spring Web Flow

Spring Web Flow stores its session state in an HTTP session by default. If a web application uses eXtreme Scale for session management, then Spring automatically stores state with eXtreme Scale. Also, fault tolerance is enabled in the same manner as the session.

See the HTTP session management information in the *Product Overview* for further details.

Packaging

The eXtreme Scale Spring extensions are in the `ogspring.jar` file. This Java archive (JAR) file must be on the class path for Spring support to work. If a Java EE application that is running in a WebSphere Extended Deployment augmented WebSphere Application Server Network Deployment, put the `spring.jar` file and its associated files in the enterprise archive (EAR) modules. You must also place the `ogspring.jar` file in the same location.

Class loader and classpath considerations

Since eXtreme Scale stores Java objects in the cache by default, you must define classes on the classpath wherever the data is accessed.

Specifically, eXtreme Scale client and container processes must include the classes or JAR files in the classpath when starting the process. When you are designing an application for use with eXtreme Scale, separate out any business logic from the persistent data objects.

See Class loading in the WebSphere Application Server information center for more information.

For considerations within a Spring Framework setting, see the packaging section under the topic on integrating with Spring framework in the *Programming Guide*.

Relationship management

Object-oriented languages such as Java, and relational databases support relationships or associations. Relationships decrease the amount of storage through the use of object references or foreign keys.

When you are using relationships in a data grid, the data must be organized in a constrained tree. One root type must exist in the tree and all children must be associated to only one root. For example: Department can have many Employees and an Employee can have many Projects. But a Project cannot have many Employees that belong to different departments. Once a root is defined, all access to that root object and its descendants are managed through the root. WebSphere eXtreme Scale uses the hash code of the root object's key to choose a partition. For example:

```
partition = (hashCode MOD numPartitions).
```

When all of the data for a relationship is tied to a single object instance, the entire tree can be collocated in a single partition and can be accessed very efficiently using one transaction. If the data spans multiple relationships, then multiple partitions must be involved which involves additional remote calls, which can lead to performance bottlenecks.

Reference data

Some relationships include look-up or reference data such as: CountryName. For look-up or reference data, the data must exist in every partition. The data can be accessed by any root key and the same result is returned. Reference data such as this should only be used in cases where the data is fairly static. Updating this data

can be expensive because the data needs to be updated in every partition. The DataGrid API is a common technique to keeping reference data up-to-date.

Costs and benefits of normalization

Normalizing the data using relationships can help reduce the amount of memory used by the data grid since duplication of data is decreased. However, in general, the more relational data that is added, the less it will scale out. When data is grouped together, it becomes more expensive to maintain the relationships and to keep the sizes manageable. Since the grid partitions data based on the key of the root of the tree, the size of the tree isn't taken into account. Therefore, if you have a lot of relationships for one tree instance, the data grid may become unbalanced, causing one partition to hold more data than the others.

When the data is denormalized or flattened, the data that would normally be shared between two objects is instead duplicated and each table can be partitioned independently, providing a much more balanced data grid. Although this increases the amount of memory used, it allows the application to scale since a single row of data can be accessed that has all of the necessary data. This is ideal for read-mostly grids since maintaining the data becomes more expensive.

For more information, see *Classifying XTP systems and scaling*.

Managing relationships using the data access APIs

The ObjectMap API is the fastest, most flexible and granular of the data access APIs, providing a transactional, session-based approach at accessing data in the grid of maps. The ObjectMap API allows clients to use common CRUD (create, read, update and delete) operations to manage key-value pairs of objects in the distributed data grid.

When using the ObjectMap API, object relationships must be expressed by embedding the foreign key for all relationships in the parent object.

An example follows.

```
public class Department {
    Collection<String> employeeIds;
}
```

The EntityManager API simplifies relationship management by extracting the persistent data from the objects including the foreign keys. When the object is later retrieved from the data grid, the relationship graph is rebuilt, as in the following example.

```
@Entity
public class Department {
    Collection<String> employees;
}
```

The EntityManager API is very similar to other Java object persistence technologies such as JPA and Hibernate in that it synchronizes a graph of managed Java object instances with the persistent store. In this case, the persistent store is an eXtreme Scale data grid, where each entity is represented as a map and the map contains the entity data rather than the object instances.

Cache key considerations

WebSphere eXtreme Scale uses hash maps to store data in the grid, where a Java object is used for the key.

Guidelines

When choosing a key, consider the following requirements:

- Keys can never change. If a portion of the key needs to change, then the cache entry should be removed and reinserted.
- Keys should be small. Since keys are used in every data access operation, it's a good idea to keep the key small so that it can be serialized efficiently and use less memory.
- Implement a good hash and equals algorithm. The `hashCode` and `equals(Object o)` methods must always be overridden for each key object.
- Cache the key's `hashCode`. If possible, cache the hash code in the key object instance to speed up `hashCode()` calculations. Since the key is immutable, the `hashCode` should be cacheable.
- Avoid duplicating the key in the value. When using the `ObjectMap` API, it is convenient to store the key inside the value object. When this is done, the key data is duplicated in memory.

Data for different time zones

When inserting data with `calendar`, `java.util.Date`, and `timestamp` attributes into an `ObjectGrid`, you must ensure these date time attributes are created based on same time zone, especially when deployed into multiple servers in various time zones. Using the same time zone based date time objects can ensure the application is time-zone safe and data can be queried by `calendar`, `java.util.Date` and `timestamp` predicates.

Without explicitly specifying a time zone when creating date time objects, Java uses the local time zone and may cause inconsistent date time values in clients and servers.

Consider an example in a distributed deployment in which `client1` is in time zone `[GMT-0]` and `client2` is in `[GMT-6]` and both want to create a `java.util.Date` object with value `'1999-12-31 06:00:00'`. Then `client1` will create `java.util.Date` object with value `'1999-12-31 06:00:00 [GMT-0]'` and `client2` will create `java.util.Date` object with value `'1999-12-31 06:00:00 [GMT-6]'`. Both `java.util.Date` objects are not equal because the time zone is different. A similar problem occurs when preloading data into partitions residing in servers in different time zones if local time zone is used to create date time objects.

To avoid the described problem, the application can choose a time zone such as `[GMT-0]` as the base time zone for creating `calendar`, `java.util.Date`, and `timestamp` objects.

Chapter 5. Developing applications



Develop applications that use the data grid. The tasks for developing applications include:

- Accessing data
- System APIs and plug-ins
- JPA integration
- Spring integration

Setting up the development environment

Before you begin developing applications, you must set up your development environment.

Before you begin

See “Planning to develop WebSphere eXtreme Scale applications” on page 116 for more information about the available programming interfaces and considerations.

Setting up a stand-alone development environment

Configure an Eclipse-based integrated development environment to build and run a Java SE application with the stand-alone version of WebSphere eXtreme Scale.

Before you begin

Install the WebSphere eXtreme Scale product into a new or empty directory and apply the latest WebSphere eXtreme Scale cumulative fix pack. You can also use the WebSphere eXtreme Scale trial version by unzipping the zip file. For more information on installation, see the information on installing the stand-alone WebSphere eXtreme Scale or WebSphere eXtreme Scale Client in the *Administration Guide*.

Procedure

- Configure Eclipse to build and run a Java SE application with WebSphere eXtreme Scale.
 1. Define a user library to allow your application to reference WebSphere eXtreme Scale application programming interfaces.
 - a. In your Eclipse or IBM® Rational Application Developer environment, click **Window > Preferences**.
 - b. Expand the **Java > Build Path** branch and select **User Libraries**. Click **New**.
 - c. Select the eXtreme Scale user library. Click **Add JARs**.
 - 1) Browse and select the `objectgrid.jar` or `ogclient.jar` files from the `wxs_root/lib` directory. Click **OK**. Select the `ogclient.jar` file if you are developing client applications or local, in-memory caches. If you are developing and testing eXtreme Scale servers, use the `objectgrid.jar` file.

- 2) To include Javadoc for the ObjectGrid APIs, select the Javadoc location for the `objectgrid.jar` or `ogclient.jar` file that you added in the previous step. Click **Edit**. In the Javadoc location path box, type the following web address:

`http://www.ibm.com/developerworks/wikis/extremescale/docs/api/`

- d. Click **OK** to apply the settings and close the Preferences window.

The eXtreme Scale libraries are now in the build path for the project.

2. Add the user library to your Java project.
 - a. From the package explorer, right-click the project and select **Properties**.
 - b. Select the **Libraries** tab.
 - c. Click **Add Library**.
 - d. Select **User Library**. Click **Next**.
 - e. Select the eXtreme Scale user library that you configured earlier.
 - f. Click **OK** to apply the changes and close the Properties window.

- Run a Java SE application with eXtreme Scale with Eclipse. Create a run configuration to execute your application.
 1. Configure Eclipse to build and run a Java SE application with eXtreme Scale. From the **Run** menu select **Run Configurations**.
 2. Right-click the Java Application category and select **New**.
 3. Select the new run configuration, named *New_Configuration*.
 4. Configure the profile.
 - **Project** (on main tabbed page): *your_project_name*
 - **Main Class** (on main tabbed page): *your_main_class*
 - **VM arguments** (on arguments tabbed page):
-Djava.endorsed.dirs=wxs_root/lib/endorsed

Problems with the **VM Arguments** often occur because the path to `java.endorsed.dirs` must be an absolute path with no variables or shortcuts.

Other common setup problems involve the Object Request Broker (ORB). You might see the following error. Refer to Configuring a custom Object Request Broker for more information:

```
Caused by: java.lang.RuntimeException: The ORB that comes  
with the Sun Java implementation does not work with  
ObjectGrid at this time.
```

If you do not have the `objectGrid.xml` or `deployment.xml` accessible to the application, you might see the following error:

```
Exception in thread "P=211046:0=0:CT" com.ibm.websphere.objectgrid.  
ObjectGridRuntimeException: Cannot start OG container at  
Client.startTestServer(Client.java:161) at Client.  
main(Client.java:82) Caused by: java.lang.IllegalArgumentException:  
The objectGridXML must not be null at com.ibm.websphere.objectgrid.  
deployment.DeploymentPolicyFactory.createDeploymentPolicy  
(DeploymentPolicyFactory.java:55) at Client.startTestServer(Client.  
java:154) .. 1 more
```

5. Click **Apply** and close the window, or click **Run**.

Running a WebSphere eXtreme Scale client or server application with Apache Tomcat in Rational Application Developer

Whether you have a client or server application, use the same basic steps to run the application in Apache Tomcat in Rational Application Developer. For a client

application, you want to configure and run a web application to use a WebSphere eXtreme Scale client in Rational Application Developer. Follow these instructions to create a web project for running a WebSphere eXtreme Scale catalog service or container. For a server application, you want to enable a Java EE application in Rational Application Developer interface with a stand-alone installation of WebSphere eXtreme Scale. Follow these instructions to configure a Java EE application project for using the WebSphere eXtreme Scale client library.

Before you begin

Install the WebSphere eXtreme Scale Trial or full product.

- Install the stand-alone version of the WebSphere eXtreme Scale product.
- Download and extract the WebSphere eXtreme Scale trial version.
- Install Apache Tomcat Version 6.0 or later.
- Install Rational Application Developer and create a Java EE web application.

Procedure

1. Add WebSphere eXtreme Scale runtime library to your Java EE build path.

Client application In this scenario, you want to configure and run a web application to use a WebSphere eXtreme Scale client in Rational Application Developer.

- a. **Window > Preferences > Java > Build Path > User Libraries.** Click **New**.
- b. Enter a **User library name** of `eXtremeScaleClient`, and click **OK**.
- c. Click **Add Jars...**, and navigate to and select the `wxs_home/lib/ogclient.jar` file. Click **Open**.
- d. Optional: (Optional) To add Javadoc, select Javadoc location and click **Edit...** In the Javadoc location path, you can either enter the URL of the API documentation, or you can download the API documentation.
 - To use the online API documentation, enter `http://www.ibm.com/developerworks/wikis/extremescale/docs/api/` in the Javadoc location path.
 - To download the API documentation, go to the WebSphere eXtreme Scale API documentation download page. For the Javadoc location path, enter your local download location.
- e. Click **OK**.
- f. Click **OK** to close out the User Libraries dialogue.
- g. Click **Project > Properties**.
- h. Click **Java Build Path**.
- i. Click **Add Library**.
- j. Select **User Library**. Click **Next**.
- k. Check the **eXtremeScaleClient** library and click **Finish**.
- l. Click **OK** to close the **Project Properties** dialog.

Server application In this scenario, you want to configure and run a web application to run an embedded WebSphere eXtreme Scale server in Rational Application Developer.

- a. Click **Window > Preferences > Java > Build Path > User Libraries**. Click **New**.
- b. Enter a **User library name** of `eXtremeScale`, and click **OK**.
- c. Click **Add Jars...**, and select `wxs_home/lib/objectgrid.jar`. Click **Open**.

- d. (Optional) To add Javadoc, select Javadoc location and click **Edit...** In the Javadoc location path, Enter `http://www.ibm.com/developerworks/wikis/extremescale/docs/api/`.
 - e. Click **OK**.
 - f. Click **OK** to close out the User Libraries dialogue.
 - g. Click **Project > Properties**.
 - h. Click **Java Build Path**.
 - i. Click **Add Library**.
 - j. Select **User Library**. Click **Next**.
 - k. Check the **eXtremeScaleClient** library and click **Finish**.
 - l. Click **OK** to close the **Project Properties** dialog.
2. Define Tomcat Server for our project.
 - a. Ensure that you are in the J2EE perspective and click the **Servers** tab in the bottom pane. You can also click **Window > Show View > Servers**.
 - b. Right-click in the Servers pane, and choose **New > Server**.
 - c. Choose **Apache, Tomcat v6.0 Server**. Click **Next**.
 - d. Click **Browse...** Select `tomcat_root`. Click **OK**.
 - e. Click **Next**.
 - f. Select your Java EE application in the left Available pane and click **Add >** to move it to the right Configured pane on the server, and click **Finish**.
 3. Resolve any remaining errors for the Project. Use the following steps to eliminate errors in the Problems pane:
 - a. Click **Project > Clean > *project_name***. Click **OK**. Build the project.
 - b. Right-click on the Java EE project, and choose **Build Path > Configure Build Path**.
 - c. Click the **Libraries** tab. Ensure that the path is configured properly:
 - **For client applications:** Ensure that Apache Tomcat, eXtremeScaleClient, and Java 1.5 JRE are on the path.
 - **For server applications:** Ensure that Apache Tomcat, eXtremeScale, and Java 1.5 JRE are on the path.
 4. Create a run configuration to run your application.
 - a. From the **Run** menu, select **Run Configurations**.
 - b. Right-click the Java Application category and select **New**.
 - c. Select the new run configuration, named *New_Configuration*.
 - d. Configure the profile.
 - **Project** (on main tabbed page): *your_project_name*
 - **Main Class** (on main tabbed page): *your_main_class*
 - **VM arguments** (on arguments tabbed page):
`-Djava.endorsed.dirs=wxs_root/lib/endorsed`

Problems with the **VM Arguments** often occur because the path to `java.endorsed.dirs` must be an absolute path with no variables or shortcuts.

Other common setup problems involve the Object Request Broker (ORB). You might see the following error. See *Configuring a custom Object Request Broker* for more information:

Caused by: `java.lang.RuntimeException: The ORB that comes with the Java implementation does not work with ObjectGrid at this time.`

If you do not have the `objectGrid.xml` or `deployment.xml` files accessible to the application, you might see the following error:

```
Exception in thread "P=211046:0=0:CT" com.ibm.websphere.objectgrid.ObjectGridRuntimeException:
Cannot start OG container
  at Client.startTestServer(Client.java:161)
  at Client.main(Client.java:82)
Caused by: java.lang.IllegalArgumentException: The objectGridXML must not be null
  at com.ibm.websphere.objectgrid.deployment.DeploymentPolicyFactory.createDeploymentPolicy
    (DeploymentPolicyFactory.java:55)
  at Client.startTestServer(Client.java:154)
... 1 more
```

5. Click **Apply** and close the window, or click **Run**.

What to do next

After you configure and run a web application with WebSphere eXtreme Scale client in Rational Application Developer, you can develop a servlet. This servlet uses the WebSphere eXtreme Scale APIs to store and retrieve data from a remote data grid.

After you enable a Java EE application in Rational Application Developer interface with a stand-alone installation of WebSphere eXtreme Scale, you can develop a servlet that uses the WebSphere eXtreme Scale system APIs to start and stop catalog services.

Running an integrated client or server application with WebSphere Application Server in Rational Application Developer

Configure and run a Java EE application with a WebSphere eXtreme Scale client or server with the WebSphere Application Server runtime embedded in Rational Application Developer. If you are configuring a server, starting WebSphere Application Server automatically starts WebSphere eXtreme Scale .

Before you begin

The following steps are for WebSphere Application Server Version 7.0 with Rational Application Developer Version 7.5. The following steps might vary if you are using different versions of these products.

Install Rational Application Developer with WebSphere Application Server Test Environment extensions.

Install WebSphere eXtreme Scale client or server into the WebSphere Application Server, Version 7.0 Test Environment in the `rad_home\runtimes\base_v7` directory. See [Installing WebSphere eXtreme Scale or WebSphere eXtreme Scale Client with WebSphere Application Server](#) for more information.

Procedure

1. Define eXtreme Scale server that is integrated with WebSphere Application Server for your project.
 - a. In the J2EE perspective, click **Window > Show View > Servers**.
 - b. Right-click in the **Servers** pane. Choose **New > Server**.
 - c. Choose **IBM WebSphere Application Server v7.0**. Click **Next**.
 - d. Select a profile to use. The default is `was70profile1`.
 - e. Enter the server name. The default is `server1`.

- f. Click **Next**.
 - g. Select your Java EE application in the **Available** pane. Click **Add >** to move it to the **Configured** pane on the server. Click **Finish**.
2. To run the Java EE application, start the application server. Right-click **WebSphere Application Server v7.0** and select **Start**.

Accessing data with client applications

After you configure your development environment, you can begin to develop applications that create, access, and manage the data in your data grid.

About this task

From the perspective of a client application, using WebSphere eXtreme Scale involves the following main steps:

- Connecting to the catalog service by obtaining a `ClientClusterContext` instance.
- Obtaining a client `ObjectGrid` instance.
- Getting a `Session` instance.
- Getting an `ObjectMap` instance.
- Using the `ObjectMap` methods.

Connecting to distributed ObjectGrid instances programmatically

You can connect to a distributed `ObjectGrid` with the connection end points for the catalog service domain. You must have the host name and listener port of each catalog server in the catalog service domain to which you want to connect.

Before you begin

- To connect to a distributed data grid, you must configure your server-side environment with a catalog service and container servers.
- You must have the listener port for each catalog service. For more information, see *Planning for network ports*.
- If the client application is running in WebSphere Application Server augmented with eXtreme Scale, configure the catalog service domain using the WebSphere Application Server administrative console or `wsadmin`.

About this task

When running in a Java EE application, consider using the eXtreme Scale resource adapter. The resource adapter allows the application to look up a `ObjectGrid` connection in Java Naming Directory Interface (JNDI) using a Java Connector Architecture (JCA) connection factory, which significantly simplifies access to the data grid and allows integration with Java Transaction API (JTA) transactions. For more information, see “Using JCA to connect transactional applications to eXtreme Scale clients” on page 58.

The `ObjectGridManager.connect()` methods connect to a catalog service domain using the supplied connection end points and returns a `ClientClusterContext` object that is used to retrieve `ObjectGrid` instances for the domain. The connection end points are a comma-delimited list of host and port combinations for each catalog server in the catalog service domain. See the following format of the catalog service endpoints:

```

catalogServiceEndpoints ::= <catalogServiceEndpoint> [,<catalogServiceEndpoint>]
catalogServiceEndpoint ::= <hostName> : <listenerPort>
hostName                 ::= The IP address or host name of a catalog service.
listenerPort             ::= The listener port that the catalog service is configured to use.

```

After you connect to the catalog service domain, use the `ObjectGridManagerFactory.getObjectGrid(ClientClusterContext ccc, String objectGridName)` method to retrieve a named `ObjectGrid` client instance. This `ObjectGrid` instance is a proxy for the named data grid and is cached in the client application. The `ObjectGrid` instance represents a logical connection to the remote data grid and is thread safe. All underlying physical connections to the data grid are managed automatically and can tolerate failure events.

The connection steps vary depending on whether you are using a stand-alone configuration or WebSphere Application Server.

Procedure

- Connect to a stand-alone distributed data grid using explicit catalog service endpoints.

```

// Retrieve an ObjectGridManager instance.
ObjectGridManager ogm = ObjectGridManagerFactory.getObjectGridManager();

// Obtain a ClientClusterContext by connecting to a catalog
// service domain, manually supplying the catalog service endpoints,
// and optionally specifying the ClientSecurityConfiguration and
// client ObjectGrid override XML file URL.
String catalogServiceEndpoints = "host1:2809,host2:2809";
ClientClusterContext ccc = ogm.connect(catalogServiceEndpoints,
    (ClientSecurityConfiguration) null, (URL) null);

// Obtain a distributed ObjectGrid using ObjectGridManager and providing
// the ClientClusterContext.
ObjectGrid og = ogm.getObjectGrid(ccc, "Mygrid");

```

- Connect to a catalog service domain from a client application that is hosted in WebSphere Application Server, where the catalog service domain was configured using the administrative console or admin task. The catalog service endpoints can be retrieved from a named domain identifier or for the default domain using the `ObjectGridManager`.

```

// Retrieve an ObjectGridManager instance.
ObjectGridManager ogm = ObjectGridManagerFactory.getObjectGridManager();

// Retrieve the domain by its ID (the name given to it in the admin console or wsadmin)
// The CatalogDomainManager also includes methods to retrieve all domains and the default domain.
CatalogDomainInfo di = ogm.getCatalogDomainManager().getDomainInfo("ProductionDomain");
if(di == null) throw new IllegalStateException("Domain not configured");

// Connect to the domain using the catalog service endpoints and the security configuration
// in the CatalogDomainInfo object. The client override ObjectGrid XML is optional
// and is manually supplied.
ClientClusterContext ccc = ogm.connect(di.getClientCatalogServiceAddresses(),
    di.getClientSecurityConfiguration(), (URL) null);

// Obtain a distributed ObjectGrid using ObjectGridManager and by providing
// the ClientClusterContext.
ObjectGrid og = ogm.getObjectGrid(ccc, "MyGrid");

```

What to do next

If the catalog service domain is hosted in a WebSphere Application Server deployment manager, clients outside of the cell, including Java Platform, Enterprise Edition clients, must connect to the catalog service using the deployment manager host name and the IIOP bootstrap port. When the catalog service runs in WebSphere Application Server cells, and the clients run outside of the cells, look to the eXtreme Scale domain configuration pages in the WebSphere Application

Server administrative console for the information that you need to point a client to the catalog service.

Tracking map updates by an application

When an application is making changes to a Map during a transaction, a LogSequence object tracks those changes. If the application changes an entry in the map, a corresponding LogElement object provides the details of the change.

Loaders are given a LogSequence object for a particular map whenever an application calls for a flush or commit to the transaction. The Loader iterates over the LogElement objects within the LogSequence object and applies each LogElement object to the backend.

ObjectGridEventListener listeners that are registered with an ObjectGrid also use LogSequence objects. These listeners are given a LogSequence object for each map in a committed transaction. Applications can use these listeners to wait for certain entries to change, like a trigger in a conventional database.

The following log-related interfaces or classes are provided by the eXtreme Scale framework:

- `com.ibm.websphere.objectgrid.plugins.LogElement`
- `com.ibm.websphere.objectgrid.plugins.LogSequence`
- `com.ibm.websphere.objectgrid.plugins.LogSequenceFilter`
- `com.ibm.websphere.objectgrid.plugins.LogSequenceTransformer`

LogElement interface

A LogElement represents an operation on an entry during a transaction. A LogElement object has several methods to get its various attributes. The most commonly used attributes are the type and the current value attributes fetched by `getType()` and `getCurrentValue()`.

The type is represented by one of the constants defined in the LogElement interface: `INSERT`, `UPDATE`, `DELETE`, `EVICT`, `FETCH`, or `TOUCH`.

The current value represents the new value for the operation if it is `INSERT`, `UPDATE` or `FETCH`. If the operation is `TOUCH`, `DELETE`, or `EVICT`, then the current value is null. This value can be cast to `ValueProxyInfo` when a `ValueInterface` is in use.

See the API documentation for more details on the LogElement interface.

LogSequence interface

In most transactions, operations to more than one entry in a map occur, so multiple LogElement objects are created. You should create an object that behaves as a composite of multiple LogElement objects. The LogSequence interface serves this purpose by containing a list of LogElement objects.

See the API documentation for more details on the LogSequence interface.

Using LogElement and LogSequence

LogElement and LogSequence are widely used in eXtreme Scale and by ObjectGrid plug-ins that are written by users when operations are propagated from one component or server to another component or server. For example, a LogSequence object can be used by the distributed ObjectGrid transaction propagation function to propagate the changes to other servers, or it can be applied to the persistence store by the loader. LogSequence is mainly used by the following interfaces.

- com.ibm.websphere.objectgrid.plugins.ObjectGridEventListener
- com.ibm.websphere.objectgrid.plugins.Loader
- com.ibm.websphere.objectgrid.plugins.Evictor
- com.ibm.websphere.objectgrid.Session

Loader example

This section demonstrates how the LogSequence and LogElement objects are used in a Loader. A Loader is used to load data from and persist data into a persistent store. The batchUpdate method of the Loader interface uses LogSequence object:

```
void batchUpdate(TxID txid, LogSequence sequence) throws  
    LoaderException, OptimisticCollisionException;
```

The batchUpdate method is called when an ObjectGrid needs to apply all current changes to the Loader. The Loader is given a list of LogElement objects for the map, encapsulated in a LogSequence object. The implementation of the batchUpdate method must iterate over the changes and apply them to the backend. The following code snippet demonstrates how a Loader uses a LogSequence object. The snippet iterates over the set of changes and builds up three batch Java database connectivity (JDBC) statements: inserts, updates, and deletes:

```
public void batchUpdate(TxID tx, LogSequence sequence) throws LoaderException  
{  
    // Get a SQL connection to use.  
    Connection conn = getConnection(tx);  
    try  
    {  
        // Process the list of changes and build a set of prepared  
        // statements for executing a batch update, insert, or delete  
        // SQL operations. The statements are cached in stmtCache.  
        Iterator iter = sequence.getPendingChanges();  
        while ( iter.hasNext() )  
        {  
            LogElement logElement = (LogElement)iter.next();  
            Object key = logElement.getCacheEntry().getKey();  
            Object value = logElement.getCurrentValue();  
            switch ( logElement.getType().getCode() )  
            {  
                case LogElement.CODE_INSERT:  
                    buildBatchSQLInsert( key, value, conn );  
                    break;  
                case LogElement.CODE_UPDATE:  
                    buildBatchSQLUpdate( key, value, conn );  
                    break;  
                case LogElement.CODE_DELETE:  
                    buildBatchSQLDelete( key, conn );  
                    break;  
            }  
        }  
        // Run the batch statements that were built by above loop.  
        Collection statements = getPreparedStatementCollection( tx, conn );  
        iter = statements.iterator();  
    }  
}
```

```

        while ( iter.hasNext() )
        {
            PreparedStatement pstmt = (PreparedStatement) iter.next();
            pstmt.executeBatch();
        }
    } catch (SQLException e)
    {
        LoaderException ex = new LoaderException(e);
        throw ex;
    }
}

```

The previous sample illustrates the high-level logic of processing the `LogSequence` argument. However, the sample does not illustrate the details of how an SQL insert, update, or delete statement is built. The `getPendingChanges` method is called on the `LogSequence` argument to obtain an iterator of `LogElement` objects that a Loader needs to process, and the `LogElement.getType().getCode()` method is used to determine whether a `LogElement` is for an SQL insert, update, or delete operation.

Evictor sample

You can also use `LogSequence` and `LogElement` objects with an Evictor. An Evictor is used to evict the map entries from the backing map based on certain criteria. The `apply` method of the Evictor interface uses `LogSequence`.

```

/**
 * This is called during cache commit to allow the evictor to track object usage
 * in a backing map. This will also report any entries that have been successfully
 * evicted.
 *
 * @param sequence LogSequence of changes to the map
 */
void apply(LogSequence sequence);

```

LogSequenceFilter and LogSequenceTransformer interfaces

Sometimes, it is necessary to filter the `LogElement` objects so that only `LogElement` objects with certain criteria are accepted, and reject other objects. For example, you might want to serialize a certain `LogElement` based on some criterion.

`LogSequenceFilter` solves this problem with the following method.

```
public boolean accept (LogElement logElement);
```

This method returns true if the given `LogElement` should be used in the operation, and returns false if the given `LogElement` should not be used.

`LogSequenceTransformer` is a class that uses the `LogSequenceFilter` function. It uses the `LogSequenceFilter` to filter out some `LogElement` objects and then serialize the accepted `LogElement` objects. This class has two methods. The first method follows.

```
public static void serialize(Collection logSequences, ObjectOutputStream stream,
    LogSequenceFilter filter, DistributionMode mode) throws IOException
```

This method allows the caller to provide a filter for determining which `LogElements` to include in the serialization process. The `DistributionMode` parameter allows the caller to control the serialization process. For example, if the distribution mode is invalidation only, then there is no need to serialize the value. The second method of this class is the `inflate` method, as follows.


```
public static Collection inflate(ObjectInputStream stream, ObjectGrid
    objectGrid) throws IOException, ClassNotFoundException
```

The inflate method reads the log sequence serialized form, which was created by the serialize method, from the provided object input stream.

Interacting with an ObjectGrid using the ObjectGridManager interface

The ObjectGridManagerFactory class and the ObjectGridManager interface provide a mechanism to create, access, and add data to ObjectGrid instances. The ObjectGridManagerFactory class is a static helper class to access the ObjectGridManager interface, a singleton. The ObjectGridManager interface includes several convenience methods to create instances of an ObjectGrid object. The ObjectGridManager interface also facilitates creation and caching of ObjectGrid instances that can be accessed by several users.

Creating ObjectGrid instances with the ObjectGridManager interface

Each of these methods creates a local instance of an ObjectGrid.

Local in-memory instance

The following code snippet illustrates how to obtain and configure a local ObjectGrid instance with eXtreme Scale.

```
// Obtain a local ObjectGrid reference
// you can create a new ObjectGrid, or get configured ObjectGrid
// defined in ObjectGrid xml file
ObjectGridManager objectGridManager =
ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid ivObjectGrid =
objectGridManager.createObjectGrid("objectgridName");

// Add a TransactionCallback into ObjectGrid
HeapTransactionCallback tcb = new HeapTransactionCallback();
ivObjectGrid.setTransactionCallback(tcb);

// Define a BackingMap
// if the BackingMap is configured in ObjectGrid xml
// file, you can just get it.
BackingMap ivBackingMap = ivObjectGrid.defineMap("myMap");

// Add a Loader into BackingMap
Loader ivLoader = new HeapCacheLoader();
ivBackingMap.setLoader(ivLoader);

// initialize ObjectGrid
ivObjectGrid.initialize();

// Obtain a session to be used by the current thread.
// Session can not be shared by multiple threads
Session ivSession = ivObjectGrid.getSession();

// Obtaining ObjectMap from ObjectGrid Session
ObjectMap objectMap = ivSession.getMap("myMap");
```

Default shared configuration

The following code is a simple case of creating an ObjectGrid to share among many users.

```

import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
final ObjectGridManager oGridManager=
    ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid employees =
    oGridManager.createObjectGrid("Employees",true);
employees.initialize();
employees.
/*sample continues..*/

```

The preceding Java code snippet creates and caches the Employees ObjectGrid. The Employees ObjectGrid is initialized with the default configuration and is ready for use. The second parameter in the createObjectGrid method is set to true, which instructs the ObjectGridManager to cache the ObjectGrid instance it creates. If this parameter is set to false, the instance is not cached. Every ObjectGrid instance has a name, and the instance can be shared among many clients or users based on that name.

If the objectGrid instance is used in peer-to-peer sharing, the caching must be set to true. For more information on peer-to-peer sharing, see [Distributing changes between peer Java Virtual Machines](#).

XML configuration

WebSphere eXtreme Scale is highly configurable. The previous example demonstrates how to create a simple ObjectGrid without any configuration. This example shows you how to create a pre-configured ObjectGrid instance that is based on an XML configuration file. You can configure an ObjectGrid instance programmatically or using an XML-based configuration file. You can also configure ObjectGrid using a combination of both approaches. The ObjectGridManager interface allows creation of an ObjectGrid instance based on the XML configuration. The ObjectGridManager interface has several methods that take a URL as an argument. Every XML file that is passed into the ObjectGridManager must be validated against the schema. XML validation can be disabled only when the file is previously validated and no changes have been made to the file since its last validation. Disabling validation saves a small amount of overhead but introduces the possibility of using an invalid XML file. The IBM Java Developer Kit (JDK) Version 5 has support for XML validation. When using a JDK that does not have this support, Apache Xerces might be required to validate the XML.

The following Java code snippet demonstrates how to pass in an XML configuration file to create an ObjectGrid.

```

import java.net.MalformedURLException;
import java.net.URL;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
boolean validateXML = true; // turn XML validation on
boolean cacheInstance = true; // Cache the instance
String objectGridName="Employees"; // Name of Object Grid
URL allObjectGrids = new URL("file:test/myObjectGrid.xml");
final ObjectGridManager oGridManager=
    ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid employees =
    oGridManager.createObjectGrid(objectGridName, allObjectGrids,
        bvalidateXML, cacheInstance);

```

The XML file can contain configuration information for several ObjectGrids. The previous code snippet specifically returns ObjectGrid Employees, assuming that the Employees configuration is defined in the file.

createObjectGrid methods

```
.  
  
/**  
 * A simple factory method to return an instance of an  
 * Object Grid. A unique name is assigned.  
 * The instance of ObjectGrid is not cached.  
 * Users can then use {@link ObjectGrid#setName(String)} to change the  
 * ObjectGrid name.  
 *  
 * @return ObjectGrid an instance of ObjectGrid with a unique name assigned  
 * @throws ObjectGridException any error encountered during the  
 * ObjectGrid creation  
 */  
public ObjectGrid createObjectGrid() throws ObjectGridException;  
  
/**  
 * A simple factory method to return an instance of an ObjectGrid with the  
 * specified name. The instances of ObjectGrid can be cached. If an ObjectGrid  
 * with the this name has already been cached, an ObjectGridException  
 * will be thrown.  
 *  
 * @param objectGridName the name of the ObjectGrid to be created.  
 * @param cacheInstance true, if the ObjectGrid instance should be cached  
 * @return an ObjectGrid instance  
 * @this name has already been cached or  
 * any error during the ObjectGrid creation.  
 */  
public ObjectGrid createObjectGrid(String objectGridName, boolean cacheInstance)  
    throws ObjectGridException;  
  
/**  
 * Create an ObjectGrid instance with the specified ObjectGrid name. The  
 * ObjectGrid instance created will be cached.  
 * @param objectGridName the Name of the ObjectGrid instance to be created.  
 * @return an ObjectGrid instance  
 * @throws ObjectGridException if an ObjectGrid with this name has already  
 * been cached, or any error encountered during the ObjectGrid creation  
 */  
public ObjectGrid createObjectGrid(String objectGridName)  
    throws ObjectGridException;  
  
/**  
 * Create an ObjectGrid instance based on the specified ObjectGrid name and the  
 * XML file. The ObjectGrid instance defined in the XML file with the specified  
 * ObjectGrid name will be created and returned. If such an ObjectGrid  
 * cannot be found in the xml file, an exception will be thrown.  
 *  
 * This ObjecGrid instance can be cached.  
 *  
 * If the URL is null, it will be simply ignored. In this case, this method behaves  
 * the same as {@link #createObjectGrid(String, boolean)}.  
 *  
 * @param objectGridName the Name of the ObjectGrid instance to be returned. It  
 * must not be null.  
 * @param xmlFile a URL to a wellformed xml file based on the ObjectGrid schema.  
 * @param enableXmlValidation if true the XML is validated  
 * @param cacheInstance a boolean value indicating whether the ObjectGrid  
 * instance(s)  
 * defined in the XML will be cached or not. If true, the instance(s) will  
 * be cached.
```

```

*
* @throws ObjectGridException if an ObjectGrid with the same name
* has been previously cached, no ObjectGrid name can be found in the xml file,
* or any other error during the ObjectGrid creation.
* @return an ObjectGrid instance
* @see ObjectGrid
*/
public ObjectGrid createObjectGrid(String objectGridName, final URL xmlFile,
final boolean enableXmlValidation, boolean cacheInstance)
throws ObjectGridException;

/**
* Process an XML file and create a List of ObjectGrid objects based
* upon the file.
* These ObjectGrid instances can be cached.
* An ObjectGridException will be thrown when attempting to cache a
* newly created ObjectGrid
* that has the same name as an ObjectGrid that has already been cached.
*
* @param xmlFile the file that defines an ObjectGrid or multiple
* ObjectGrids
* @param enableXmlValidation setting to true will validate the XML
* file against the schema
* @param cacheInstances set to true to cache all ObjectGrid instances
* created based on the file
* @return an ObjectGrid instance
* @throws ObjectGridException if attempting to create and cache an
* ObjectGrid with the same name as
* an ObjectGrid that has already been cached, or any other error
* occurred during the
* ObjectGrid creation
*/
public List createObjectGrids(final URL xmlFile, final boolean enableXmlValidation,
boolean cacheInstances) throws ObjectGridException;

/** Create all ObjectGrids that are found in the XML file. The XML file will be
* validated against the schema. Each ObjectGrid instance that is created will
* be cached. An ObjectGridException will be thrown when attempting to cache a
* newly created ObjectGrid that has the same name as an ObjectGrid that has
* already been cached.
* @param xmlFile The XML file to process. ObjectGrids will be created based
* on what is in the file.
* @return A List of ObjectGrid instances that have been created.
* @throws ObjectGridException if an ObjectGrid with the same name as any of
* those found in the XML has already been cached, or
* any other error encountered during ObjectGrid creation.
*/
public List createObjectGrids(final URL xmlFile) throws ObjectGridException;

/**
* Process the XML file and create a single ObjectGrid instance with the
* objectGridName specified only if an ObjectGrid with that name is found in
* the file. If there is no ObjectGrid with this name defined in the XML file,
* an ObjectGridException
* will be thrown. The ObjectGrid instance created will be cached.
* @param objectGridName name of the ObjectGrid to create. This ObjectGrid
* should be defined in the XML file.
* @param xmlFile the XML file to process
* @return A newly created ObjectGrid
* @throws ObjectGridException if an ObjectGrid with the same name has been
* previously cached, no ObjectGrid name can be found in the xml file,
* or any other error during the ObjectGrid creation.
*/
public ObjectGrid createObjectGrid(String objectGridName, URL xmlFile)
throws ObjectGridException;

```

Retrieving a ObjectGrid instance with the ObjectGridManager interface

Use the ObjectGridManager.getObjectGrid methods to retrieve cached instances.

Retrieving a cached instance

Since the Employees ObjectGrid instance was cached by the ObjectGridManager interface, another user can access it with the following code snippet:

```
ObjectGrid myEmployees = oGridManager.getObjectGrid("Employees");
```

The following are the two getObjectGrid methods that return cached ObjectGrid instances:

- **Retrieving all cached instances**

To obtain all of the ObjectGrid instances that have been previously cached, use the getObjectGrids method, which returns a list of each instance. If no cached instances exist, the method will return null.

- **Retrieving a cached instance by name**

To obtain a single cached instance of an ObjectGrid, use getObjectGrid(String objectGridName), passing the name of the cached instance into the method. The method either returns the ObjectGrid instance with the specified name or returns null if there is no ObjectGrid instance with that name.

Note: You can also use the getObjectGrid method to connect to a distributed grid. See “Connecting to distributed ObjectGrid instances programmatically” on page 130 for more information.

Removing ObjectGrid instances with the ObjectGridManager interface

You can use two different removeObjectGrid methods to remove ObjectGrid instances from the cache.

Remove an ObjectGrid instance

To remove ObjectGrid instances from the cache, use one of the removeObjectGrid methods. The ObjectGridManager interface does not keep a reference of the instances that are removed. Two remove methods exist. One method takes a boolean parameter. If the boolean parameter is set to true, the destroy method is called on the ObjectGrid. The call to the destroy method on the ObjectGrid shuts down the ObjectGrid and frees up any resources the ObjectGrid is using. A description of how to use the two removeObjectGrid methods follows:

```
/**
 * Remove an ObjectGrid from the cache of ObjectGrid instances
 *
 * @param objectGridName the name of the ObjectGrid instance to remove
 * from the cache
 *
 * @throws ObjectGridException if an ObjectGrid with the objectGridName
 * was not found in the cache
 */
public void removeObjectGrid(String objectGridName) throws ObjectGridException;

/**
 * Remove an ObjectGrid from the cache of ObjectGrid instances and
 * destroy its associated resources
 *
 * @param objectGridName the name of the ObjectGrid instance to remove
 * from the cache
```

```

*
* @param destroy destroy the objectgrid instance and its associated
* resources
*
* @throws ObjectGridException if an ObjectGrid with the objectGridName
* was not found in the cache
*/
public void removeObjectGrid(String objectGridName, boolean destroy)
    throws ObjectGridException;

```

Controlling the lifecycle of an ObjectGrid with the ObjectGridManager interface

You can use the ObjectGridManager interface to control the lifecycle of an ObjectGrid instance using either a startup bean or a servlet.

Managing lifecycle with a startup bean

A startup bean is used to control the lifecycle of an ObjectGrid instance. A startup bean loads when an application starts. With a startup bean, code can run whenever an application starts or stops as expected. To create a startup bean, use the `com.ibm.websphere.startupservice.AppStartUpHome` interface and use the `remote.com.ibm.websphere.startupservice.AppStartUp` interface. Implement the `start` and `stop` methods on the bean. The `start` method is invoked whenever the application starts up. The `stop` method is invoked when the application shuts down. The `start` method is used to create ObjectGrid instances. The `stop` method is used to remove ObjectGrid instances. A code snippet that demonstrates this ObjectGrid lifecycle management in a startup bean follows:

```

public class MyStartupBean implements javax.ejb.SessionBean {
    private ObjectGridManager objectGridManager;

    /* The methods on the SessionBean interface have been
    * left out of this example for the sake of brevity */

    public boolean start(){
        // Starting the startup bean
        // This method is called when the application starts
        objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
        try {
            // create 2 ObjectGrids and cache these instances
            ObjectGrid bookstoreGrid = objectGridManager.createObjectGrid("bookstore", true);
            bookstoreGrid.defineMap("book");
            ObjectGrid videostoreGrid = objectGridManager.createObjectGrid("videostore", true);
            // within the JVM,
            // these ObjectGrids can now be retrieved from the
            //ObjectGridManager using the getObjectGrid(String) method
        } catch (ObjectGridException e) {
            e.printStackTrace();
            return false;
        }

        return true;
    }

    public void stop(){
        // Stopping the startup bean
        // This method is called when the application is stopped
        try {
            // remove the cached ObjectGrids and destroy them
            objectGridManager.removeObjectGrid("bookstore", true);
            objectGridManager.removeObjectGrid("videostore", true);
        } catch (ObjectGridException e) {
            e.printStackTrace();
        }
    }
}

```

After the `start` method is called, the newly created ObjectGrid instances are retrieved from the ObjectGridManager interface. For example, if a servlet is included in the application, the servlet accesses the eXtreme Scale using the following code snippet:

```

ObjectGridManager objectGridManager =
    ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid bookstoreGrid = objectGridManager.getObjectGrid("bookstore");
ObjectGrid videostoreGrid = objectGridManager.getObjectGrid("videostore");

```

Managing lifecycle with a servlet

To manage the lifecycle of an ObjectGrid in a servlet, you can use the `init` method to create an ObjectGrid instance and the `destroy` method to remove the ObjectGrid instance. If the ObjectGrid instance is cached, it is retrieved and manipulated in the servlet code. Sample code that demonstrates ObjectGrid creation, manipulation, and destruction within a servlet follows:

```

public class MyObjectGridServlet extends HttpServlet implements Servlet {
    private ObjectGridManager objectGridManager;

    public MyObjectGridServlet() {
        super();
    }

    public void init(ServletConfig arg0) throws ServletException {
        super.init();
        objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
        try {
            // create and cache an ObjectGrid named bookstore
            ObjectGrid bookstoreGrid =
                objectGridManager.createObjectGrid("bookstore", true);
            bookstoreGrid.defineMap("book");
        } catch (ObjectGridException e) {
            e.printStackTrace();
        }
    }

    protected void doGet(HttpServletRequestRequest req, HttpServletResponse res)
        throws ServletException, IOException {
        ObjectGrid bookstoreGrid = objectGridManager.getObjectGrid("bookstore");
        Session session = bookstoreGrid.getSession();
        ObjectMap bookMap = session.getMap("book");
        // perform operations on the cached ObjectGrid
        // ...
        // Close the session (optional in Version 7.1.1 and later) for improved performance
        session.close();
    }

    public void destroy() {
        super.destroy();
        try {
            // remove and destroy the cached bookstore ObjectGrid
            objectGridManager.removeObjectGrid("bookstore", true);
        } catch (ObjectGridException e) {
            e.printStackTrace();
        }
    }
}

```

Accessing the ObjectGrid shard

WebSphere eXtreme Scale achieves high processing rates by moving the logic to where the data is and returning only results back to the client.

Application logic in a client Java virtual machine (JVM) needs to pull data from the server JVM that is holding the data and push it back when the transaction commits. This process slows down the rate the data can be processed. If the application logic was on the same JVM as the shard that is holding the data, then the network latency and marshalling cost is eliminated and can provide a significant performance boost.

Local reference to shard data

The ObjectGrid APIs provide a Session to the server-side method. This session is a direct reference to the data for that shard. No routing logic is on that path. The application logic can work with the data for that shard directly. The session cannot be used to access data in another partition because no routing logic exists.

A Loader plug-in also provides a way to receive an event when a shard becomes a primary partition. An application can implement a Loader and implement the ReplicaPreloadController interface. The check preload status method is only called when a shard becomes a primary. The session provided to that method is a local reference to the shards data. This approach is typically used if a partition primary needs to start some threads or subscribe to a message fabric for partition-related traffic. It might start a thread to listen for messages in a local Map using the getNextKey API.

Collocated client-server optimization

If an application uses the client APIs to access a partition that happens to be collocated with the JVM that contains the client, then the network is avoided but some marshalling still occurs because of current implementation issues. If a partitioned grid is used, then no impact on the performance of the application is made because (N-1)/N number of calls route to a different JVM. If you need local access always with a shard, then use the Loader or ObjectGrid APIs to invoke that logic.

Accessing data with indexes (Index API)

Use indexing for more efficient data access.

About this task

The HashIndex class is the built-in index plug-in implementation that can support both of the built-in application index interfaces: MapIndex and MapRangeIndex. You also can create your own indexes. You can add HashIndex as either a static or dynamic index into the backing map, obtain either MapIndex or MapRangeIndex index proxy object, and use the index proxy object to find cached objects.

If you want to iterate through the keys in a local map, you can use the default index. This index does not require any configuration, but it must be used against the shard, using an agent or an ObjectGrid instance retrieved from the ShardEvents.shardActivated(ObjectGrid shard) method.

Note: In a distributed environment, if the index object is obtained from a client ObjectGrid, the index has a type client index object and all index operations run in a remote server ObjectGrid. If the map is partitioned, the index operations run on each partition remotely. The results from each partition are merged before returning the results to the application. The performance is determined by the number of partitions and the size of the result returned by each partition. Poor performance might occur if both factors are high.

Procedure

1. If you want to use indexes other than the default local index, add index plug-ins to the backing map.
 - **XML configuration:**


```

<backingMapPluginCollection id="person">
  <bean id="MapIndexplugin"
    className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
    <property name="Name" type="java.lang.String" value="CODE"
      description="index name" />
    <property name="RangeIndex" type="boolean" value="true"
      description="true for MapRangeIndex" />
    <property name="AttributeName" type="java.lang.String" value="employeeCode"
      description="attribute name" />
    </bean>
</backingMapPluginCollection>

```

In this XML configuration example, the built-in HashIndex class is used as the index plug-in. The HashIndex class supports properties that users can configure, such as Name, RangeIndex, and AttributeName in the previous example.

- The **Name** property is configured as CODE, a string identifying this index plug-in. The Name property value must be unique within the scope of the BackingMap, and can be used to retrieve the index object by name from the ObjectMap instance for the BackingMap.
 - The **RangeIndex** property is configured as true, which means the application can cast the retrieved index object to the MapRangeIndex interface. If the RangeIndex property is configured as false, the application can only cast the retrieved index object to the MapIndex interface. A MapRangeIndex supports functions to find data using range functions such as greater than, less than, or both, while a MapIndex only supports equals functions. If the index is used by query, the **RangeIndex** property must be configured to true on single-attribute indexes. For a relationship index and composite index, the RangeIndex property must be configured to false.
 - The **AttributeName** property is configured as employeeCode, which means the **employeeCode** attribute of the cached object is used to build a single-attribute index. If an application needs to search for cached objects with multiple attributes, the **AttributeName** property can be set to a comma-delimited list of attributes, yielding a composite index.
- **Programmatic configuration:**

The BackingMap interface has two methods that you can use to add static index plug-ins: addMapIndexplugin and setMapIndexplugins. For more information, see the API documentation. The following example creates the same configuration as the XML configuration example:

```

import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;

ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid ivObjectGrid = ogManager.createObjectGrid( "grid" );
BackingMap personBackingMap = ivObjectGrid.getMap("person");

// use the builtin HashIndex class as the index plugin class.
HashIndex mapIndexplugin = new HashIndex();
mapIndexplugin.setName("CODE");
mapIndexplugin.setAttributeName("EmployeeCode");
mapIndexplugin.setRangeIndex(true);
personBackingMap.addMapIndexplugin(mapIndexplugin);

```

2. Access map keys and values with indexes.

- **Local index:**

To iterate through the keys and values in a local map, you can use the default index. The default index only works against the shard, using an agent or using the ObjectGrid instance retrieved from the ShardEvents.shardActivated(ObjectGrid shard) method. See the following example:

```

MapIndex keyIndex = (MapIndex)
objMap.getIndex(MapIndexPlugin.SYSTEM_KEY_INDEX_NAME);
Iterator keyIterator = keyIndex.findAll();

```

- **Static indexes:**

After a static index plug-in is added to a BackingMap configuration and the containing ObjectGrid instance is initialized, applications can retrieve the index object by name from the ObjectMap instance for the BackingMap. Cast the index object to the application index interface. Operations that the application index interface supports can now run.

```
Session session = ivObjectGrid.getSession();
ObjectMap map = session.getMap("person ");
MapRangeIndex codeIndex = (MapRangeIndex) m.getIndex("CODE");
Iterator iter = codeIndex.findLessEqual(new Integer(15));
while (iter.hasNext()) {
    Object key = iter.next();
    Object value = map.get(key);
}
// Close the session (optional in Version 7.1.1 and later) for improved performance
session.close();
```

- **Dynamic indexes:**

You can create and remove dynamic indexes from a BackingMap instance programmatically at any time. A dynamic index differs from a static index in that the dynamic index can be created even after the containing ObjectGrid instance is initialized. Unlike static indexing, the dynamic indexing is an asynchronous process and needs to be in ready state before you use it. This method uses the same approach for retrieving and using the dynamic indexes as static indexes. You can remove a dynamic index if it is no longer needed. The BackingMap interface has methods to create and remove dynamic indexes.

See the BackingMap API documentation for more information about the createDynamicIndex and removeDynamicIndex methods.

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;

ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid("grid");
BackingMap bm = og.getMap("person");
og.initialize();

// create index after ObjectGrid initialization without DynamicIndexCallback.
bm.createDynamicIndex("CODE", true, "employeeCode", null);

try {
    // If not using DynamicIndexCallback, need to wait for the Index to be ready.
    // The waiting time depends on the current size of the map
    Thread.sleep(3000);
} catch (Throwable t) {
    // ...
}

// When the index is ready, applications can try to get application index
// interface instance.
// Applications have to find a way to ensure that the index is ready to use,
// if not using DynamicIndexCallback interface.
// The following example demonstrates the way to wait for the index to be ready
// Consider the size of the map in the total waiting time.

Session session = og.getSession();
ObjectMap m = session.getMap("person");
MapRangeIndex codeIndex = null;

int counter = 0;
int maxCounter = 10;
boolean ready = false;
while (!ready && counter < maxCounter) {
    try {
        counter++;
        codeIndex = (MapRangeIndex) m.getIndex("CODE");
        ready = true;
    } catch (IndexNotReadyException e) {
        // implies index is not ready, ...
        System.out.println("Index is not ready. continue to wait.");
        try {
            Thread.sleep(3000);
        } catch (Throwable tt) {
            // ...
        }
    } catch (Throwable t) {
        // unexpected exception
        t.printStackTrace();
    }
}

if (!ready) {
    System.out.println("Index is not ready. Need to handle this situation.");
}
```

```

// Use the index to perform queries
// Refer to the MapIndex or MapRangeIndex interface for supported operations.
// The object attribute on which the index is created is the EmployeeCode.
// Assume that the EmployeeCode attribute is Integer type: the
// parameter that is passed into index operations has this data type.

Iterator iter = codeIndex.findLessEqual(new Integer(15));

// remove the dynamic index when no longer needed

bm.removeDynamicIndex("CODE");
// Close the session (optional in Version 7.1.1 and later) for improved performance
session.close();

```

What to do next

You can use the `DynamicIndexCallback` interface to get notifications at the indexing events. See “`DynamicIndexCallback` interface” for more information.

DynamicIndexCallback interface

The `DynamicIndexCallback` interface is designed for applications that want to get notifications at the indexing events of ready, error, or destroy. The `DynamicIndexCallback` is an optional parameter for the `createDynamicIndex` method of the `BackingMap`. With a registered `DynamicIndexCallback` instance, applications can run business logic upon receiving notification of an indexing event.

Indexing events

For example, the ready event means that the index is ready for use. When a notification for this event is received, an application can try to retrieve and use the application index interface instance.

Example: Using the DynamicIndexCallback interface

```

BackingMap personBackingMap = ivObjectGrid.getMap("person");
DynamicIndexCallback callback = new DynamicIndexCallbackImpl();
personBackingMap.createDynamicIndex("CODE", true, "employeeCode", callback);

class DynamicIndexCallbackImpl implements DynamicIndexCallback {
    public DynamicIndexCallbackImpl() {
    }

    public void ready(String indexName) {
        System.out.println("DynamicIndexCallbackImpl.ready() -> indexName = " + indexName);

        // Simulate what an application would do when notified that the index is ready.
        // Normally, the application would wait until the ready state is reached and then proceed
        // with any index usage logic.
        if("CODE".equals(indexName)) {
            ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
            ObjectGrid og = ogManager.createObjectGrid("grid");
            Session session = og.getSession();
            ObjectMap map = session.getMap("person");
            MapIndex codeIndex = (MapIndex) map.getIndex("CODE");
            Iterator iter = codeIndex.findAll(codeValue);

            // Close the session (optional in Version 7.1.1 and later) for improved performance
            session.close();
        }
    }

    public void error(String indexName, Throwable t) {
        System.out.println("DynamicIndexCallbackImpl.error() -> indexName = " + indexName);
        t.printStackTrace();
    }

    public void destroy(String indexName) {
        System.out.println("DynamicIndexCallbackImpl.destroy() -> indexName = " + indexName);
    }
}

```

Using Sessions to access data in the grid

Applications can begin and end transactions through the `Session` interface. The `Session` interface also provides access to the application-based `ObjectMap` and `JavaMap` interfaces.

Each ObjectMap or JavaMap instance is directly tied to a specific Session object. Each thread that wants access to an eXtreme Scale must first obtain a Session from the ObjectGrid object. A Session instance cannot be shared concurrently between threads. WebSphere eXtreme Scale does not use any thread local storage, but platform restrictions might limit the opportunity to pass a Session from one thread to another.

Methods

Get method

An application obtains a Session instance from an ObjectGrid object using the ObjectGrid.getSession method. The following example demonstrates how to obtain a Session instance:

```
ObjectGrid objectGrid = ...; Session sess = objectGrid.getSession();
```

After a Session is obtained, the thread keeps a reference to the session for its own use. Calling the getSession method multiple times returns a new Session object each time.

Transactions and Session methods

A Session can be used to begin, commit, or rollback transactions. Operations against BackingMaps using ObjectMaps and JavaMaps are most efficiently performed within a Session transaction. After a transaction has started, any changes to one or more BackingMaps in that transaction scope are stored in a special transaction cache until the transaction is committed. When a transaction is committed, the pending changes are applied to the BackingMaps and Loaders and become visible to any other clients of that ObjectGrid.

WebSphere eXtreme Scale also supports the ability to automatically commit transactions, also known as auto-commit. If any ObjectMap operations are performed outside of the context of an active transaction, an implicit transaction is started before the operation and the transaction is automatically committed before returning control to the application.

```
Session session = objectGrid.getSession();
ObjectMap objectMap = session.getMap("someMap");
session.begin();
objectMap.insert("key1", "value1");
objectMap.insert("key2", "value2");
session.commit();
objectMap.insert("key3", "value3"); // auto-commit
```

Session.flush method

The Session.flush method only makes sense when a Loader is associated with a BackingMap. The flush method invokes the Loader with the current set of changes in the transaction cache. The Loader applies the changes to the backend. These changes are not committed when the flush is invoked. If a Session transaction is committed after a flush invocation, only updates that happen after the flush invocation are applied to the Loader. If a Session transaction is rolled back after a flush invocation, the flushed changes are discarded with all other pending changes in the transaction. Use the Flush method sparingly because it limits the opportunity for batch operations against a Loader. Following is an example of the usage of the Session.flush method:

```

Session session = objectGrid.getSession();
session.begin();
// make some changes
...
session.flush(); // push these changes to the Loader, but don't commit yet
// make some more changes
...
session.commit();

```

NoWriteThrough method

Some maps are backed by a Loader, which provides persistent storage for the data in the map. Sometimes it is useful to commit data just to the eXtreme Scale map and not push data out to the Loader. The Session interface provides the `beginNoWriteThrough` method for this purpose. The `beginNoWriteThrough` method starts a transaction like the `begin` method. With the `beginNoWriteThrough` method, when the transaction is committed, the data is only committed to the in-memory map and is not committed to the persistent storage that is provided by the Loader. This method is very useful when performing data preload on the map.

When using a distributed ObjectGrid instance, the `beginNoWriteThrough` method is useful for making changes to the near cache only, without modifying the far cache on the server. If the data is known to be stale in the near cache, using the `beginNoWriteThrough` method can allow entries to be invalidated on the near cache without invalidating them on the server as well.

The Session interface also provides the `isWriteThroughEnabled` method to determine what type of transaction is currently active.

```

Session session = objectGrid.getSession();
session.beginNoWriteThrough();
// make some changes ...
session.commit(); // these changes will not get pushed to the Loader

```

Obtain the TxID object method

The TxID object is an opaque object that identifies the active transaction. Use the TxID object for the following purposes:

- For comparison when you are looking for a particular transaction.
- To store shared data between the TransactionCallback and Loader objects.

See TransactionCallback plug-in and Loaders for additional information about the Object slot feature.

Performance monitoring method

If you are using eXtreme Scale within WebSphere Application Server, it might be necessary to reset the transaction type for performance monitoring. You can set the transaction type with the `setTransactionType` method. See Monitoring ObjectGrid performance with WebSphere Application Server performance monitoring infrastructure (PMI) for more information about the `setTransactionType` method.

Process a complete LogSequence method

WebSphere eXtreme Scale can propagate sets of map changes to ObjectGrid listeners as a means of distributing maps from one Java virtual machine to another. To make it easier for the listener to process the received LogSequences, the Session interface provides the `processLogSequence` method. This method examines each

LogElement within the LogSequence and performs the appropriate operation, for example, insert, update, invalidate, and so on, against the BackingMap that is identified by the LogSequence MapName. An ObjectGrid Session must be available before the processLogSequence method is invoked. The application is also responsible for issuing the appropriate commit or rollback calls to complete the Session. Autocommit processing is not available for this method invocation. Normal processing by the receiving ObjectGridEventListener at the remote JVM would be to start a Session using the beginNoWriteThrough method, which prevents endless propagation of changes, followed by a call to this processLogSequence method, and then committing or rolling back the transaction.

```
// Use the Session object that was passed in during
//ObjectGridEventListener.initialization...
session.beginNoWriteThrough();
// process the received LogSequence
try {
    session.processLogSequence(receivedLogSequence);
} catch (Exception e) {
    session.rollback(); throw e;
}
// commit the changes
session.commit();
```

markRollbackOnly method

This method is used to mark the current transaction as "rollback only". Marking a transaction "rollback only" ensures that even if the commit method is called by application, the transaction is rolled back. This method is typically used by ObjectGrid itself or by the application when it knows that data corruption could occur if the transaction was allowed to be committed. After this method is called, the Throwable object that is passed to this method is chained to the com.ibm.websphere.objectgrid.TransactionException exception that results by the commit method if it is called on a Session that was previously marked a "rollback only". Any subsequent calls to this method for a transaction that is already marked as "rollback only" is ignored. That is, only the first call that passes a non-null Throwable reference is used. Once the marked transaction is completed, the "rollback only" mark is removed so that the next transaction that is started by the Session can be committed.

isMarkedRollbackOnly method

Returns if Session is currently marked as "rollback only". Boolean true is returned by this method if and only if markRollbackOnly method was previously called on this Session and the transaction started by the Session is still active.

setTransactionTimeout method

Set transaction timeout for next transaction started by this Session to a specified number of seconds. This method does not affect the transaction timeout of any transactions previously started by this Session. It only affects transactions that are started after this method is called. If this method is never called, then the timeout value that was passed to the setTxTimeout method of the com.ibm.websphere.objectgrid.ObjectGrid method is used.

getTransactionTimeout method

This method returns the transaction timeout value in seconds. The last value that was passed as the timeout value to the setTransactionTimeout method is returned

by this method. If the `setTransactionTimeout` method is never called, then the timeout value that was passed to the `setTxTimeout` method of the `com.ibm.websphere.objectgrid.ObjectGrid` method is used.

transactionTimedOut

This method returns boolean true if the current transaction that was started by this Session has timed out.

isFlushing method

This method returns boolean true if and only if all transaction changes are being flushed out to the Loader plug-in as a result of the flush method of Session interface being invoked. A Loader plug-in may find this method useful when it needs to know why its `batchUpdate` method was invoked.

isCommitting method

This method returns boolean true if and only if all transaction changes are being committed as a result of the commit method of Session interface being invoked. A Loader plug-in might find this method useful when it needs to know why its `batchUpdate` method was invoked.

setRequestRetryTimeout method

This method sets the request retry timeout value for the session in milliseconds. If the client set a request retry timeout, the session setting overrides the client value.

getRequestRetryTimeout method

This method gets the current request retry timeout setting on the session. A value of -1 indicates that the timeout is not set. A value of 0 indicates it is in fail-fast mode. A value greater than 0 indicates the timeout setting in milliseconds.

SessionHandle for routing

When you are using a per-container partition placement policy, you can use a SessionHandle object. A SessionHandle object contains partition information for the current Session and can be reused for a new Session.

A SessionHandle object includes information for the partition to which the current Session is bound. SessionHandle is extremely useful for the per-container partition placement policy and can be serialized with standard Java serialization.

If you have a SessionHandle object, you can apply that handle to a Session with the `setSessionHandle(SessionHandle target)` method, passing the handle in as the target. You can retrieve the SessionHandle object with the `Session.getSessionHandle` method.

Because it is only applicable in a per-container placement scenario, getting the SessionHandle object throws an `IllegalStateException` if a given data grid has multiple per-container map sets or has no per-container map sets. If you do not invoke the `setSessionHandle` method before calling the `getSessionHandle` method, the appropriate SessionHandle object is selected based on your client properties configuration.

You can also use the `SessionHandleTransformer` helper class to convert the handle into different formats. The methods of this class can change a handle's representation from byte array to instance, string to instance, and vice versa for both cases, and can also write the handle's contents into the output stream.

For an example on how you can use a `SessionHandle` object, see the zone-preferred routing topic in the .

SessionHandle integration

A `SessionHandle` object includes the partition information for the `Session` to which it is bound and facilitates request routing. `SessionHandle` objects apply to the per-container partition placement scenario only.

SessionHandle object for request routing

You can bind a `SessionHandle` object to a `Session` in the following ways:

Tip: In each of the following method calls, after a `SessionHandle` object is bound to a `Session`, the `SessionHandle` object can be obtained from the `Session.getSessionHandle` method.

- **Call the `Session.getSessionHandle` method:** When this method is called, if no `SessionHandle` object is bound to the `Session`, a `SessionHandle` object is selected randomly and bound to the `Session`.
- **Call transactional create, read, update, delete operations:** When these methods are called or at commit time, if no `SessionHandle` object is bound to the `Session`, a `SessionHandle` object is selected randomly and bound to the `Session`.
- **Call `ObjectMap.getNextKey` method:** When this method is called, if no `SessionHandle` object is bound to the `Session`, the operation request is randomly routed to individual partitions until a key is obtained. If a key is returned from a partition, a `SessionHandle` object that corresponds to that partition is bound to the `Session`. If no key is found, no `SessionHandle` is bound to the `Session`.
- **Call the `QueryQueue.getNextEntity` or `QueryQueue.getNextEntities` methods:** At the time this method is called, if no `SessionHandle` object is bound to the `Session`, the operation request is randomly routed to individual partitions until an object is obtained. If an object is returned from a partition, a `SessionHandle` object that corresponds to that partition is bound to the `Session`. If no object is found, no `SessionHandle` is bound to the `Session`.
- **Set a `SessionHandle` with the `Session.setSessionHandle(SessionHandle sh)` method:** If a `SessionHandle` is obtained from the `Session.getSessionHandle` method, the `SessionHandle` can be bound to a `Session`. Setting a `SessionHandle` influences request routing within the scope of the `Session` to which it is bound.

The `Session.getSessionHandle` method always returns a `SessionHandle` object. The method also automatically binds a `SessionHandle` on the `Session` if no `SessionHandle` object is bound to the `Session`. If you want to verify whether a `Session` has a `SessionHandle` object only, call the `Session.isSessionHandleSet` method. If this method returns a value of `false`, no `SessionHandle` object is currently bound to the `Session`.

Major operation types in the per-container placement scenario

A summary of the routing behavior of major operation types in the per-container partition placement scenario with respect to `SessionHandle` objects follows.

- **Session object with bound `SessionHandle` object**

- Index - MapIndex and MapRangeIndex API: SessionHandle
- Query and ObjectQuery: SessionHandle
- Agent - MapGridAgent and ReduceGridAgent API: SessionHandle
- ObjectMap.Clear: SessionHandle
- ObjectMap.getNextKey: SessionHandle
- QueryQueue.getNextEntity, QueryQueue.getNextEntities: SessionHandle
- Transactional create, retrieve, update, and delete operations (ObjectMap API and EntityManager API): SessionHandle
- **Session object without bound SessionHandle object**
 - Index - MapIndex and MapRangeIndex API: All current active partitions
 - Query and ObjectQuery: Specified partition with setPartition method of Query and ObjectQuery
 - Agent - MapGridAgent and ReduceGridAgent
 - Not supported: ReduceGridAgent.reduce(Session s, ObjectMap map, Collection keys) and MapGridAgent.process(Session s, ObjectMap map, Object key) method.
 - All current active partitions: ReduceGridAgent.reduce(Session s, ObjectMap map) and MapGridAgent.processAllEntries(Session s, ObjectMap map) method.
 - ObjectMap.clear: All current active partitions.
 - ObjectMap.getNextKey: Binds a SessionHandle to the Session if a key is returned from one of the randomly selected partitions. If no key is returned, the Session is not bound to a SessionHandle.
 - QueryQueue: Specifies a partition with the QueryQueue.setPartition method. If no partition is set, the method randomly selects a partition to return. If an object is returned, the current Session is bound with the SessionHandle that is bound to the partition that returns the object. If no object is returned, the Session is not bound to a SessionHandle.
 - Transactional create, retrieve, update, and delete operations (ObjectMap API and EntityManager API): Randomly select a partition.

In most cases, use SessionHandle to control routing to a particular partition. You can retrieve and cache the SessionHandle from the Session that inserts data. After caching the SessionHandle, you can set it on another Session so that you can route requests to the partition specified by the cached SessionHandle. To perform operations such as ObjectMap.clear without SessionHandle, you can temporarily set the SessionHandle to null by calling Session.setSessionHandle(null). Without a specified SessionHandle, operations run on all current active partitions.

- **QueryQueue routing behavior**

In the per-container partition placement scenario, you can use SessionHandle to control routing of getNextEntity and getNextEntities methods of the QueryQueue API. If the Session is bound to a SessionHandle, requests route to the partition to which the SessionHandle is bound. If the Session is not bound to a SessionHandle, requests are routed to the partition set with the QueryQueue.setPartition method if a partition has been set in this way. If the Session has no bound SessionHandle or partition, a randomly selected partition are returned. If no such partition is found, the process stops and no SessionHandle is bound to the current Session.

The following snippet of code shows how to use SessionHandle objects.

```

Session ogSession = objectGrid.getSession();

// binding the SessionHandle
SessionHandle sessionHandle = ogSession.getSessionHandle();

ogSession.begin();
ObjectMap map = ogSession.getMap("planet");
map.insert("planet1", "mercury");

// transaction is routed to partition specified by SessionHandle
ogSession.commit();

// cache the SessionHandle that inserts data
SessionHandle cachedSessionHandle = ogSession.getSessionHandle();

// verify if SessionHandle is set on the Session
boolean isSessionHandleSet = ogSession.isSessionHandleSet();

// temporarily unbind the SessionHandle from the Session
if(isSessionHandleSet) {
    ogSession.setSessionHandle(null);
}

// if the Session has no SessionHandle bound,
// the clear operation will run on all current active partitions
// and thus remove all data from the map in the entire grid
map.clear();

// after clear is done, reset the SessionHandle back,
// if the Session needs to use previous SessionHandle.
// Optionally, calling getSessionHandle can get a new SessionHandle
ogSession.setSessionHandle(cachedSessionHandle);

```

Application design considerations

In the per-container placement strategy scenario, use the SessionHandle object for most operations. The SessionHandle object controls routing to partitions. To retrieve data, the SessionHandle object that you bind to the Session must be same SessionHandle object from any insert data transaction.

When you want to perform an operation without a SessionHandle set on the Session, you can unbind a SessionHandle from a Session by making a Session.setSessionHandle(null) method call.

When a Session is bound to a SessionHandle, all operation requests route to the partition that is specified by the SessionHandle object. Without the SessionHandle object set, operations route to either all partitions or a randomly selected partition.

Caching objects with no relationships involved (ObjectMap API)

ObjectMaps are like Java Maps that allow data to be stored as key-value pairs. ObjectMaps provide a simple and intuitive approach for the application to store data. An ObjectMap is ideal for caching objects that have no relationships involved. If object relationships are involved, then you should use the EntityManager API.

For more information about the EntityManager API, see “Caching objects and their relationships (EntityManager API)” on page 164.

Applications typically obtain a WebSphere eXtreme Scale reference and then obtain a Session object from the reference for each thread. Sessions cannot be shared between threads. The getMap method of Session returns a reference to an ObjectMap to use for this thread.

Routing cache objects to the same partition

When an eXtreme Scale configuration uses the fixed partition placement strategy, it depends on hashing the key to a partition to insert, get, update, or remove the value. The hashCode method is called on the key and it must be well defined if a custom key is created. However, another option is to use the PartitionableKey interface. With the PartitionableKey interface, you can use an object other than the key to hash to a partition.

You can use the PartitionableKey interface in situations where there are multiple maps and the data you commit is related and thus should be put on the same partition. WebSphere eXtreme Scale does not support two-phase commit so multiple map transactions should not be committed if they span multiple partitions. If the PartitionableKey hashes to the same partition for keys in different maps in the same map set, they can be committed together.

You can also use the PartitionableKey interface when groups of keys should be put on the same partition, but not necessarily during a single transaction. If keys should be hashed based on location, department, domain type, or some other type of identifier, children keys can be given a parent PartitionableKey.

For example, employees should hash to the same partition as their department. Each employee key would have a PartitionableKey object that belongs to the department map. Then, both the employee and department would hash to the same partition.

The PartitionableKey interface supplies one method, called `ibmGetPartition`. The object returned from this method must implement the `hashCode` method. The result returned from using the alternate `hashCode` will be used to route the key to a partition.

Example

See the following example key that demonstrates how to use the PartitionableKey interface and the hashCode method to clone an existing key, and route the resulting keys to the same partition.

```
package com.ibm.websphere.cjtester;

import java.io.Serializable;

import com.ibm.websphere.objectgrid.plugins.PartitionableKey;

public class RoutableKey implements Serializable, Cloneable, PartitionableKey {
    private static final long serialVersionUID = 1L;

    // The data that makes up the actual data object key.
    public final String realKey;

    // The key of the data object you want to use for routing.
    // This is typically the key of a parent object.
    public final Object keyToRouteWith;

    public RoutableKey(String realKey, Object keyToRouteWith) {
        super();
        this.realKey = realKey;
    }
}
```

```

        this.keyToRouteWith = keyToRouteWith;
    }

    /**
     * Return the hashCode of the key we are using for routing.
     * If not supplied, eXtreme Scale will use the hashCode of THIS key.
     */
    public Object ibmGetPartition() {
        return new Integer(keyToRouteWith.hashCode());
    }

    @Override
    public RoutableKey clone() throws CloneNotSupportedException {
        return (RoutableKey) super.clone();
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((keyToRouteWith == null) ? 0 : keyToRouteWith.hashCode());
        result = prime * result + ((realKey == null) ? 0 : realKey.hashCode());
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        RoutableKey other = (RoutableKey) obj;
        if (keyToRouteWith == null) {
            if (other.keyToRouteWith != null) return false;
        } else if (!keyToRouteWith.equals(other.keyToRouteWith)) return false;
        if (realKey == null) {
            if (other.realKey != null) return false;
        } else if (!realKey.equals(other.realKey)) return false;
        return true;
    }
}

```

Introduction to ObjectMap

The ObjectMap interface is used for transactional interaction between applications and BackingMaps.

Purpose

An ObjectMap instance is obtained from a Session object that corresponds to the current thread. The ObjectMap interface is the main vehicle that applications use to make changes to entries in a BackingMap.

Obtain an ObjectMap instance

An application gets an ObjectMap instance from a Session object using the Session.getMap(String) method. The following code snippet demonstrates how to obtain an ObjectMap instance:

```

ObjectGrid objectGrid = ...;
BackingMap backingMap = objectGrid.defineMap("mapA");
Session sess = objectGrid.getSession();
ObjectMap objectMap = sess.getMap("mapA");

```

Each ObjectMap instance corresponds to a particular Session object. Calling the getMap method multiple times on a particular Session object with the same

BackingMap name always returns the same ObjectMap instance.

Automatically commit transactions

Operations against BackingMaps that use ObjectMaps and JavaMaps are performed most efficiently within a Session transaction. WebSphere eXtreme Scale provides autocommit support when methods on the ObjectMap and JavaMap interfaces are called outside of a Session transaction. The methods start an implicit transaction, perform the requested operation, and commit the implicit transaction.

Method semantics

An explanation of the semantics behind each method on the ObjectMap and JavaMap interfaces follows.

containsKey method

The containsKey method determines if a key has a value in the BackingMap or Loader. If null values are supported by an application, this method can be used to determine if a null reference that is returned from a get operation refers to a null value or indicates that the BackingMap and Loader do not contain the key.

flush method

The flush method semantics are similar to the flush method on the Session interface. The notable difference is that the Session flush applies the current pending changes for all of the maps that are modified in the current session. With this method, only the changes in this ObjectMap instance are flushed to the loader.

get method

The get method fetches the entry from the BackingMap instance. If the entry is not found in the BackingMap instance but a Loader is associated with the BackingMap instance, the BackingMap instance attempts to fetch the entry from the Loader. The getAll method is provided to allow batch fetch processing.

getForUpdate method

The getForUpdate method is the same as the get method, but using the getForUpdate method tells the BackingMap and Loader that the intention is to update the entry. A Loader can use this hint to issue a SELECT for UPDATE query to a database backend. If a pessimistic locking strategy is defined for the BackingMap, the lock manager locks the entry. The getAllForUpdate method is provided to allow batch fetch processing.

insert method

The insert method inserts an entry into the BackingMap and the Loader. Using this method tells the BackingMap and Loader that you want to insert an entry that did not previously exist. When you invoke this method on an existing entry, an exception occurs when the method is invoked or when the current transaction is committed.

invalidate method

The semantics of the invalidate method depend on the value of the **isGlobal** parameter that is passed to the method. The invalidateAll method is provided to allow batch invalidate processing.

Local invalidation is specified when the value false is passed as the **isGlobal** parameter of the invalidate method. Local invalidation discards any changes to the entry in the transaction cache. If the application issues a

get method, the entry is fetched from the last committed value in the BackingMap. If no entry is present in the BackingMap, the entry is fetched from the last flushed or committed value in the Loader. When a transaction is committed, any entries that are marked as locally invalidated have no impact on the BackingMap. Any changes that were flushed to the Loader are still committed even if the entry was invalidated.

Global invalidation is specified when true is passed as the **isGlobal** parameter of the invalidate method. Global invalidation discards any pending changes to the entry in the transaction cache and bypasses the BackingMap value on subsequent operations that are performed on the entry. When a transaction is committed, any entries that are marked as globally invalidated are evicted from the BackingMap. Consider the following use case for invalidation as an example: The BackingMap is backed by a database table that has an auto increment column. Increment columns are useful for assigning unique numbers to records. The application inserts an entry. After the insert, the application needs to know the sequence number for the inserted row. It knows that its copy of the object is old, so it uses global invalidation to get the value from the Loader. The following code demonstrates this use case:

```
Session sess = objectGrid.getSession();
ObjectMap map = sess.getMap("mymap");
sess.begin();
map.insert("Billy", new Person("Joe", "Bloggs", "Manhattan"));
sess.flush();
map.invalidate("Billy", true);
Person p = map.get("Billy");
System.out.println("Version column is: " + p.getVersion());
map.commit();
// Close the session (optional in Version 7.1.1 and later) for improved performance
session.close();
```

This code sample adds an entry for Billy. The version attribute of Person is set using an auto-increment column in the database. The application first performs an insert command. It then issues a flush, which causes the insert to be sent to the Loader and database. The database sets the version column to the next number in the sequence, which makes the Person object in the transaction outdated. To update the object, the application is globally invalidated. The next get method that is issued gets the entry from the Loader, ignoring the transaction value. The entry is fetched from the database with the updated version value.

put method

The semantics of the put method are dependent on whether a previous get method was invoked in the transaction for the key. If the application issues a get operation that returns an entry that exists in the BackingMap or Loader, the put method invocation is interpreted as an update and returns the previous value in the transaction. If a put method invocation ran without a previous get method invocation, or a previous get method invocation did not find an entry, the operation is interpreted as an insert. The semantics of the insert and update methods apply when the put operation is committed. The putAll method is provided to enable batch insert and update processing.

remove method

The remove method removes the entry from the BackingMap and the Loader, if a Loader is plugged in. The value of the object that was removed

is returned by this method. If the object does not exist, this method returns a null value. The `removeAll` method is provided to enable batch deletion processing without the return values.

setCopyMode method

The `setCopyMode` method specifies a `CopyMode` value for this `ObjectMap`. With this method, an application can override the `CopyMode` value that is specified on the `BackingMap`. The specified `CopyMode` value is in effect until `clearCopyMode` method is invoked. Both methods are invoked outside of transactional bounds. A `CopyMode` value cannot be changed in the middle of a transaction.

touch method

The `touch` method updates the last access time for an entry. This method does not retrieve the value from the `BackingMap`. Use this method in its own transaction. If the provided key does not exist in the `BackingMap` because of invalidation or removal, an exception occurs during commit processing.

update method

The `update` method explicitly updates an entry in the `BackingMap` and the `Loader`. Using this method indicates to the `BackingMap` and `Loader` that you want to update an existing entry. An exception occurs if you invoke this method on an entry that does not exist when the method is invoked or during commit processing.

getIndex method

The `getIndex` method attempts to obtain a named index that is built on the `BackingMap`. The index cannot be shared between threads and works on the same rules as a `Session`. The returned index object should be cast to the right application index interface such as the `MapIndex` interface, the `MapRangeIndex` interface, or a custom index interface.

clear method

The `clear` method removes all cache entries from a map from all partitions. This operation is an auto-commit function, so no active transaction should be present when calling `clear`.

Note: The `clear` method only clears out the map on which it is called, leaving any related entity maps unaffected. This method does not invoke the `Loader` plug-in.

Dynamic maps

With dynamic maps, you can create maps after the data grid has already been initialized.

In previous versions, WebSphere eXtreme Scale has required you to define maps before initializing the `ObjectGrid`. As a result, you had to create all of the maps to be used before running transactions against any of the maps.

Advantages of dynamic maps

The introduction of dynamic maps reduces the restriction of having to define all maps before initialization. Through the use of template maps, you can create maps after the `ObjectGrid` has been initialized.

Template maps are defined in the `ObjectGrid` XML file. Template comparisons are run when a `Session` requests a map that has not been previously defined. If the new map name matches the regular expression of a template map, the map is

created dynamically and assigned the name of the requested map. This newly created map inherits all of the settings of the template map as defined by the ObjectGrid XML file.

Creating dynamic maps

Dynamic map creation is tied to the `Session.getMap(String)` method. Calls to this method return an `ObjectMap` based on the `BackingMap` that was configured by the ObjectGrid XML file.

Passing in a `String` that matches the regular expression of a template map results in the creation of an `ObjectMap` and an associated `BackingMap`.

See the API documentation for more information about the `Session.getMap(String cacheName)` method.

Defining a template map in XML is as simple as setting a template Boolean attribute on the `BackingMap` element. When `template` is set to `true`, the name of the `BackingMap` is interpreted as a regular expression.

WebSphere eXtreme Scale uses Java regular expression pattern matching. For more information about the regular expression engine in Java, see the API documentation for the `java.util.regex` package and classes.

Note: When you are defining template maps, verify that the map names are unique enough so that the application can match to only one of the template maps with the `Session.getMap(String mapName)` method. If the `getMap()` method matches more than one template map pattern, an `IllegalArgumentException` exception results.

A sample ObjectGrid XML file with a template map defined follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="accounting">
      <BackingMap name="payroll" readOnly="false" />
      <BackingMap name="templateMap.*" template="true"
        pluginCollectionRef="templatePlugins" lockStrategy="PESSIMISTIC" />
    </objectGrid>
  </objectGrids>

  <BackingMapPluginCollections>
    <BackingMapPluginCollection id="templatePlugins">
      <bean id="Evictor"
        className="com.ibm.websphere.objectgrid.plugins.builtins.LFUEvictor" />
    </BackingMapPluginCollection>
  </BackingMapPluginCollections>
</objectGridConfig>
```

The previous XML file defines one template map and one non-template map. The name of the template map is a regular expression: `templateMap.*`. When the `Session.getMap(String)` method is called with a map name matching this regular expression, the application creates a map.

Example

Configuration of a template map is required in order to create a dynamic map. Add the template Boolean to a backingMap in the ObjectGrid XML file.

```
<backingMap name="templateMap.*" template="true" />
```

The name of the template map is treated as a regular expression.

Calling the `Session.getMap(String cacheName)` method with a `cacheName` that is a match for the regular expression results in the creation of the dynamic map. An `ObjectMap` object is returned from this method call, and an associated `BackingMap` object is created.

```
Session session = og.getSession();
ObjectMap map = session.getMap("templateMap1");
```

The newly created map is configured with all the attributes and plug-ins that were defined on the template map definition. Consider again the previous ObjectGrid XML file.

A dynamic map created based on the template map in this XML file would have an evictor configured and its lock strategy would be pessimistic.

Note: A template is not an actual `BackingMap`. That is, the “accounting” ObjectGrid does not contain an actual “`templateMap.*`” map. The template is only used as a basis for dynamic map creation. However, you must include the dynamic map in the `mapRef` element of the deployment policy XML file named exactly as in the ObjectGrid XML. This element identifies which `mapSet` in which the dynamic maps are defined.

Consider the change in behavior of the `Session.getMap(String cacheName)` method when using template maps. Before WebSphere eXtreme Scale Version 7.0, all calls to the `Session.getMap(String cacheName)` method resulted in an `UndefinedMapException` exception if the map requested did not exist. With dynamic maps, every name that matches the regular expression for a template map results in map creation. Be sure to note the number of maps that your application creates, particularly if your regular expression is generic.

Also, `ObjectGridPermission.DYNAMIC_MAP` is required for dynamic map creation when eXtreme Scale security is enabled. This permission is checked when the `Session.getMap(String)` method is called. For more information, see the information about application client authorization in the *Product Overview*.

Additional examples

objectGrid.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
  <objectGrid name="session">
    <backingMap name="objectgrid.session.metadata.dynamicmap.*" template="true"
      lockStrategy="PESSIMISTIC" ttlEvictorType="LAST_ACCESS_TIME">
    <backingMap name="objectgrid.session.attribute.dynamicmap.*"
      template="true" lockStrategy="OPTIMISTIC"/>
    <backingMap name="datagrid.session.global.ids.dynamicmap.*"
```

```

        lockStrategy="PESSIMISTIC"/>
    </objectGrid>
</objectGrids>
</objectGridConfig>

```

objectGridDeployment.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<deploymentPolicy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/deploymentPolicy
        ../deploymentPolicy.xsd"
    xmlns="http://ibm.com/ws/objectgrid/deploymentPolicy">
    <objectgridDeployment objectgridName="session">
        <mapSet name="mapSet2" numberOfPartitions="5" minSyncReplicas="0"
            maxSyncReplicas="0" maxAsyncReplicas="1" developmentMode="false"
            placementStrategy="PER_CONTAINER">
            <map ref="logical.name"/>
            <map ref="objectgrid.session.metadata.dynamicmap.*"/>
            <map ref="objectgrid.session.attribute.dynamicmap.*"/>
            <map ref="datagrid.session.global.ids"/>
        </mapSet>
    </objectgridDeployment>
</deploymentPolicy>

```

Limitations and considerations

Limitations:

- The QuerySchema element does not support the template for mapName.
- You cannot use entities with dynamic maps.
- An entity BackingMap is implicitly defined, mapped to the entity through the class name.

Considerations:

- Many plug-ins have no way of determining the map with which each plug-in is associated.
- Other plug-ins differentiate themselves by using a map name or BackingMap name as an argument.
- When you are defining template maps, verify that the map names are unique enough so that the application can match to only one of the template maps using the Session.getMap(String mapName) method. If the getMap() method matches more than one template map pattern, an IllegalArgumentException exception results.

ObjectMap and JavaMap

A JavaMap instance is obtained from an ObjectMap object. The JavaMap interface has the same method signatures as ObjectMap, but with different exception handling. JavaMap extends the java.util.Map interface, so all exceptions are instances of the java.lang.RuntimeException class. Because JavaMap extends the java.util.Map interface, it is easy to quickly use WebSphere eXtreme Scale with an existing application that uses a java.util.Map interface for object caching.

Obtain a JavaMap instance

An application gets a JavaMap instance from an ObjectMap object using the ObjectMap.getJavaMap method. The following code snippet demonstrates how to obtain a JavaMap instance.

```

ObjectGrid objectGrid = ...;
BackingMap backingMap = objectGrid.defineMap("mapA");
Session sess = objectGrid.getSession();
ObjectMap objectMap = sess.getMap("mapA");
java.util.Map map = objectMap.getJavaMap();
JavaMap javaMap = (JavaMap) javaMap;

```

A JavaMap is backed by the ObjectMap from which it was obtained. Calling the getJavaMap method multiple times using a particular ObjectMap always returns the same JavaMap instance.

Methods

The JavaMap interface only supports a subset of the methods on the java.util.Map interface. The java.util.Map interface supports the following methods:

containsKey(java.lang.Object) method

get(java.lang.Object) method

put(java.lang.Object, java.lang.Object) method

putAll(java.util.Map) method

remove(java.lang.Object) method

clear()

All other methods inherited from the java.util.Map interface result in a java.lang.UnsupportedOperationException exception.

Maps as FIFO queues

With WebSphere eXtreme Scale, you can provide a first-in first-out (FIFO) queue-like capability for all maps. WebSphere eXtreme Scale tracks the insertion order for all maps. A client can ask a map for the next unlocked entry in a map in the order of insertion and lock the entry. This process allows multiple clients to consume entries from the map efficiently.

FIFO example

The following code snippet shows a client entering a loop to process entries from the map until the map is exhausted. The loop starts a transaction, then calls the ObjectMap.getNextKey(5000) method. This method returns the key of the next available unlocked entry and locks it. If the transaction is blocked for more than 5000 milliseconds, then the method returns null.

```

Session session = ...;
ObjectMap map = session.getMap("xxx");
// this needs to be set somewhere to stop this loop
boolean timeToStop = false;

while(!timeToStop)
{
    session.begin();
    Object msgKey = map.getNextKey(5000);
    if(msgKey == null)
    {
        // current partition is exhausted, call it again in
        // a new transaction to move to next partition
        session.rollback();
        continue;
    }
    Message m = (Message)map.get(msgKey);
    // now consume the message
}

```

```

...
// need to remove it
map.remove(msgKey);
session.commit();
}

```

Local mode versus client mode

If the application is using a local core, that is, it is not a client, then the mechanism works as described previously.

For client mode, if the Java virtual machine (JVM) is a client, then the client initially connects to a random partition primary. If no work exists in that partition, then the client moves to the next partition to look for work. The client either finds a partition with entries or loops around to the initial random partition. If the client loops around to the initial partition, then it returns a null value to the application. If the client finds a partition with a map that has entries, then it consumes entries from there until no entries are available for the timeout period. After the timeout passes, then null is returned. This action means that when null is returned and a partitioned map is used, then it you should start a new transaction and resume listening. The previous code sample fragment has this behavior.

Example

When you are running as a client and a key is returned, that transaction is now bound to the partition with the entry for that key. If you do not want to update any other maps during that transaction, then a problem does not exist. If you do want to update, then you can only update maps from the same partition as the map from which you got the key. The entry that is returned from the getNextKey method needs to give the application a way to discover relevant data in that partition. As an example, if you have two maps; one for events and another for jobs that the events impact. You define the two maps with the following entities:

Job.java

```

package tutorial.fifo;

import com.ibm.websphere.projector.annotations.Entity;
import com.ibm.websphere.projector.annotations.Id;

@Entity
public class Job
{
    @Id String jobId;

    int jobState;
}

```

JobEvent.java

```

package tutorial.fifo;

import com.ibm.websphere.projector.annotations.Entity;
import com.ibm.websphere.projector.annotations.Id;
import com.ibm.websphere.projector.annotations.OneToOne;

@Entity
public class JobEvent
{
    @Id String eventId;
    @OneToOne Job job;
}

```

The job has an ID and state, which is an integer. Suppose you want to increment the state whenever an event arrived. The events are stored in the JobEvent Map. Each entry has a reference to the job the event concerns. The code for the listener to do this looks like the following example:

JobEventListener.java

```
package tutorial.fifo;

import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.em.EntityManager;

public class JobEventListener
{
    boolean stopListening;

    public synchronized void stopListening()
    {
        stopListening = true;
    }

    synchronized boolean isStopped()
    {
        return stopListening;
    }

    public void processJobEvents(Session session)
        throws ObjectGridException
    {
        EntityManager em = session.getEntityManager();
        ObjectMap jobEvents = session.getMap("JobEvent");
        while(!isStopped())
        {
            em.getTransaction().begin();

            Object jobEventKey = jobEvents.getNextKey(5000);
            if(jobEventKey == null)
            {
                em.getTransaction().rollback();
                continue;
            }
            JobEvent event = (JobEvent)em.find(JobEvent.class, jobEventKey);
            // process the event, here we just increment the
            // job state
            event.job.jobState++;
            em.getTransaction().commit();
        }
    }
}
```

The listener is started on a thread by the application. The listener runs until the stopListening method is called. The processJobEvents method is run on the thread until the stopListening method is called. The loop blocks waiting for an eventKey from the JobEvent Map and then uses the EntityManager to access the event object, dereference to the job and increment the state.

The EntityManager API does not have a getNextKey method, but the ObjectMap does. So, the code uses the ObjectMap for JobEvent to get the key. If a map is used with entities then it does not store objects anymore. Instead, it stores Tuples; a Tuple object for the key and a Tuple object for the value. The EntityManager.find method accepts a Tuple for the key.

The code to create an event looks like the following example:

```
em.getTransaction().begin();
Job job = em.find(Job.class, "Job Key");
JobEvent event = new JobEvent();
event.id = Random.toString();
event.job = job;
em.persist(event); // insert it
em.getTransaction().commit();
```

You find the job for the event, construct an event, point it to the job, insert it in the JobEvent Map and commit the transaction.

Loaders and FIFO maps

If you want to back a map that is used as a FIFO queue with a Loader, then you might need to do some additional work. If the order of the entries in the map is not a concern, you have no extra work. If the order is important, then you need to add a sequence number to all of the inserted records when you are persisting the records to the backend. The preload mechanism should be written to insert the records on startup using this order.

Caching objects and their relationships (EntityManager API)

Most cache products use map-based APIs to store data as key-value pairs. The ObjectMap API and the dynamic cache in WebSphere Application Server, among others, use this approach. However, map-based APIs have limitations. The EntityManager API simplifies the interaction with the data grid by providing an easy way to declare and interact with a complex graph of related objects.

Map-based API limitations

If you are using a map-based API, such as the dynamic cache in WebSphere Application Server or the ObjectMap API, take the following limitations into consideration:

- Indexes and queries must use reflection to query fields and properties in cache objects.
- Custom data serialization is required to achieve performance for complex objects.
- It is difficult to work with graphs of objects. You must develop the application to store artificial references between objects and manually join the objects.

Benefits of the EntityManager API

The EntityManager API uses the existing map-based infrastructure, but it converts entity objects to and from tuples before storing or reading them from the map. An entity object is transformed into a key tuple and a value tuple, which are then stored as key-value pairs. A tuple is an array of primitive attributes.

This set of APIs follows the Plain Old Java Object (POJO) style of programming that is adopted by most frameworks.

Relationship management

Object-oriented languages such as Java, and relational databases support relationships or associations. Relationships decrease the amount of storage through the use of object references or foreign keys.

When you are using relationships in a data grid, the data must be organized in a constrained tree. One root type must exist in the tree and all children must be

associated to only one root. For example: Department can have many Employees and an Employee can have many Projects. But a Project cannot have many Employees that belong to different departments. Once a root is defined, all access to that root object and its descendants are managed through the root. WebSphere eXtreme Scale uses the hash code of the root object's key to choose a partition. For example:

```
partition = (hashCode MOD numPartitions).
```

When all of the data for a relationship is tied to a single object instance, the entire tree can be collocated in a single partition and can be accessed very efficiently using one transaction. If the data spans multiple relationships, then multiple partitions must be involved which involves additional remote calls, which can lead to performance bottlenecks.

Reference data

Some relationships include look-up or reference data such as: CountryName. For look-up or reference data, the data must exist in every partition. The data can be accessed by any root key and the same result is returned. Reference data such as this should only be used in cases where the data is fairly static. Updating this data can be expensive because the data needs to be updated in every partition. The DataGrid API is a common technique to keeping reference data up-to-date.

Costs and benefits of normalization

Normalizing the data using relationships can help reduce the amount of memory used by the data grid since duplication of data is decreased. However, in general, the more relational data that is added, the less it will scale out. When data is grouped together, it becomes more expensive to maintain the relationships and to keep the sizes manageable. Since the grid partitions data based on the key of the root of the tree, the size of the tree isn't taken into account. Therefore, if you have a lot of relationships for one tree instance, the data grid may become unbalanced, causing one partition to hold more data than the others.

When the data is denormalized or flattened, the data that would normally be shared between two objects is instead duplicated and each table can be partitioned independently, providing a much more balanced data grid. Although this increases the amount of memory used, it allows the application to scale since a single row of data can be accessed that has all of the necessary data. This is ideal for read-mostly grids since maintaining the data becomes more expensive.

For more information, see *Classifying XTP systems and scaling*.

Managing relationships using the data access APIs

The ObjectMap API is the fastest, most flexible and granular of the data access APIs, providing a transactional, session-based approach at accessing data in the grid of maps. The ObjectMap API allows clients to use common CRUD (create, read, update and delete) operations to manage key-value pairs of objects in the distributed data grid.

When using the ObjectMap API, object relationships must be expressed by embedding the foreign key for all relationships in the parent object.

An example follows.

```
public class Department {  
    Collection<String> employeeIds;  
}
```

The EntityManager API simplifies relationship management by extracting the persistent data from the objects including the foreign keys. When the object is later retrieved from the data grid, the relationship graph is rebuilt, as in the following example.

```
@Entity  
public class Department {  
    Collection<String> employees;  
}
```

The EntityManager API is very similar to other Java object persistence technologies such as JPA and Hibernate in that it synchronizes a graph of managed Java object instances with the persistent store. In this case, the persistent store is an eXtreme Scale data grid, where each entity is represented as a map and the map contains the entity data rather than the object instances.

Defining an entity schema

An ObjectGrid can have any number of logical entity schemas. Entities are defined using annotated Java classes, XML, or a combination of both XML and Java classes. Defined entities are then registered with an eXtreme Scale server and bound to BackingMaps, indexes and other plug-ins.

When designing an entity schema, you must complete the following tasks:

1. Define the entities and their relationships.
2. Configure eXtreme Scale.
3. Register the entities.
4. Create entity-based applications that interact with the eXtreme Scale EntityManager APIs.

Entity schema configuration

An entity schema is a set of entities and the relationships between the entities. In an eXtreme Scale application with multiple partitions, the following restrictions and options apply to entity schemas:

- Each entity schema must have a single root defined. This is known as the schema root.
- All the entities for a given schema must be in the same map set, which means that all the entities that are reachable from a schema root with key or non-key relationships must be defined in the same map set as the schema root.
- Each entity can belong to only one entity schema.
- Each eXtreme Scale application can have multiple schemas.

Entities are registered with an ObjectGrid instance before it is initialized. Each defined entity must be uniquely named and is automatically bound to an ObjectGrid BackingMap of the same name. The initialization method varies depending on the configuration you are using:

Local eXtreme Scale configuration

If you are using a local ObjectGrid, you can programmatically configure the entity schema. In this mode, you can use the ObjectGrid.registerEntities methods to register annotated entity classes or an entity metadata descriptor file.

Distributed eXtreme Scale configuration

If you are using a distributed eXtreme Scale configuration, you must provide an entity metadata descriptor file with the entity schema.

For more details, see “Entity manager in a distributed environment” on page 173.

Entity requirements

Entity metadata is configured using Java class files, an entity descriptor XML file or both. At minimum, the entity descriptor XML is required to identify which eXtreme Scale BackingMaps are to be associated with entities. The persistent attributes of the entity and its relationships to other entities are described in either an annotated Java class (entity metadata class) or the entity descriptor XML file. The entity metadata class, when specified, is also used by the EntityManager API to interact with the data in the grid.

An eXtreme Scale grid can be defined without providing any entity classes. This can be beneficial when the server and client are interacting directly with the tuple data stored in the underlying maps. Such entities are defined completely in the entity descriptor XML file and are referred to as classless entities.

Classless entities

Classless entities are useful when it is not possible to include application classes in the server or client classpath. Such entities are defined in the entity metadata descriptor XML file, where the class name of the entity is specified using a classless entity identifier in the form: @<entity identifier>. The @ symbol identifies the entity as classless and is used for mapping associations between entities. See the "Classless entity metadata" figure an example of an entity metadata descriptor XML file with two classless entities defined.

If an eXtreme Scale server or client does not have access to the classes, either can still use the EntityManager API using classless entities. Common use cases include the following:

- The eXtreme Scale container is hosted in a server that does not allow application classes in the classpath. In this case, the clients can still access the grid using the EntityManager API from a client, where the classes are allowed.
- The eXtreme Scale client does not require access to the entity classes because the client is either using a non-Java client, such as the eXtreme Scale REST data service or the client is accessing the tuple data in the grid using the ObjectMap API.

If the entity metadata is compatible between the client and server, entity metadata can be created using entity metadata classes, an XML file, or both.

For example, the "Programmatic entity class" in the following figure is compatible with the classless metadata code in the next section.

Programmatic entity class

```
@Entity
public class Employee {
    @Id long serialNumber;
    @Basic byte[] picture;
    @Version int ver;
    @ManyToOne(fetch=FetchType.EAGER, cascade=CascadeType.PERSIST)
    Department department;
}
```

```

@Entity
public static class Department {
    @Id int number;
    @Basic String name;
    @OneToMany(fetch=FetchType.LAZY, cascade=CascadeType.ALL, mappedBy="department")
    Collection<Employee> employees;
}

```

Classless fields, keys, and versions

As previously mentioned, classless entities are configured completely in the entity XML descriptor file. Class-based entities define their attributes using Java fields, properties and annotations. So classless entities need to define key and attribute structure in the entity XML descriptor with the `<basic>` and `<id>` tags.

Classless entity metadata

```

<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">

<entity class-name="@Employee" name="Employee">
    <attributes>
        <id name="serialNumber" type="long"/>
        <basic name="firstName" type="java.lang.String"/>
        <basic name="picture" type="[B]/>
        <version name="ver" type="int"/>
        <many-to-one
            name="department"
            target-entity="@Department"
            fetch="EAGER">
            <cascade><cascade-persist/></cascade>
        </many-to-one>
    </attributes>
</entity>

<entity class-name="@Department" name="Department" >
    <attributes>
        <id name="number" type="int"/>
        <basic name="name" type="java.lang.String"/>
        <version name="ver" type="int"/>
        <one-to-many
            name="employees"
            target-entity="@Employee"
            fetch="LAZY"
            mapped-by="department">
            <cascade><cascade-all/></cascade>
        </one-to-many>
    </attributes>
</entity>

```

Note that each entity above has an `<id>` element. A classless entity must have either one or more of an `<id>` element defined, or a single-valued association that represents the key for the entity. The fields of the entity are represented by `<basic>` elements. The `<id>`, `<version>`, and `<basic>` elements require a name and type in classless entities. See the following supported attribute types section for details on supported types.

Entity class requirements

Class-based entities are identified by associating various metadata with a Java class. The metadata can be specified using Java Platform, Standard Edition 5

annotations, an entity metadata descriptor file, or a combination of annotations and the descriptor file. Entity classes must meet the following criteria:

- The `@Entity` annotation is specified in the entity XML descriptor file.
- The class has a public or protected no-argument constructor.
- It must be a top-level class. Interfaces and enumerated types are not valid entity classes.
- Cannot use the final keyword.
- Cannot use inheritance.
- Must have a unique name and type for each ObjectGrid instance.

Entities all have a unique name and type. The name, if using annotations, is the simple (short) name of the class by default, but can be overridden using the name attribute of the `@Entity` annotation.

Persistent attributes

The persistent state of an entity is accessed by clients and the entity manager by using either fields (instance variables) or Enterprise JavaBeans-style property accessors. Each entity must define either field- or property-based access. Annotated entities are field-access if the class fields are annotated and are property-access if the getter method of the property is annotated. A mixture of field- and property-access is not allowed. If the type cannot be automatically determined, the **accessType** attribute on the `@Entity` annotation or equivalent XML can be used to identify the access type.

Persistent fields

Field-access entity instance variables are accessed directly from the entity manager and clients. Fields that are marked with the transient modifier or transient annotation are ignored. Persistent fields must not have final or static modifiers.

Persistent properties

Property-access entities must adhere to the JavaBeans signature conventions for read and write properties. Methods that do not follow JavaBeans conventions or have the Transient annotation on the getter method are ignored. For a property of type T, there must be a getter method `getProperty` which returns a value of type T and a void setter method `setProperty(T)`. For boolean types, the getter method can be expressed as `isProperty`, returning true or false. Persistent properties cannot have the static modifier.

Supported attribute types

The following persistent field and property types are supported:

- Java primitive types including wrappers
- `java.lang.String`
- `java.math.BigInteger`
- `java.math.BigDecimal`
- `java.util.Date`
- `java.util.Calendar`
- `java.sql.Date`
- `java.sql.Time`
- `java.sql.Timestamp`
- `byte[]`

- java.lang.Byte[]
- char[]
- java.lang.Character[]
- enum

User serializable attribute types are supported but have performance, query and change-detection limitations. Persistent data that cannot be proxied, such as arrays and user serializable objects, must be reassigned to the entity if altered.

Serializable attributes are represented in the entity descriptor XML file using the class name of the object. If the object is an array, the data type is represented using the Java internal form. For example, if an attribute data type is java.lang.Byte[[[]], the string representation is [[Ljava.lang.Byte;

User serializable types should adhere to the following best practices:

- Implement high performance serialization methods. Implement the java.lang.Cloneable interface and public clone method.
- Implement the java.io.Externalizable interface.
- Implement equals and hashCode

Entity associations

Bi-directional and uni-directional entity associations, or relationships between entities can be defined as one-to-one, many-to-one, one-to-many and many-to-many. The entity manager automatically resolves the entity relationships to the appropriate key references when storing the entities.

The eXtreme Scale grid is a data cache and does not enforce referential integrity like a database. Although relationships allow cascading persist and remove operations for child entities, it does not detect or enforce broken links to objects. When removing a child object, the reference to that object must be removed from the parent.

If you define a bi-directional association between two entities, you must identify the owner of the relationship. In a to-many association, the many side of the relationship is always the owning side. If ownership cannot be determined automatically, then the **mappedBy** attribute of the annotation, or XML equivalent, must be specified. The **mappedBy** attribute identifies the field in the target entity that is the owner of the relationship. This attribute also helps identify the related fields when there are multiple attributes of the same type and cardinality.

Single-valued associations

One-to-one and many-to-one associations are denoted using the @OneToOne and @ManyToOne annotations or equivalent XML attributes. The target entity type is determined by the attribute type. The following example defines a uni-directional association between Person and Address. The Customer entity has a reference to one Address entity. In this case, the association could also be many-to-one since there is no inverse relationship.

```
@Entity
public class Customer {
    @Id id;
    @OneToOne Address homeAddress;
}
```

```

@Entity
public class Address{
    @Id id
    @Basic String city;
}

```

To specify a bi-directional relationship between the Customer and Address classes, add a reference to the Customer class from the Address class and add the appropriate annotation to mark the inverse side of the relationship. Because this association is one-to-one, you have to specify an owner of the relationship using the mappedBy attribute on the @OneToOne annotation.

```

@Entity
public class Address{
    @Id id
    @Basic String city;
    @OneToOne(mappedBy="homeAddress") Customer customer;
}

```

Collection-valued associations

One-to-many and many-to-many associations are denoted using the @OneToMany and @ManyToMany annotations or equivalent XML attributes. All many relationships are represented using the types: java.util.Collection, java.util.List or java.util.Set. The target entity type is determined by the generic type of the Collection, List or Set or explicitly using the **targetEntity** attribute on the @OneToMany or @ManyToMany annotation (or XML equivalent).

In the previous example, it is not practical to have one address object per customer because many customers might share an address or might have multiple addresses. This situation is better solved using a many association:

```

@Entity
public class Customer {
    @Id id;
    @ManyToOne Address homeAddress;
    @ManyToOne Address workAddress;
}

@Entity
public class Address{
    @Id id
    @Basic String city;
    @OneToMany(mappedBy="homeAddress") Collection<Customer> homeCustomers;

    @OneToMany(mappedBy="workAddress", targetEntity=Customer.class)
    Collection workCustomers;
}

```

In this example, two different relationships exist between the same entities: a Home and Work address relationship. A non-generic Collection is used for the **workCustomers** attribute to demonstrate how to use the **targetEntity** attribute when generics are not available.

Classless associations

Classless entity associations are defined in the entity metadata descriptor XML file similar to how class-based associations are defined. The only difference is that instead of the target entity pointing to an actual class, it points to the classless entity identifier used for the class name of the entity.

An example follows:

```
<many-to-one name="department" target-entity="@Department" fetch="EAGER">
  <cascade><cascade-all/></cascade>
</many-to-one>
<one-to-many name="employees" target-entity="@Employee" fetch="LAZY">
  <cascade><cascade-all/></cascade>
</one-to-many>
```

Primary keys

All entities must have a primary key, which can be a simple (single attribute) or composite (multiple attribute) key. The key attributes are denoted using the `Id` annotation or defined in the entity XML descriptor file. Key attributes have the following requirements:

- The value of a primary key cannot change.
- A primary key attribute should be one of the following types: Java primitive type and wrappers, `java.lang.String`, `java.util.Date` or `java.sql.Date`.
- A primary key can contain any number of single-valued associations. The target entity of the primary key association must not have an inverse association directly or indirectly to the source entity.

Composite primary keys can optionally define a primary key class. An entity is associated with a primary key class using the `@IdClass` annotation or the entity XML descriptor file. An `@IdClass` annotation is useful in conjunction with the `EntityManager.find` method.

Primary key classes have the following requirements:

- It should be public with a no-argument constructor.
- The access type of the primary key class is determined by the entity that declares the primary key class.
- If property-access, the properties of the primary key class must be public or protected.
- The primary key fields or properties must match the key attribute names and types defined in the referencing entity.
- Primary key classes must implement the `equals` and `hashCode` methods.

An example follows:

```
@Entity
@IdClass(CustomerKey.class)
public class Customer {
    @Id @ManyToOne Zone zone;
    @Id int custId;
    String name;
    ...
}

@Entity
public class Zone{
    @Id String zoneCode;
    String name;
}

public class CustomerKey {
    Zone zone;
    int custId;
}
```

```

    public int hashCode() {...}
    public boolean equals(Object o) {...}
}

```

Classless primary keys

Classless entities are required to either have at least one <id> element or an association in the XML file with the attribute id=true. An example of both would look like the following:

```

<id name="serialNumber" type="int"/>
<many-to-one name="department" target-entity="@Department" id="true">
<cascade><cascade-all/></cascade>
</many-to-one>

```

Remember:

The <id-class> XML tag is not supported for classless entities.

Entity proxies and field interception

Entity classes and mutable supported attribute types are extended by proxy classes for property-access entities and bytecode-enhanced for Java Development Kit (JDK) 5 field-access entities. All access to the entity, even by internal business methods and the equals methods, must use the appropriate field or property access methods.

Proxies and field interceptors are used to allow the entity manager to track the state of the entity, determine if the entity has changed, and improve performance. Field interceptors are only available on Java SE 5 platforms when the entity instrumentation agent is configured.

Attention: When using property-access entities, the equals method should use the instanceof operator for comparing the current instance to the input object. All introspection of the target object should be through the properties of the object, not the fields themselves, because the object instance will be the proxy.

Entity manager in a distributed environment

You can use EntityManager API with a local ObjectGrid or in a distributed eXtreme Scale environment. The main difference is how you connect to this remote environment. After you establish a connection, there is no difference between using a Session object or using the EntityManager API.

Required configuration files

The following XML configuration files are required:

- ObjectGrid descriptor XML file
- Entity descriptor XML file
- Deployment or data grid descriptor XML file

These files specify the entities and BackingMaps that a server hosts.

The entity metadata descriptor file contains a description of the entities that are used. At minimum, you must specify the entity class and name. If you are running in a Java Platform, Standard Edition 5 environment, eXtreme Scale automatically reads the entity class and its annotations. You can define additional XML attributes

if the entity class has no annotations or if you are required to override the class attributes. If you are registering the entities classless , provide all of entity information in the XML file only.

You can use the following XML configuration snippet to define a data grid with entities. In this snippet, the server creates an ObjectGrid with the name bookstore and an associated backing map with the name order. The objectgrid.xml file snippet refers to the entity.xml file. In this case, the entity.xml file contains one entity, the Order entity.

objectgrid.xml

```
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">

  <objectGrids>
    <objectGrid name="bookstore" entityMetadataXMLFile="entity.xml">
      <backingMap name="Order"/>
    </objectGrid>
  </objectGrids>

</objectGridConfig>
```

This objectgrid.xml file specifies the entity.xml file with the **entityMetadataXMLFile** attribute. The value can be a relative directory or an absolute path.

- **For a relative directory:** Specify the location relative to the location of the objectgrid.xml file.
- **For an absolute path:** Specify the location with a URL format, such as file:// or http://.

An example of the entity.xml file follows:

entity.xml

```
<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ../emd.xsd">
  <entity class-name="com.ibm.websphere.tutorials.objectgrid.em.
    distributed.step1.Order" name="Order"/>
</entity-mappings>
```

This example assumes that the Order class would have the **orderNumber** and **desc** fields annotated similarly.

An equivalent classless entity.xml file would be as follows:

```
classless entity.xml
<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ../emd.xsd">
  <entity class-name="@Order" name="Order">
    <description>Entity named: Order</description>
    <attributes>
      <id name="orderNumber" type="int"/>
      <basic name="desc" type="java.lang.String"/>
    </attributes>
  </entity>
</entity-mappings>
```

For information about starting servers, see the *Administration Guide*. You use both the deployment.xml and objectgrid.xml files to start the catalog server.

Connecting to a distributed eXtreme Scale server

The following code enables the connect mechanism for a client and server on the same computer:


```
String catalogEndpoints="localhost:2809";
URL clientOverrideURL= new URL("file:etc/emtutorial/distributed/step1/objectgrid.xml");
ClientClusterContext clusterCtx = ogMgr.connect(catalogEndpoints, null, clientOverrideURL);
ObjectGrid objectGrid=ogMgr.getObjectGrid(clusterCtx, "bookstore");
```

In the preceding code snippet, note the reference to the remote eXtreme Scale server. After you establish a connection, you can invoke EntityManager API methods such as persist, update, remove and find.

Attention: When you are using entities, pass the client override ObjectGrid descriptor XML file to the connect method. If a null value is passed to the clientOverrideURL property and the client has a different directory structure than the server, then the client might fail to locate the ObjectGrid or entity descriptor XML files. At minimum, the ObjectGrid and entity XML files for the server can be copied to the client.

Previously, using entities on an ObjectGrid client required you to make the ObjectGrid XML and entity XML available to the client in one of the following two ways:

1. Pass an overriding ObjectGrid XML to the ObjectGridManager.connect(String catalogServerEndpoints, ClientSecurityConfiguration securityProps, URL overRideObjectGridXml) method.

```
String catalogEndpoints="myHost:2809";
URL clientOverrideURL= new URL("file:etc/emtutorial/distributed/step1/objectgrid.xml");
ClientClusterContext clusterCtx = ogMgr.connect(catalogEndpoints, null, clientOverrideURL);
ObjectGrid objectGrid=ogMgr.getObjectGrid(clusterCtx, "bookstore");
```

2. Pass null for the override file and ensure that the ObjectGrid XML and referenced entity XML are available to the client on the same path as on the server.

```
String catalogEndpoints="myHost:2809";
ClientClusterContext clusterCtx = ogMgr.connect(catalogEndpoints, null, null);
ObjectGrid objectGrid=ogMgr.getObjectGrid(clusterCtx, "bookstore");
```

The XML files were required regardless of whether or not you wanted to use subset entities on the client side. These files are no longer required to use the entities as defined by the server. Instead, pass null as the overRideObjectGridXml parameter as in option 2 of the previous section. If the XML file is not found on the same path set on the server, the client uses the entity configuration on the server.

However, if you use subset entities on the client, you must provide an overriding ObjectGrid XML as in option 1.

Client and server-side schema

The server-side schema defines the type of data stored in the maps on a server. The client-side schema is a mapping to application objects from the schema on the server. For example, you might have the following server-side schema:

```
@Entity
class ServerPerson
{
    @Id String ssn;
    String firstName;
    String surname;
    int age;
    int salary;
}
```

A client can have an object annotated as in the following example:

```

@Entity(name="ServerPerson")
class ClientPerson
{
    @Id @Basic(alias="ssn") String socialSecurityNumber;
    String surname;
}

```

This client then takes a server-side entity and projects the subset of the entity into the client object. This projection leads to bandwidth and memory savings on a client because the client has only the information it needs instead of all of the information that is in the server-side entity. Different applications can use their own objects instead of forcing all applications to share a set of classes for data access.

The client-side entity descriptor XML file is required in the following cases: if the server is running with class-based entities while the client side is running classless; or if the server is classless and the client uses class-based entities. A classless client mode allows the client to still run entity queries without having access to the physical classes. Assuming the server has registered the `ServerPerson` entity above, the client would override the data grid with an `entity.xml` file such as follows:

```

<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">
<entity class-name="@ServerPerson" name="Order">
    <description>Entity named: Order</description>
    <attributes>
        <id name="socialSecurityNumber" type="java.lang.String"/>
        <basic name="surname" type="java.lang.String"/>
    </attributes>
</entity>
</entity-mappings>

```

This file achieves an equivalent subset entity on the client, without requiring the client to provide the actual annotated class. If the server is classless, and the client is not classless, then the client provides an overriding entity descriptor XML file. This entity descriptor XML file contains an override to the class file reference.

Referencing the schema

If your application is running in Java SE 5, then the application can be added to the objects by using annotations. The `EntityManager` can read the schema from the annotations on those objects. The application provides the eXtreme Scale run time with references to these objects using the `entity.xml` file, which is referenced from the `objectgrid.xml` file. The `entity.xml` file lists all the entities, each of which is associated with either a class or a schema. If a proper class name is specified, then the application attempts to read the Java SE 5 annotations from those classes to determine the schema. If you do not annotate the class file or specify a classless identifier as the class name, then the schema is taken from the XML file. The XML file is used to specify all the attributes, keys, and relationships for each entity.

A local data grid does not need XML files. The program can obtain an `ObjectGrid` reference and invoke the `ObjectGrid.registerEntities` method to specify a list of Java SE 5 annotated classes or an XML file.

The run time uses the XML file or a list of annotated classes to find entity names, attribute names and types, key fields and types, and relationships between entities. If eXtreme Scale is running on a server or in stand-alone mode, then it automatically makes a map named after each entity. These maps can be customized further using the `objectgrid.xml` file or APIs set either by the application or injection frameworks such as Spring.

Entity metadata descriptor file

See `emd.xsd` file for more information about the metadata descriptor file.

Interacting with EntityManager

Applications typically first obtain an `ObjectGrid` reference, and then a `Session` from that reference for each thread. Sessions cannot be shared between threads. An extra method on `Session`, the `getEntityManager` method, is available. This method returns a reference to an entity manager to use for this thread. The `EntityManager` interface can replace the `Session` and `ObjectMap` interfaces for all applications. You can use these `EntityManager` APIs if the client has access to the defined entity classes.

Obtaining an EntityManager instance from a session

The `getEntityManager` method is available on a `Session` object. The following code example illustrates how to create a local `ObjectGrid` instance and access the `EntityManager`. See the `EntityManager` interface in the API documentation for details about all the supported methods.

```
ObjectGrid og =  
ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("intro-grid");  
Session s = og.getSession();  
EntityManager em = s.getEntityManager();
```

A one-to-one relationship exists between the `Session` object and `EntityManager` object. You can use the `EntityManager` object more than once.

Persisting an entity

Persisting an entity means saving the state of a new entity in an `ObjectGrid` cache. After the `persist` method is called, the entity is in the managed state. `Persist` is a transactional operation, and the new entity is stored in the `ObjectGrid` cache after the transaction commits.

Every entity has a corresponding `BackingMap` in which the tuples are stored. The `BackingMap` has the same name as the entity, and is created when the class is registered. The following code example demonstrates how to create an `Order` object by using the `persist` operation.

```
Order order = new Order(123);  
em.persist(order);  
order.setX();  
...
```

The `Order` object is created with the key 123, and the object is passed to the `persist` method. You can continue to modify the state of the object before you commit the transaction.

Important: The preceding example does not include any required transactional boundaries, such as `begin` and `commit`. See the “Tutorial: Storing order information in entities” on page 9 the entity manager tutorial in the *Product Overview* for more information.

Finding an entity

You can locate the entity in the `ObjectGrid` cache with the `find` method by providing a key after the entity is stored in the cache. This method does not

require any transactional boundary, which is useful for read-only semantics. The following example illustrates that only one line of code is needed to locate the entity.

```
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
```

Removing an entity

The remove method, like the persist method, is a transactional operation. The following example shows the transactional boundary by calling the begin and commit methods.

```
em.getTransaction().begin();
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
em.remove(foundOrder );
em.getTransaction().commit();
```

The entity must first be managed before it can be removed, which you can accomplish by calling the find method within the transactional boundary. Then call the remove method on the EntityManager interface.

Invalidating an entity

The invalidate method behaves much like the remove method, but does not invoke any Loader plug-ins. Use this method to remove entities from the ObjectGrid, but to preserve them in the backend data store.

```
em.getTransaction().begin();
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
em.invalidate(foundOrder );
em.getTransaction().commit();
```

The entity must first be managed before it can be invalidated, which you can accomplish by calling the find method within the transactional boundary. After you call the find method, you can call the invalidate method on the EntityManager interface.

Updating an entity

The update method is also a transactional operation. The entity must be managed before any updates can be applied.

```
em.getTransaction().begin();
Order foundOrder = (Order)em.find(Order.class, new Integer(123));
foundOrder.date = new Date(); // update the date of the order
em.getTransaction().commit();
```

In the preceding example, the persist method is not called after the entity is updated. The entity is updated in the ObjectGrid cache when the transaction is committed.

Queries and query queues

With the flexible query engine, you can retrieve entities by using EntityManager API. Create SELECT type queries over an entity or Object-based schema by using the ObjectGrid query language. Query interface explains in detail how you can run the queries by using the EntityManager API. See the Query API for more information about using queries.

An entity QueryQueue is a queue-like data structure associated with an entity query. It selects all the entities that match the WHERE condition on the query filter

and puts the result entities in a queue. Clients can then iteratively retrieve entities from this queue. See “Entity query queues” on page 186 for more information.

Entity listeners and callback methods:

Applications can be notified when the state of an entity transitions from state to state. Two callback mechanisms exist for state change events: life cycle callback methods that are defined on an entity class and are invoked whenever the entity state changes, and entity listeners, which are more general because the entity listener can be registered on several entities.

Life cycle of an entity instance

An entity instance has the following states:

- **New:** A newly created entity instance that does not exist in the eXtreme Scale cache.
- **Managed:** The entity instance exists in the eXtreme Scale cache and is retrieved or persisted using the entity manager. An entity must be associated with an active transaction to be in the managed state.
- **Detached:** The entity instance exists in the eXtreme Scale cache, but is no longer associated with an active transaction.
- **Removed:** The entity instance is removed, or is scheduled to be removed, from the eXtreme Scale cache when the transaction is flushed or committed.
- **Invalidated:** The entity instance is invalidated, or is scheduled to be invalidated, from the eXtreme Scale cache when the transaction is flushed or committed.

When entities change from state to state, you can invoke life-cycle, call-back methods.

The following sections further describe the meanings of New, Managed, Detached, Removed and Invalidated states as the states apply to an entity.

Entity life cycle callback methods

Entity life cycle callback methods can be defined on the entity class and are invoked when the entity state changes. These methods are useful for validating entity fields and updating transient state that is not usually persisted with the entity. Entity life cycle callback methods can also be defined on classes that are not using entities. Such classes are entity listener classes, which can be associated with multiple entity types. life cycle callback methods can be defined using both metadata annotations and a entity metadata XML descriptor file:

- **Annotations:** life cycle callback methods can be denoted using the PrePersist, PostPersist, PreRemove, PostRemove, PreUpdate, PostUpdate, and PostLoad annotations in an entity class.
- **Entity XML descriptor :** life cycle callback methods can be described using XML when annotations are not available.

Entity listeners

An entity listener class is a class that does not use entities that defines one or more entity life cycle callback methods. Entity listeners are useful for general purpose auditing or logging applications. Entity listeners can be defined using both metadata annotations and a entity metadata XML descriptor file:

- **Annotation:** The EntityListeners annotation can be used to denote one or more entity listener classes on an entity class. If multiple entity listeners are defined, the order in which they are invoked is determined by the order in which they are specified in the EntityListeners annotation.
- **Entity XML descriptor:** The XML descriptor can be used as an alternative to specify the invocation order of entity listeners or to override the order that is specified in metadata annotations.

Callback method requirements

Any subset or combination of annotations can be specified on an entity class or a listener class. A single class cannot have more than one life cycle callback method for the same life cycle event. However, the same method can be used for multiple callback events. The entity listener class must have a public no-arg constructor. Entity listeners are stateless. The life cycle of an entity listener is unspecified. eXtreme Scale does not support entity inheritance, so callback methods can only be defined in the entity class, but not in the superclass.

Callback method signature

Entity life cycle callback methods can be defined on an entity listener class, directly on an entity class, or both. Entity life cycle callback methods can be defined using both metadata annotations and the entity XML descriptor. The annotations used for callback methods on the entity class and on the entity listener class are the same. The signatures of the callback methods are different when defined on an entity class versus an entity listener class. Callback methods defined on an entity class or mapped superclass have the following signature:

```
void <METHOD>()
```

Callback methods that are defined on an entity listener class have the following signature:

```
void <METHOD>(Object)
```

The Object argument is the entity instance for which the callback method is invoked. The Object argument can be declared as a java.lang.Object object or the actual entity type.

Callback methods can have public, private, protected, or package level access, but must not be static or final.

The following annotations are defined to designate life cycle event callback methods of the corresponding types:

- com.ibm.websphere.projector.annotations.PrePersist
- com.ibm.websphere.projector.annotations.PostPersist
- com.ibm.websphere.projector.annotations.PreRemove
- com.ibm.websphere.projector.annotations.PostRemove
- com.ibm.websphere.projector.annotations.PreUpdate
- com.ibm.websphere.projector.annotations.PostUpdate
- com.ibm.websphere.projector.annotations.PostLoad

See the API Documentation for more details. Each annotation has an equivalent XML attribute defined in the entity metadata XML descriptor file.

Life cycle callback method semantics

Each of the different life cycle callback methods has a different purpose and is called in different phases of the entity life cycle:

PrePersist

Invoked for an entity before the entity has been persisted to the store, which includes entities that have been persisted due to a cascading operation. This method is invoked on the thread of the EntityManager.persist operation.

PostPersist

Invoked for an entity after the entity has been persisted to the store, which includes entities that have been persisted due to a cascading operation. This method is invoked on the thread of the EntityManager.persist operation. It is called after the EntityManager.flush or EntityManager.commit is called.

PreRemove

Invoked for an entity before the entity has been removed, which includes entities that have been removed due to a cascading operation. This method is invoked on the thread of the EntityManager.remove operation.

PostRemove

Invoked for an entity after the entity has been removed, which includes entities that have been removed due to a cascading operation. This method is invoked on the thread of the EntityManager.remove operation. It is called after the EntityManager.flush or EntityManager.commit is called.

PreUpdate

Invoked for an entity before the entity has been updated to the store. This method is invoked on the thread of the transaction flush or commit operation.

PostUpdate

Invoked for an entity after the entity has been updated to the store. This method is invoked on the thread of the transaction flush or commit operation.

PostLoad

Invoked for an entity after the entity has been loaded from the store which includes any entities that are loaded through an association. This method is invoked on the thread of the loading operation, such as EntityManager.find or a query.

Duplicate life cycle callback methods

If multiple callback methods are defined for an entity life cycle event, the ordering of the invocation of these methods is as follows:

1. **life cycle callback methods defined in the entity listeners:** The life cycle callback methods that are defined on the entity listener classes for an entity class are invoked in the same order as the specification of the entity listener classes in the EntityListeners annotation or the XML descriptor.
2. **Listener super class:** Callback methods defined in the super class of the entity listener are invoked before the children.
3. **Entity life cycle methods:** WebSphere eXtreme Scale does not support entity inheritance, so the entity life cycle methods can only be defined in the entity class.

Exceptions

Life cycle callback methods might result in run time exceptions. If a life cycle callback method results in a run time exception within a transaction, the transaction is rolled back. No further life cycle callback methods are invoked after a runtime exception results.

Entity listener examples:

You can write EntityListeners based on your requirements. Several example scripts follow.

EntityListeners example using annotations

The following example shows the life-cycle callback method invocations and order of the invocations. Assume an entity class Employee and two entity listeners exist: EmployeeListener and EmployeeListener2.

```
@Entity
@EntityListeners({EmployeeListener.class, EmployeeListener2.class})
public class Employee {
    @PrePersist
    public void checkEmployeeID() {
        ....
    }
}

public class EmployeeListener {
    @PrePersist
    public void onEmployeePrePersist(Employee e) {
        ....
    }
}

public class PersonListener {
    @PrePersist
    public void onPersonPrePersist(Object person) {
        ....
    }
}

public class EmployeeListener2 extends PersonListener {
    @PrePersist
    public void onEmployeePrePersist2(Object employee) {
        ....
    }
}
```

If a PrePersist event occurs on an Employee instance, the following methods are called in order:

1. onEmployeePrePersist method
2. onPersonPrePersist method
3. onEmployeePrePersist2 method
4. checkEmployeeID method

Entity listeners example using XML

The following example shows how to set an entity listener on an entity using the entity descriptor XML file:


```

<entity
  class-name="com.ibm.websphere.objectgrid.sample.Employee"
  name="Employee" access="FIELD">
  <attributes>
    <id name="id" />
    <basic name="value" />
  </attributes>
  <entity-listeners>
    <entity-listener
      class-name="com.ibm.websphere.objectgrid.sample.EmployeeListener">
      <pre-persist method-name="onListenerPrePersist" />
      <post-persist method-name="onListenerPostPersist" />
    </entity-listener>
  </entity-listeners>
  <pre-persist method-name="checkEmployeeID" />
</entity>

```

The entity Employee is configured with a `com.ibm.websphere.objectgrid.sample.EmployeeListener` entity listener class, which has two life-cycle callback methods defined. The `onListenerPrePersist` method is for the PrePersist event, and the `onListenerPostPersist` method is for the PostPersist event. Also, the `checkEmployeeID` method in the Employee class is configured to listen for the PrePersist event.

EntityManager fetch plan support

A FetchPlan is the strategy that the entity manager uses for retrieving associated objects if the application needs to access relationships.

Example

Assume for example that your application has two entities: Department and Employee. The relationship between the Department entity and the Employee entity is a bi-directional one-to-many relationship: One department has many employees, and one employee belongs to only one department. Since most of the time, when Department entity is fetched, its employees are likely to be fetched, the fetch type of this one-to-many relationship is set to be EAGER.

Here is a snippet of the Department class.

```

@Entity
public class Department {

    @Id
    private String deptId;

    @Basic
    String deptName;

    @OneToMany(fetch = FetchType.EAGER, mappedBy="department", cascade = {CascadeType.PERSIST})
    public Collection<Employee> employees;

}

```

In a distributed environment, when an application calls `em.find(Department.class, "dept1")` to find a Department entity with key "dept1", this find operation will get the Department entity and all its eager-fetched relations. In the case of the preceding snippet, these are all the employees of department "dept1".

Prior to WebSphere eXtreme Scale 6.1.0.5, the retrieval of one Department entity and N Employee entities incurred N+1 client-server trips because the client retrieved one entity for one client-server trip. You can improve performance if you retrieve these N+1 entities in one trip.

Fetch plan

A fetch plan can be used to customize how to fetch eager relationships by customizing the maximum depth of the relationships. The fetch depth overrides eager relations greater than the specified depth to lazy relations. By default, the fetch depth is the maximum fetch depth. This means that eager relationships of all levels that are eager-navigable from the root entity will be fetched. An EAGER relationship is eager-navigable from a root entity if and only if all the relations starting from the root entity to it are configured as eager-fetched.

In the previous example, the Employee entity is eager-navigable from the Department entity because the Department-Employee relationship is configured as eager-fetched.

If the Employee entity has another eager relationship to an Address entity for instance, then the Address entity is also eager-navigable from the Department entity. However, if the Department-Employee relationships were configured as lazy-fetch, then the Address entity is not eager-navigable from the Department entity because the Department-Employee relationship breaks the eager fetch chain.

A FetchPlan object can be retrieved from the EntityManager instance. The application can use the setMaxFetchDepth method to change the maximum fetch depth.

A fetch plan is associated with an EntityManager instance. The fetch plan applies to any fetch operation, more specifically as follows.

- EntityManager find(Class class, Object key) and findForUpdate(Class class, Object key) operations
- Query operations
- QueryQueue operations

The FetchPlan object is mutable. Once changed, the changed value will be applied to the fetch operations executed afterward.

A fetch plan is important for a distributed deployment because it decides whether the eager-fetched relationship entities are retrieved with the root entity in one client-server trip or more than one.

Continuing with the previous example, consider further that the fetch plan has maximum depth set to infinity. In that case, when an application calls `em.find(Department.class, "dept1")` to find a Department, this find operation will get one Department entity and N employee entities in one client-server trip. However, for a fetch plan with maximum fetch depth set to zero, only the Department object will be retrieved from the server, while the Employee entities are retrieved from the server only when the employees collection of the Department object is accessed.

Different fetch plans

You have several different fetch plans based on your requirements, explained in the following sections.

Impact on a distributed grid

- *Infinite-depth fetch plan*: An infinite-depth fetch plan has its maximum fetch depth set to `FetchPlan.DEPTH_INFINITE`.

In a client-server environment, if an infinite-depth fetch plan is used, then all the relations that are eager-navigable from the root entity will be retrieved in one client-server trip.

Example: If the application is interested in all the Address entities of all employees of a particular Department, then it uses an infinite-depth fetch plan to retrieve all the associated Address entities. The following code only incurs one client-server trip.

```
em.getFetchPlan().setMaxFetchDepth(FetchPlan.DEPTH_INFINITE);

tran.begin();
Department dept = (Department) em.find(Department.class, "dept1");
// do something with Address object.
for (Employee e: dept.employees) {
    for (Address addr: e.addresses) {
        // do something with addresses.
    }
}
tran.commit();
```

- *Zero-depth fetch plan:* A zero-depth fetch plan has its maximum fetch depth set to 0.

In a client-server environment, if a zero fetch plan is used, then only the root entity will be retrieved in the first client-server trip. All the eager relationships are treated as if they were lazy.

Example: In this example, the application is only interested in the Department entity attribute. It does not need to access its employees, so the application sets the fetch plan depth to 0.

```
Session session = objectGrid.getSession();
EntityManager em = session.getEntityManager();
EntityTransaction tran = em.getTransaction();
em.getFetchPlan().setMaxFetchDepth(0);

tran.begin();
Department dept = (Department) em.find(Department.class, "dept1");
// do something with dept object.
tran.commit();
```

- *Fetch plan with depth k :*

A k -depth fetch plan has its maximum fetch depth set to k .

In a client-server eXtreme Scale environment, if a k -depth fetch plan is used, then all the relationships eager-navigable from the root entity within k steps will be retrieved in the first client-server trip.

The infinite-depth fetch plan ($k = \text{infinity}$) and zero-depth fetch plan ($k = 0$) are just two examples of the k -depth fetch plan.

To continue expanding on the previous example, assume there is another eager relationship from the entity Employee to the entity Address. If the fetch plan has maximum fetch depth set to 1, then the `em.find(Department.class, "dept1")` operation will retrieve the Department entity and all its Employee entities in one client-server trip. However, the Address entities will not be retrieved because they are not eager-navigable to the Department entity within 1 step, but 2 steps.

If you use a fetch plan with depth set to 2, then the `em.find(Department.class, "dept1")` operation will retrieve the Department entity, all its Employee entities, and all Address entities associated with the Employee entities in one client-server trip.

Tip: The default fetch plan has maximum fetch depth set to infinity, so the default behavior of a fetch operation can change. All the eager-navigable relationships from the root entity are retrieved. Instead of multiple trips, now the fetch operation only incurs one client-server trip with the default fetch plan. To keep the settings for the product from the prior version, set the fetch depth to 0.

- *Fetch plan used on query:*

If you execute an entity query you can also use a fetch plan to customize relationship retrieval.

For example, the query `SELECT d FROM Department d WHERE "d.deptName='Department'"` result has a relationship to the Department entity. Notice the fetch plan depth starts with the query result association: In this case, the Department entity, not the query result itself. That is, the Department entity is on fetch-depth level 0. Therefore a fetch plan with maximum fetch depth 1 will retrieve the Department entity and its Employee entities in one client-server trip.

Example: In this example, the fetch plan depth is set to 1, so the Department entity and its Employee entities are retrieved in one client-server trip, but the Address entities will not be retrieved in the same trip.

Important: If a relationship is ordered, using either `OrderBy` annotation or configuration, then it is considered an eager relationship even if it is configured as lazy-fetch.

Performance considerations in a distributed environment

By default, all relationships that are eager-navigable from the root entity will be retrieved in one client-server trip. This can improve performance if all the relationships are going to be used. However, in certain usage scenarios, not all relationships eager-navigable from the root entity are used, so they incur both run-time overhead and bandwidth overhead by retrieving those unused entities.

For such cases, the application can set the maximum fetch depth to a small number to decrease the depth of entities to be retrieved by making all the eager relations after that certain depth lazy. This setting can improve performance.

Proceeding still further with the previous Department-Employee-Address example, by default, all the Address entities associated with employees of the Department "dept1" will be retrieved when `em.find(Department.class, "dept1")` is called. If the application does not use Address entities, it can set the maximum fetch depth to 1, so the Address entities will not be retrieved with the Department entity.

Entity query queues

Query queues allow applications to create a queue qualified by a query in the server-side or local eXtreme Scale over an entity. Entities from the query result are stored in this queue. Currently, query queue is only supported in a map that is using the pessimistic lock strategy.

A query queue is shared by multiple transactions and clients. After the query queue becomes empty, the entity query associated with this queue is rerun and new results are added to the queue. A query queue is uniquely identified by the entity query string and parameters. There is only one instance for each unique query queue in one ObjectGrid instance. See the EntityManager API documentation for additional information.

Query queue example

The following example shows how query queue can be used.

```
/**
 * Get a unassigned question type task
 */
private void getUnassignedQuestionTask() throws Exception {
```

```

EntityManager em = og.getSession().getEntityManager();
EntityTransaction tran = em.getTransaction();

QueryQueue queue = em.createQueryQueue("SELECT t FROM Task t
WHERE t.type=?1 AND t.status=?2", Task.class);
queue.setParameter(1, new Integer(Task.TYPE_QUESTION));
queue.setParameter(2, new Integer(Task.STATUS_UNASSIGNED));

tran.begin();
Task nextTask = (Task) queue.getNextEntity(10000);
System.out.println("next task is " + nextTask);
if (nextTask != null) {
    assignTask(em, nextTask);
}
tran.commit();
}

```

The previous example first creates a QueryQueue with a entity query string, "SELECT t FROM Task t WHERE t.type=?1 AND t.status=?2". Then it sets the parameters for the QueryQueue object. This query queue represents all "unassigned" tasks of the type "question". The QueryQueue object is very similar to an entity Query object.

After the QueryQueue is created, an entity transaction is started and the getNextEntity method is invoked, which retrieves the next available entity with a timeout value set to 10 seconds. After the entity is retrieved, it is processed in the assignTask method. The assignTask modifies the Task entity instance and changes the status to "assigned" which effectively removes it from the queue since it no longer matches the QueryQueue's filter. Once assigned, the transaction is committed.

From this simple example, you can see a query queue is similar to an entity query. However, they differ in the following ways:

1. Entities in the query queue can be retrieved in an iterative manner. The user application decides the number of entities to be retrieved. For example, if QueryQueue.getNextEntity(timeout) is used, only one entity is retrieved, and if QueryQueue.getNextEntities(5, timeout) is used, 5 entities are retrieved. In a distributed environment, the number of entities directly decides the number of bytes to be transferred from the server to client.
2. When an entity is retrieved from the query queue, a U lock is placed on the entity so no other transactions can access it.

Retrieve entities in a loop

You can retrieve entities in a loop. An example that illustrates how to get all the unassigned, question type tasks completed follows.

```

/**
 * Get all unassigned question type tasks
 */
private void getAllUnassignedQuestionTask() throws Exception {
    EntityManager em = og.getSession().getEntityManager();
    EntityTransaction tran = em.getTransaction();

    QueryQueue queue = em.createQueryQueue("SELECT t FROM Task t WHERE
t.type=?1 AND t.status=?2", Task.class);
    queue.setParameter(1, new Integer(Task.TYPE_QUESTION));
    queue.setParameter(2, new Integer(Task.STATUS_UNASSIGNED));

    Task nextTask = null;

```

```

do {
    tran.begin();
    nextTask = (Task) queue.getNextEntity(10000);
    if (nextTask != null) {
        System.out.println("next task is " + nextTask);
    }
    tran.commit();
} while (nextTask != null);
}

```

If there are 10 unassigned question-type tasks in the entity map, you might expect that you will have 10 entities printed to the console. However, if this example is run, you will see the program never exits, which might be contrary to what you assumed.

When a query queue is created and the `getNextEntity` is called, the entity query associated with the queue is executed and the 10 results are populated into the queue. When `getNextEntity` is called, an entity is taken off the queue. After 10 `getNextEntity` calls are executed, the queue is empty. The entity query will automatically re-run. Since these 10 entities still exist and match the query queue's filter criteria, they are populated into the queue again.

If the following line is added after the `println()` statement, you will see only 10 entities printed.

```
em.remove(nextTask);
```

For information on using `SessionHandle` with `QueryQueue` in a per-container placement deployment, read about `SessionHandle` integration.

Query queues deployed to all partitions

In a distributed eXtreme Scale, a query queue can be created for one partition or all partitions. If a query queue is created for all partitions, there will be one query queue instance in each partition.

When a client tries to get the next entity using the `QueryQueue.getNextEntity` or `QueryQueue.getNextEntities` method, the client sends a request to one of the partitions. A client sends peek and pin requests to the server:

- With a peek request, the client sends a request to one partition and the server returns immediately. If there is an entity in the queue, the server sends a response with the entity; if there is not, the server sends a response with no entity. In either case, the server will return immediately.
- With a pin request, the client sends a request to one partition and the server waits until an entity is available. If there is an entity in the queue, the server sends a response with the entity immediately; if there is not, the server waits on the queue until either an entity is available or the request times out.

An example of how an entity is retrieved for a query queue which is deployed to all partitions (n) follows:

1. When a `QueryQueue.getNextEntity` or `QueryQueue.getNextEntities` method is called, the client picks a random partition number from 0 to n-1.
2. The client sends peek request to the random partition.
 - If an entity is available, the `QueryQueue.getNextEntity` or `QueryQueue.getNextEntities` method exits by returning the entity.

- If an entity is not available and is not the last unvisited partition, the client sends a peek request to the next partition.
- If an entity is not available and it is the last unvisited partition, the client instead sends a pin request.
- If the pin request to the last partition times-out and there is still no data available, the client will make a last effort by sending peek request to all partitions serially one more round. Therefore, if any entity is available in the previous partitions, the client will be able to get it.

Subset entity and no-entity support

The method to create a QueryQueue object in the entity manager follows:

```
public QueryQueue createQueryQueue(String qlString, Class entityClass);
```

The result in the query queue should be projected to the object defined by the second parameter to the method, Class entityClass.

If this parameter is specified, the class must have the same entity name as specified in the query string. This is useful if you want to project an entity into a subset entity. If a null value is used as the entity class, then the result will not be projected. The value stored in the map will be in an entity tuple format.

Client-side key collision

In distributed eXtreme Scale environment, query queue is only supported for eXtreme Scale maps with pessimistic locking mode. Therefore, there is no near cache on the client side. However, a client could have data (key and value) in the transactional map. This potentially could lead to a key collision when an entity retrieved from the server share the same key as an entry already in the transactional map.

When a key collision happens, the eXtreme Scale client run time uses the following rule to either throw an exception or silently override the data.

1. If the collided key is the key of the entity specified in the entity query associated with the query queue, then an exception is thrown. In this case, the transaction is rolled back, and the U lock on this entity key will be released on the server side.
2. Otherwise, if the collided key is the key of the entity association, the data in the transactional map will be overridden without warning.

The key collision only happens when there is a data in the transactional map. In other words, it only happens when a getNextEntity or getNextEntities call is called in a transaction which has already been dirtied (a new data has been inserted or a data has been updated). If an application does not want a key collision happen, it should always call getNextEntity or getNextEntities in a transaction which has not been dirtied.

Client failures

After a client sends a getNextEntity or getNextEntities request to the server, the client could fail as follows:

1. The client sends a request to the server and then goes down.
2. The client gets one or more entities from the server and then goes down.

In the first case, the server discovers that the client is going down when it tries to send back the response to the client. In the second case, when the client gets one or more entities from the server, an X lock is placed on these entities. If the client goes down, the transaction will eventually time out, and the X lock will be released.

Query with ORDER BY clause

Generally, query queues do not honor the ORDER BY clause. If you call getNextEntity or getNextEntities from the query queue, there is no guarantee the entities are returned according to the order. The reason is that the entities cannot be ordered across partitions. In the case that the query queue is deployed to all partitions, when a getNextEntity or getNextEntities call is executed, a random partition is picked to process the request. Therefore, the order is not guaranteed.

ORDER BY is honored if a query queue is deployed to a single partition.

For more information see “EntityManager Query API” on page 200.

One call per transaction

Each QueryQueue.getNextEntity or QueryQueue.getNextEntities call retrieves matched entities from one random partition. Applications should call exactly one QueryQueue.getNextEntity or QueryQueue.getNextEntities on one transaction. Otherwise eXtreme Scale could end up touching entities from multiple partitions, causing an exception to be thrown at the commit time.

EntityTransaction interface

You can use the EntityTransaction interface to demarcate transactions.

Purpose

To demarcate a transaction, you can use the EntityTransaction interface, which is associated with an entity manager instance. Use the EntityManager.getTransaction method to retrieve the EntityTransaction instance for the entity manager. Each EntityManager and EntityTransaction instance are associated with the Session. You can demarcate transactions with either the EntityTransaction or Session. Methods on the EntityTransaction interface do not have any checked exceptions. Only runtime exceptions of type PersistenceException or its subclasses result.

For more information about the EntityTransaction interface, see the API documentationEntityTransaction interface in the API documentation.

Retrieving entities and objects (Query API)

WebSphere eXtreme Scale provides a flexible query engine for retrieving entities using the EntityManager API and Java objects using the ObjectQuery API.

WebSphere eXtreme Scale query capabilities

With the eXtreme Scale query engine, you can perform SELECT type queries over an entity or object-based schema using the eXtreme Scale query language.

This query language provides the following capabilities:

- Single and multi-valued results
- Aggregate functions

- Sorting and grouping
- Joins
- Conditional expressions with subqueries
- Named and positional parameters
- eXtreme Scale index use
- Path expression syntax for object navigation
- Pagination

Query interface

Use the query interface to control entity query execution.

Use the `EntityManager.createQuery(String)` method to create a `Query`. You can use each query instance multiple times with the `EntityManager` instance in which it was retrieved.

Each query result produces an entity, where the entity key is the row ID (of type `long`) and the entity value contains the field results of the `SELECT` clause. You can use each query result in subsequent queries.

The following methods are available on the `com.ibm.websphere.objectgrid.em.Query` interface.

`public ObjectMap getResultMap()`

The `getResultMap` method runs a `SELECT` query and returns the results in an `ObjectMap` object with the results in query-specified order. The resulting `ObjectMap` is valid only for the current transaction.

The map key is the result number, of type `long`, starting at 1. The map value is of type `com.ibm.websphere.projector.Tuple` where each attribute and association is named based on its ordinal position within the select clause of the query. Use the method to retrieve the `EntityMetadata` for the `Tuple` object that is stored within the map.

The `getResultMap` method is the fastest method for retrieving query result data where multiple results can exist. You can retrieve the name of the resulting entity using the `ObjectMap.getEntityMetadata()` and `EntityMetadata.getName()` methods.

Example: The following query returns two rows.

```
String q1 = SELECT e.name, e.id, d from Employee e join e.dept d WHERE d.number=5
Query q = em.createQuery(q1);
ObjectMap resultMap = q.getResultMap();
long rowID = 1; // starts with index 1
Tuple tResult = (Tuple) resultMap.get(new Long(rowID));
while(tResult != null) {
    // The first attribute is name and has an attribute name of 1
    // But has an ordinal position of 0.
    String name = (String)tResult.getAttribute(0);
    Integer id = (String)tResult.getAttribute(1);

    // Dept is an association with a name of 3, but
    // an ordinal position of 0 since it's the first association.
    // The association is always a OneToOne relationship,
    // so there is only one key.
    Tuple deptKey = tResult.getAssociation(0,0);
    ...
    ++rowID;
    tResult = (Tuple) resultMap.get(new Long(rowID));
}
}
```

public Iterator getResultIterator

The `getResultIterator` method runs a `SELECT` query and returns the query results using an `Iterator` where each result is either an `Object` for a single-valued query, or an `Object` array for a multiple-valued query. The values in the `Object[]` result are stored in query order. The result `Iterator` is valid for the current transaction only.

This method is preferred for retrieving query results within the `EntityManager` context. You can use the optional `setResultEntityName(String)` method to name the resulting entity so that it can be used in further queries.

Example: The following query returns two rows.

```
String q1 = SELECT e.name, e.id, e.dept from Employee e WHERE e.dept.number=5
Query q = em.createQuery(q1);
Iterator results = q.getResultIterator();
while(results.hasNext()) {
    Object[] curEmp = (Object[]) results.next();
    String name = (String) curEmp[0];
    Integer id = (Integer) curEmp[1];
    Dept d = (Dept) curEmp[2];
    ...
}
```

public Iterator getResultIterator(Class resultType)

The `getResultIterator(Class resultType)` method runs a `SELECT` query and returns the query results using an entity `Iterator`. The entity type is determined by the `resultType` parameter. The result `Iterator` is valid only for the current transaction.

Use this method when you want to use the `EntityManager` APIs to access the resulting entities.

Example: The following query returns all of the employees and the department to which they belong for one division, ordering by salary. To print out the five employees with the highest salaries and then select work with employees from only one department in the same working set, use the following code.

```
String string_q1 = "SELECT e.name, e.id, e.dept from Employee e WHERE
    e.dept.division='Manufacturing' ORDER BY e.salary DESC";
Query query1 = em.createQuery(string_q1);
query1.setResultEntityName("AllEmployees");
Iterator results1 = query1.getResultIterator(EmployeeResult.class);
int curEmployee = 0;
System.out.println("Highest paid employees");
while (results1.hasNext() && curEmployee++ < 5) {
    EmployeeResult curEmp = (EmployeeResult) results1.next();
    System.out.println(curEmp);
    // Remove the employee from the resultset.
    em.remove(curEmp);
}

// Flush the changes to the result map.
em.flush();

// Run a query against the local working set without the employees we
// removed
String string_q2 = "SELECT e.name, e.id, e.dept from AllEmployees e
    WHERE e.dept.name='Hardware'";
Query query2 = em.createQuery(string_q2);
Iterator results2 = query2.getResultIterator(EmployeeResult.class);
System.out.println("Subset list of Employees");
while (results2.hasNext()) {
    EmployeeResult curEmp = (EmployeeResult) results2.next();
    System.out.println(curEmp);
}
```

public Object getSingleResult

The `getSingleResult` method runs a `SELECT` query that returns a single result.

If the `SELECT` clause has more than one field defined, then the result is an object array, where each element in the array is based on its ordinal position within the `SELECT` clause of the query.

```
String q1 = "SELECT e from Employee e WHERE e.id=100"
Employee e = em.createQuery(q1).getSingleResult();

String q1 = "SELECT e.name, e.dept from Employee e WHERE e.id=100"
Object[] empData = em.createQuery(q1).getSingleResult();
String empName = (String) empData[0];
Department empDept = (Department) empData[1];
```

public Query setResultEntityName(String entityName)

The `setResultEntityName(String entityName)` method specifies the name of the query result entity.

Each time the `getResultIterator` or `getResultMap` methods are invoked, an entity with an `ObjectMap` is dynamically created to hold the results of the query. If the entity is not specified, or null, the entity and `ObjectMap` name are automatically generated.

Because all query results are available for the duration of a transaction, a query name cannot be reused in a single transaction.

public Query setPartition(int partitionId)

Set the partition to where the query routes.

This method is required if the maps in the query are partitioned and if the entity manager does not have affinity to a single schema root entity partition.

Use the `PartitionManager` Interface to determine the number of partitions for the backing map of a given entity.

The following table provides descriptions of the other methods that are available through the query interface.

Table 3. Other methods.

Method	Result
<code>public Query setMaxResults(int maxResult)</code>	Set the maximum number of results to retrieve.
<code>public Query setFirstResult(int startPosition)</code>	Set the position of the first result to retrieve.
<code>public Query setParameter(String name, Object value)</code>	Bind an argument to a named parameter.
<code>public Query setParameter(int position, Object value)</code>	Bind an argument to a positional parameter.
<code>public Query setFlushMode(FlushModeType flushMode)</code>	Set the flush mode type to be used when the query runs, overriding the flush mode type set on the <code>EntityManager</code> .

eXtreme Scale query elements

With the eXtreme Scale query engine, you can use a single query language for searching the eXtreme Scale cache. This query language can query Java objects that are stored in ObjectMap objects and Entity objects. Use the following syntax for creating a query string.

An eXtreme Scale query is a string that contains the following elements:

- A SELECT clause that specifies the objects or values to return.
- A FROM clause that names the object collections.
- An optional WHERE clause that contains search predicates over the collections.
- An optional GROUP BY and HAVING clause (see eXtreme Scale query aggregation functions).
- An optional ORDER BY clause that specifies the ordering of the result collection.

Collections of Java objects are identified in queries through the use of their name in the query FROM clause.

The elements of query language are discussed in more detail in the following related topics:

- “ObjectGrid query Backus-Naur Form” on page 212 syntax
- “Reference for eXtreme Scale queries” on page 204

The following topics describe the means to use the Query API:

- “EntityManager Query API” on page 200
- “Using the ObjectQuery API” on page 196

Querying data in multiple time zones

In a distributed scenario, queries actually run on servers. When querying data with predicates of type calendar, java.util.Date and timestamp, the specified date time value in a query is based on the local time zone of the server.

In a single time-zone system where all clients and servers run on same time zone, you do not need to consider issues related to predicate types with calendar, java.util.Date and timestamp. However, when clients and servers are in different time zones, the specified date time value in queries is based on the server time zone and may return unwanted data back to client. Without knowing the server time zone, the specified date time value is meaningless. So the specified date time value should consider the time zone offset difference between the target time zone and the server time zone.

Time zone offset

For example, assume that a client is in [GMT-0] time zone and the server is in [GMT-6] time zone. The server time zone is 6 hours behind the client. The client would like to run the following query:

```
SELECT e FROM Employee e WHERE e.birthDate='1999-12-31 06:00:00'
```

Assuming the entity Employee has a birthDate attribute that is of type java.util.Date, the client is in [GMT-0] time zone and wants to retrieve Employees with birthDate value as '1999-12-31 06:00:00 [GMT-0]' based on its time zone.

The query will run on the server and the birthDate value used by the query engine will be '1999-12-31 06:00:00 [GMT-6]' that equals to '1999-12-31 12:00:00 [GMT-0]'.

Employees with birthDate value equal to '1999-12-31 12:00:00 [GMT-0]' will be returned to the client. Thus, the client will not get wanted Employees with birthDate value '1999-12-31 06:00:00 [GMT-0]'.

The problem described occurs because of the time zone difference between client and server. To solve this problem, one approach is to calculate the time zone offset between client and server and apply the time zone offset on the target date time value in the query. In the previous query example, the time zone offset is -6 hours, and the adjusted birthDate predicate should be "birthDate='1999-12-31 00:00:00'" if the client intends to retrieve Employees with birthDate value '12-31 06:00:00 [GMT-0]'. With the adjusted birthDate value, the server will use '1999-12-31 00:00:00 [GMT-6]' that equals to target value '12-31 06:00:00 [GMT-0]', and the required Employees will be returned to the client.

Distributed deployment in multiple time zones

If the distributed eXtreme Scale grid is deployed into multiple ObjectGrid servers in various time zones, the adjusting time zone offset approach will not work because the client will not know which server will run the query and thus cannot determine the time zone offset to use. The only solution is to use suffix 'Z' (not case sensitive) on JDBC date and time escape format to indicate using GMT time zone based date time value. The suffix 'Z' (not case sensitive) indicates to use GMT time zone based date time value. Without the suffix 'Z', the local time zone based date time value will be used in the process that runs the query.

The following query is equivalent to the previous example, but uses the suffix 'Z' instead:

```
SELECT e FROM Employee e WHERE e.birthDate='1999-12-31 06:00:00Z'
```

The query should find Employees with birthDate value '1999-12-31 06:00:00'. The suffix 'Z' indicates the specified birthDate value is GMT time zone based, so the GMT time zone based birthDate value '1999-12-31 06:00:00 [GMT-0]' will be used by the query engine for matching criteria value. Employees with birthDate attribute value equal to this GMT based birthDate value '1999-12-31 06:00:00 [GMT-0]' will be included in query result. Using the suffix 'Z' on JDBC date time escape format in any query is crucial to make applications time zone safe. Without this approach, the date time value is server time zone based and is meaningless from the client perspective when clients and servers are in different time zones.

For more information, see the topic on inserting data for different time zones in the *Product Overview*.

Data for different time zones

When inserting data with calendar, java.util.Date, and timestamp attributes into an ObjectGrid, you must ensure these date time attributes are created based on same time zone, especially when deployed into multiple servers in various time zones. Using the same time zone based date time objects can ensure the application is time-zone safe and data can be queried by calendar, java.util.Date and timestamp predicates.

Without explicitly specifying a time zone when creating date time objects, Java uses the local time zone and may cause inconsistent date time values in clients and servers.

Consider an example in a distributed deployment in which client1 is in time zone [GMT-0] and client2 is in [GMT-6] and both want to create a java.util.Date object

with value '1999-12-31 06:00:00'. Then client1 will create java.util.Date object with value '1999-12-31 06:00:00 [GMT-0]' and client2 will create java.util.Date object with value '1999-12-31 06:00:00 [GMT-6]'. Both java.util.Date objects are not equal because the time zone is different. A similar problem occurs when preloading data into partitions residing in servers in different time zones if local time zone is used to create date time objects.

To avoid the described problem, the application can choose a time zone such as [GMT-0] as the base time zone for creating calendar, java.util.Date, and timestamp objects.

Using the ObjectQuery API

The ObjectQuery API provides methods for querying data in the ObjectGrid that is stored using the ObjectMap API. When a schema is defined in the ObjectGrid instance, the ObjectQuery API can be used to create and run queries over the heterogeneous objects stored in the object maps.

Query and object maps

You can use an enhanced query capability for objects that are stored using the ObjectMap API. These queries allow retrieval of objects using non-key attributes and performs simple aggregations such as sum, avg, min, and max against all the data that matches a query. Applications can construct a query using the Session.createObjectQuery method. This method returns an ObjectQuery object which can then be interrogated to obtain the query results. The query object also allows the query to be customized before running the query. The query is run automatically when any method returning the result is called.

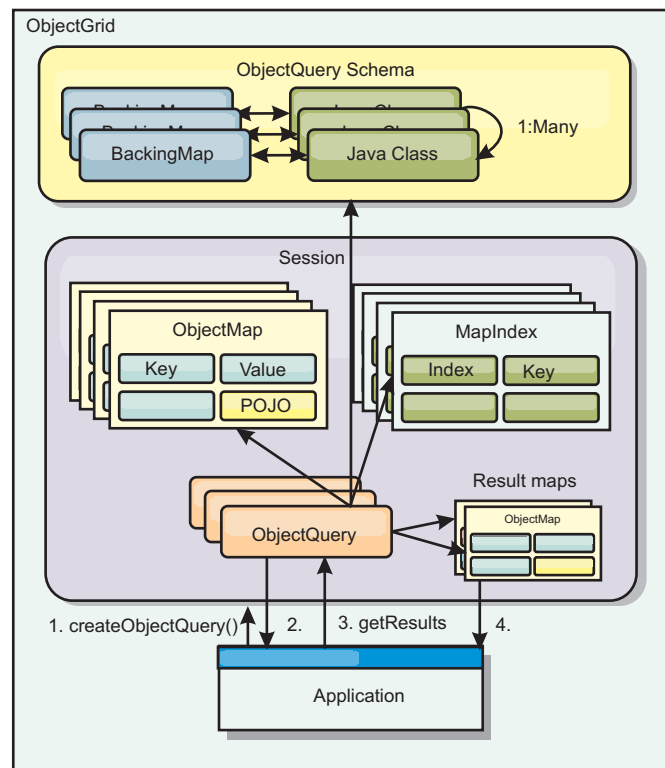


Figure 25. The interaction of the query with the ObjectGrid object maps and how a schema is defined for classes and associated with an ObjectGrid map

Defining an ObjectMap schema

Object maps are used to store objects in various forms and are largely unaware of the format. A schema must be defined in the ObjectGrid that defines the format of the data. A schema is composed of the following pieces:

- The type of object stored in the ObjectMap
- Relationships between ObjectMaps
- The method for which each query should access the data attributes in the objects (fields or property methods)
- The primary key attribute name in the object.

See *Configuring an ObjectQuery schema* for details.

For an example on creating a schema programmatically or using the ObjectGrid descriptor XML file, see “ObjectQuery tutorial - step 3” on page 3the tutorial on the ObjectQuery in the *Product Overview*.

Querying objects with the ObjectQuery API

The ObjectQuery interface allows the querying of non-entity objects, which are heterogeneous objects that are stored directly in the ObjectGrid ObjectMaps. The ObjectQuery API provides an easy way to find ObjectMap objects without using the index mechanism directly.

There are two methods for retrieving results from an ObjectQuery: getResultIterator and getResultMap.

Retrieving query results using getResultIterator

Query results are basically a list of attributes. Suppose the query was select a,b,c from X where y=z. This query returns a list of rows containing a, b and c. This list is actually stored in a transaction scoped Map, which means that you must associate an artificial key with each row and use an integer that increases with each row. This map is obtained using the ObjectQuery.getResultMap() method. You can access the elements of each row using code similar to the following:

```
ObjectQuery q = session.createQuery(
    "select c.id, c.firstName, c.surname from Customer c where c.surname=?1");

q.setParameter(1, "Claus");

Iterator iter = q.getResultIterator();
while(iter.hasNext())
{
    Object[] row = (Object[])iter.next();
    System.out.println("Found a Claus with id "
        + row[objectgrid: 0 ] + ", firstName: "
        + row[objectgrid: 1 ] + ", surname: "
        + row[objectgrid: 2 ]);
}
```

Retrieving query results using getResultMap

Query results can also be retrieved using the result map directly. The following example shows a query retrieving specific parts of the matching Customers and demonstrates how to access the resulting rows. Notice that if you use the

ObjectQuery object to access the data, then the generated long row identifier is hidden. The long row is only visible when using the ObjectMap to access the result.

When the transaction is completed this map disappears. The map is also only visible to the session used, that is, normally to just the thread that created it. The map uses a key of type Long which represents the row ID. The values stored in the map either are of type Object or Object[], where each element matches the type of the element in the select clause of query.

```
ObjectQuery q = em.createQuery(
    "select c.id, c.firstName, c.surname from Customer c where c.surname=?1");
q.setParameter(1, "Claus");
ObjectMap qmap = q.getResultMap();
for(long rowId = 0; true; ++rowId)
{
    Object[] row = (Object[]) qmap.get(new Long(rowId));
    if(row == null) break;
    System.out.println(" I Found a Claus with id " + row[0]
        + ", firstName: " + row[1]
        + ", surname: " + row[2]);
}
```

For examples on using the ObjectQuery, see “Tutorial: Querying a local in-memory data grid” on page 1the tutorial on the ObjectQuery API in the *Product Overview*.

Configuring an ObjectQuery schema:

ObjectQuery relies on schema or shape information to perform semantic checking and to evaluate path expressions. This section describes how to define the schema in XML or programmatically.

Defining the schema

The ObjectMap schema is defined in the ObjectGrid deployment descriptor XML or programmatically using the normal eXtreme Scale configuration techniques. For an example on how to create a schema, see “Configuring an ObjectQuery schema”

Schema information describes plain old Java objects (POJOs): which attributes they consist of and what types of attributes there might be, whether the attributes are primary key fields, single-valued or multi-valued relationships, or bidirectional relationships. Schema information directs ObjectQuery to use field access or property access.

Queryable attributes

When the schema is defined in the ObjectGrid, the objects in the schema are introspected using reflection to determine which attributes are available for querying. You can query the following attribute types:

- Java primitive types including wrappers
- java.lang.String
- java.math.BigInteger
- java.math.BigDecimal
- java.util.Date
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp

- java.util.Calendar
- byte[]
- java.lang.Byte[]
- char[]
- java.lang.Character[]
- J2SE enum

Embedded serializable types other than those stated previously can also be included in a query result, but cannot be included in the WHERE or FROM clause of the query. Serializable attributes are not navigable.

Attribute types can be excluded from the schema if the type is not serializable, the field or property is static, or the field is transient. Since all map objects must be serializable, the ObjectGrid only includes attributes that can be persisted from the object. Other objects are ignored.

Field attributes

When the schema is configured to access the object using fields, all serializable, non-transient fields are automatically incorporated into the schema. To select a field attribute in a query, use the field identifier name as it exists in the class definition.

All public, private, protected and package protected fields are included in the schema.

Property attributes

When the schema is configured to access the object using properties, all serializable methods that follow the JavaBeans property naming conventions will automatically be incorporated into the schema. To select a property attribute for the query, use the JavaBeans style property name conventions.

All public, private, protected and package protected properties are included in the schema.

In the following class, the following attributes are added to the schema: name, birthday, valid.

```
public class Person {
    public String getName(){
    private java.util.Date getBirthday(){
    boolean isValid(){
    public NonSerializableObject getData(){
}
```

When using a CopyMode of COPY_ON_WRITE, the query schema must always use property-based access. COPY_ON_WRITE creates proxy objects whenever objects are retrieved from the map and can only access those objects using property methods. Failure to do so will result in each query result being set to null.

Relationships

Each relationship must be explicitly defined in the schema configuration. The cardinality of the relationship is automatically determined by the type of the

attribute. If the attribute implements the `java.util.Collection` interface, then the relationship is either a one-to-many or many-to-many relationship.

Unlike entity queries, attributes that refer to other cached objects must not store direct references to the object. References to other objects are serialized as part of the containing object's data. Instead, store the key to the related object.

For example, if there is a many-to-one relationship between a Customer and Order:

Incorrect. Storing an object reference.

```
public class Customer {
    String customerId;
    Collection<Order> orders;
}
```

```
public class Order {
    String orderId;
    Customer customer;
}
```

Correct. The key to the related object.

```
public class Customer {
    String customerId;
    Collection<String> orders;
}
```

```
public class Order {
    String orderId;
    String customer;
}
```

When a query is run that joins the two map objects together, the key will automatically be inflated. For example, the following query would return Customer objects:

```
SELECT c FROM Order o JOIN Customer c WHERE orderId=5
```

Using indexes

ObjectGrid uses index plugins to add indexes to maps. The query engine automatically incorporates any indexes that are defined on a schema map element of the type: `com.ibm.websphere.objectgrid.plugins.index.HashIndex` and the `rangeIndex` property is set to true. If the index type is not `HashIndex` and the `rangeIndex` property is not set to true, then the index is ignored by the query. See "ObjectQuery tutorial - step 2" on page 2 the ObjectQuery tutorial in the *Product Overview* for an example on how to add an index to the schema.

EntityManager Query API

The EntityManager API provides methods for querying data in the ObjectGrid that is stored using the EntityManager API. The EntityManager Query API is used to create and run queries over one or more entities defined in eXtreme Scale.

Query and ObjectMaps for entities

WebSphere Extended Deployment v6.1 introduced an enhanced query capability for entities stored in eXtreme Scale. These queries allow objects to be retrieved using non-key attributes and to perform simple aggregations such as the sum, average, minimum, and maximum against all the data that matches a query. Applications construct a query using the `EntityManager.createQuery` API. This

returns a Query object and can then be interrogated to obtain the query results. The query object also allows the query to be customized before running the query. The query is run automatically when any method returning the result is called.

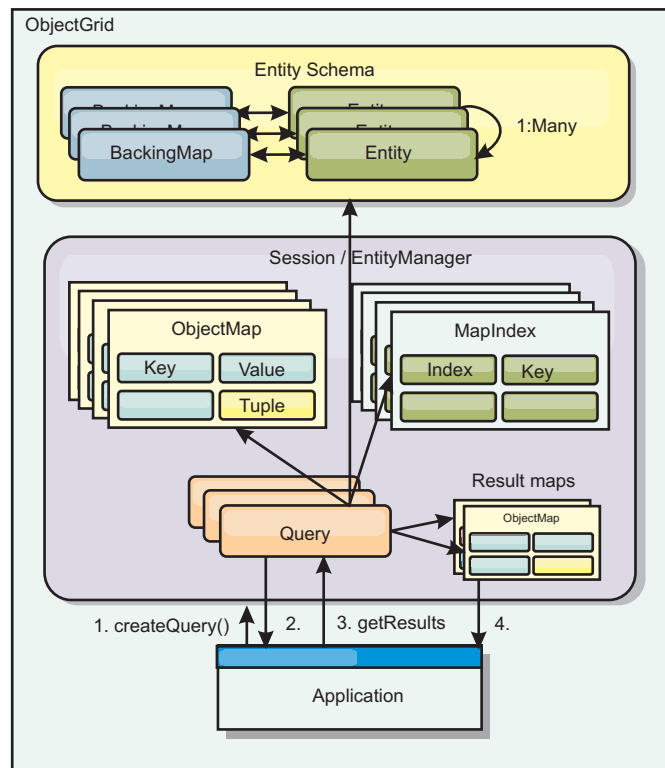


Figure 26. The interaction of the query with the ObjectGrid object maps and how the entity schema is defined and associated with an ObjectGrid map.

Retrieving query results using the getResultIterator method

Query results are a list of attributes. If the query was `select a,b,c from X where y=z`, then a list of rows containing a, b and c is returned. This list is stored in a transaction scoped Map, which means that you must associated an artificial key with each row and use an integer that increases with each row. This map is obtained using the `Query.getResultMap` method. The map has `EntityMetaData`, which describes each row in the Map associated with it. You can access the elements of each row using code similar to the following:

```
Query q = em.createQuery("select c.id, c.firstName, c.surname from Customer c where c.surname=?1");
q.setParameter(1, "Claus");

Iterator iter = q.getResultIterator();
while(iter.hasNext())
{
    Object[] row = (Object[])iter.next();
    System.out.println("Found a Claus with id " + row[objectgrid: 0 ]
        + ", firstName: " + row[objectgrid: 1 ]
        + ", surname: " + row[objectgrid: 2 ]);
}
```

Retrieving query results using getResultMap

The following code shows the retrieval of specific parts of the matching Customers and shows how to access the resulting rows. If you use the Query object to access the data, then the generated long row identifier is hidden. The long is only visible when using the ObjectMap to access the result. When the transaction is completed,

then this Map disappears. The Map is only visible to the Session used, that is, normally to just the thread that created it. The Map uses a Tuple for the key with a single attribute, a long with the row ID. The value is another tuple with an attribute for each column in the result set.

The following sample code demonstrates this:

```
Query q = em.createQuery("select c.id, c.firstName, c.surname from
Customer c where c.surname=?1");
q.setParameter(1, "Claus");
ObjectMap qmap = q.getResultMap();
Tuple keyTuple = qmap.getEntityMetadata().getKeyMetadata().createTuple();
for(long i = 0; true; ++i)
{
    keyTuple.setAttribute(0, new Long(i));
    Tuple row = (Tuple)qmap.get(keyTuple);
    if(row == null) break;
    System.out.println(" I Found a Claus with id " + row.getAttribute(0)
        + ", firstName: " + row.getAttribute(1)
        + ", surname: " + row.getAttribute(2));
}
```

Retrieving query results using an entity result iterator

The following code shows the query and the loop that retrieves each result row using the normal Map APIs. The key for the Map is a Tuple. So, construct one of the correct types using the createTuple method result in keyTuple. Try to retrieve all rows with rowIds from 0 onwards. When you get returns null (indicating key not found), then the loop finishes. Set the first attribute of keyTuple to be the long that you want to find. The value returned by get is also a Tuple with an attribute for each column in the query result. Then, pull each attribute from the value Tuple using getAttribute.

Following is the next code fragment:

```
Query q2 = em.createQuery("select c.id, c.firstName, c.surname from Customer c where c.surname=?1");
q2.setResultEntityName("CustomerQueryResult");
q2.setParameter(1, "Claus");

Iterator iter2 = q2.getResultIterator(CustomerQueryResult.class);
while(iter2.hasNext())
{
    CustomerQueryResult row = (CustomerQueryResult)iter2.next();
    // firstName is the id not the firstName.
    System.out.println("Found a Claus with id " + row.id
        + ", firstName: " + row.firstName
        + ", surname: " + row.surname);
}

em.getTransaction().commit();
```

Specified is a ResultEntityName value on the query. This value tells the query engine that you want to project each row to a specific object, CustomerQueryResult in this case. The class follows:

```
@Entity
public class CustomerQueryResult {
    @Id long rowId;
    String id;
    String firstName;
    String surname;
};
```

In the first snippet, notice that the each query row is returned as a CustomerQueryResult object rather than an Object[]. The result columns of the query are projected to the CustomerQueryResult object. Projecting the result is slightly slower at run time but more readable. Query result Entities should not be

registered with eXtreme Scale at startup. If the entities are registered, then a global Map with the same name is created, and the query fails with an error indicating duplicate Map name.

Simple queries with EntityManager:

WebSphere eXtreme Scale comes with EntityManager query API.

The EntityManager query API is very similar to SQL other query engines that query over objects. A query is defined, then the result is retrieved from the query using various getResult methods.

The following examples refer to the entities used in the EntityManager tutorial in the Product Overview.

Running a simple query

In this example, customers with the surname of Claus are queried:

```
em.getTransaction().begin();

    Query q = em.createQuery("select c from Customer c where c.surname='Claus'");

    Iterator iter = q.getResultIterator();
    while(iter.hasNext())
    {
        Customer c = (Customer)iter.next();
        System.out.println("Found a claus with id " + c.id);
    }

    em.getTransaction().commit();
```

Using parameters

Since you want to find all customers with a surname of Claus, a parameter to specify the surname is used since you might want to use this query more than once.

Positional Parameter Example

```
Query q = em.createQuery("select c from Customer c where c.surname=?1");
    q.setParameter(1, "Claus");
```

Using parameters is very important when the query is used more than once. The EntityManager needs to parse the query string and build a plan for the query, which is expensive. By using a parameter, the EntityManager caches the plan for the query, thereby reducing the time it takes to run a query.

Both positional and named parameters are used:

Named Parameter Example

```
Query q = em.createQuery("select c from Customer c where c.surname=:name");
    q.setParameter("name", "Claus");
```

Using an index to improve performance

If there are millions of customers, then the previous query needs to scan over all rows in the Customer Map. This is not very efficient. But eXtreme Scale provides a

mechanism for defining indexes over individual attributes in an entity. The query automatically uses this index when appropriate, which can speed up queries dramatically.

You can specify which attributes to index very simply by using the `@Index` annotation on the entity attribute:

```
@Entity
public class Customer
{
    @Id String id;
    String firstName;
    @Index String surname;
    String address;
    String phoneNumber;
}
```

The `EntityManager` creates an appropriate `ObjectGrid` index for the `surname` attribute in the `Customer` entity and the query engine automatically uses the index, which greatly decreases the query time.

Using pagination to improve performance

If there are a million customers named `Claus`, then it is not likely that you would want to display a page displaying a million customers. It is more likely that you would want to display 10 or 25 customers at a time.

The Query `setFirstResult` and `setMaxResults` methods helps by only returning a subset of the results.

Pagination Example

```
Query q = em.createQuery("select c from Customer c where c.surname=:name");
q.setParameter("name", "Claus");
// Display the first page
q.setFirstResult=1;
q.setMaxResults=25;
displayPage(q.getResultIterator());

// Display the second page
q.setFirstResult=26;
displayPage(q.getResultIterator());
```

Reference for eXtreme Scale queries

WebSphere eXtreme Scale has its own language by which the user can query data.

ObjectGrid query FROM clause

The `FROM` clause specifies the collections of objects to which to apply the query. Each collection is identified either by an abstract schema name and an identification variable, called a range variable, or by a collection member declaration that identifies either a single or multi-valued relationship and an identification variable.

Conceptually, the semantics of the query is to first form a temporary collection of tuples, referred to as `R`. Tuples are composed of elements from the collections that are identified in the `FROM` clause. Each tuple contains one element from each of the collections in the `FROM` clause. All possible combinations are formed subject to the constraints that are imposed by the collection member declarations. If any schema name identifies a collection for which there are no records in the persistent store, then the temporary collection `R` is empty.

Examples using FROM

The DeptBean object contains records 10, 20 and 30. The EmpBean object contains records 1, 2 and 3 that are related to department 10 and records 4 and 5 that are related to department 20. Department 30 has no related employees.

```
FROM DeptBean d, EmpBean e
```

This clause forms a temporary collection R that contains 15 tuples.

```
FROM DeptBean d, DeptBean d1
```

This clause forms a temporary collection R that contains 9 tuples.

```
FROM DeptBean d, IN (d.emps) AS e
```

This clause forms a temporary collection R that contains 5 tuples. Department 30 is not in the R temporary collection because it contains no employees. Department 10 is contained in the R temporary collection three times and department 20 is contained in R twice.

Instead of using IN(d.emps) as e, you can use a JOIN predicate:

```
FROM DeptBean d JOIN d.emps as e
```

After forming the temporary collection, the search conditions of the WHERE clause are applied to the R temporary collection, yielding a new temporary collection R1. The ORDER BY and SELECT clauses are applied to R1 to yield the final result set.

An identification variable is a variable that is declared in the FROM clause using the IN operator or the optional AS operator.

```
FROM DeptBean AS d, IN (d.emps) AS e
```

is equivalent to:

```
FROM DeptBean d, IN (d.emps) e
```

An identification variable that is declared to be an abstract schema name is called a range variable. In the previous query, "d" is a range variable. An identification variable that is declared to be a multi-valued path expression is called a collection member declaration. The "d" and "e" values in the previous example are collection member declarations.

An example of using a single-valued path expression in the FROM clause follows:

```
FROM EmpBean e, IN(e.dept.mgr) as m
```

ObjectGrid query SELECT clause

The syntax of the SELECT clause is illustrated in the following example:

```
SELECT { ALL | DISTINCT } [ selection , ]* selection
```

```

selection ::= {single_valued_path_expression |
              identification_variable |
              OBJECT ( identification_variable) |
              aggregate_functions } [[ AS ] id ]

```

The SELECT clause consists of one or more of the following elements: a single identification variable that is defined in the FROM clause, a single-valued path expression that evaluates to object references or values, and an aggregate function. You can use the DISTINCT keyword to eliminate duplicate references.

A scalar-subselect is a subselect that returns a single value.

Examples using SELECT

Find all employees that earn more than the John employee:

```

SELECT OBJECT(e) FROM EmpBean ej, EmpBean e WHERE ej.name = 'John' and
e.salary > ej.salary

```

Find all departments that have one or more employees who earn less than 20000:

```

SELECT DISTINCT e.dept FROM EmpBean e where e.salary < 20000

```

A query can have a path expression that evaluates to an arbitrary value:

```

SELECT e.dept.name FROM EmpBean e where e.salary < 20000

```

The previous query returns a collection of name values for the departments that have employees who earn less than 20000.

A query can return an aggregate value:

```

SELECT avg(e.salary) FROM EmpBean e

```

A query that retrieves the names and object references for underpaid employees follows:

```

SELECT e.name as name, object(e) as emp from EmpBean e where e.salary <
50000

```

ObjectGrid query WHERE clause

The WHERE clause contains search conditions that are composed of the elements presented below. When a search condition evaluates to TRUE, the tuple is added to the result set.

ObjectGrid query literals

A string literal is enclosed in single quotes. A single quotation mark that occurs within a string literal is represented by two single quotes, for example: 'Tom's'.

A numeric literal can be any of the following values:

- An exact value such as 57, -957, or +66
- Any value supported by Java long type
- A decimal literal such as 57.5 or -47.02
- An approximate numeric value such as 7E3 or -57.4E-2

- Float types must include the "F" qualifier, for example 1.0F
- Long types must include the "L" qualifier, for example 123L

Boolean literals are TRUE and FALSE.

Temporal literals follow JDBC escape syntax based on the type of attribute:

- java.util.Date: yyyy-mm-ss
- java.sql.Date: yyyy-mm-ss
- java.sql.Time: hh-mm-ss
- java.sql.Timestamp: yyyy-mm-dd hh:mm:ss.f...
- java.util.Calendar: yyyy-mm-dd hh:mm:ss.f...

Enum literals are expressed using Java enum literal syntax using the fully qualified enum class name.

ObjectGrid query input parameters

You can specify input parameters by either using an ordinal position or by using a variable name. Writing queries that use input parameters is strongly encouraged, because using input parameters increases performance by allowing the ObjectGrid to catch the query plan between running actions.

An input parameter can be any of the following types: Byte, Short, Integer, Long, Float, Double, BigDecimal, BigInteger, String, Boolean, Char, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.util.Calendar, a Java SE 5 enum, an Entity or POJO Object, or a binary data string in the form of Java byte[].

An input parameter must not have a NULL value. To search for the occurrence of a NULL value, use the NULL predicate.

Positional Parameters

Positional input parameters are defined by using question mark followed by a positive number:

?[positive integer].

Positional input parameters are numbered starting at 1 and correspond to the arguments of the query; therefore, a query must not contain an input parameter that exceeds the number of input arguments.

Example: SELECT e FROM Employee e WHERE e.city = ?1 and e.salary >= ?2

Named Parameters

Named input parameters are defined using a variable name in the format: :[parameter name].

Example: SELECT e FROM Employee e WHERE e.city = :city and e.salary >= :salary

ObjectGrid query BETWEEN predicate

The BETWEEN predicate determines whether a given value lies between two other given values.

```
expression [NOT] BETWEEN expression-2 AND expression-3
```

Example 1

```
e.salary BETWEEN 50000 AND 60000
```

is equivalent to:

```
e.salary >= 50000 AND e.salary <= 60000
```

Example 2

```
e.name NOT BETWEEN 'A' AND 'B'
```

is equivalent to:

```
e.name < 'A' OR e.name > 'B'
```

ObjectGrid query IN predicate

The IN predicate compares a value to a set of values. You can use the IN predicate in one of two forms:

```
expression [NOT] IN ( subselect )expression [NOT] IN ( value1, value2, .... )
```

The ValueN value can either be a literal value or an input parameter. The expression cannot evaluate to a reference type.

Example 1

```
e.salary IN ( 10000, 15000 )
```

is equivalent to

```
( e.salary = 10000 OR e.salary = 15000 )
```

Example 2

```
e.salary IN ( select e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

is equivalent to

```
e.salary = ANY ( select e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

Example 3

```
e.salary NOT IN ( select e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

is equivalent to

```
e.salary <> ALL ( select e1.salary from EmpBean e1 where e1.dept.deptno = 10)
```

ObjectGrid query LIKE predicate

The LIKE predicate searches a string value for a certain pattern.

```
string-expression [NOT] LIKE pattern [ ESCAPE escape-character ]
```

The pattern value is a string literal or parameter marker of type string in which the underscore (`_`) stands for any single character and percent (`%`) stands for any sequence of characters, including an empty sequence. Any other character stands for itself. The escape character can be used to search for character `_` and `%`. The escape character can be specified as a string literal or as an input parameter.

If the string-expression is null, then the result is unknown.

If both string-expression and pattern are empty, then the result is true.

Example

```
' ' LIKE ' ' is true
' ' LIKE '%' is true
e.name LIKE '12%3' is true for '123' '12993' and false for '1234'
e.name LIKE 's_me' is true for 'some' and 'same', false for 'soome'
e.name LIKE '/_foo' escape '/' is true for '_foo', false for 'afoo'
e.name LIKE '//_foo' escape '/' is true for '/afoo' and for '/bfoo'
e.name LIKE '///_foo' escape '/' is true for '/_foo' but false for '/afoo'
```

ObjectGrid query NULL predicate

The NULL predicate tests for null values.

```
{single-valued-path-expression | input_parameter} IS [NOT] NULL
```

Example

```
e.name IS NULL
e.dept.name IS NOT NULL
e.dept IS NOT NULL
```

ObjectGrid query EMPTY collection predicate

Use the EMPTY collection predicate to test for an empty collection.

To test if a multi-valued relationship is empty, use the following syntax:

```
collection-valued-path-expression IS [NOT] EMPTY
```

Example

Empty collection predicate To find all the departments that have no employees:

```
SELECT OBJECT(d) FROM DeptBean d WHERE d.emps IS EMPTY
```

ObjectGrid query MEMBER OF predicate

The following expression tests whether the object reference that is specified by the single valued path expression or input parameter is a member of the designated

collection. If the collection valued path expression designates an empty collection, then the value of the MEMBER OF expression is FALSE.

```
{ single-valued-path-expression | input_parameter } [ NOT ] MEMBER [ OF ]  
collection-valued-path-expression
```

Example

Find employees that are not members of a given department number:

```
SELECT OBJECT(e) FROM EmpBean e , DeptBean d  
WHERE e NOT MEMBER OF d.emps AND d.deptno = ?1
```

Find employees whose manager is a member of a given department number:

```
SELECT OBJECT(e) FROM EmpBean e, DeptBean d  
WHERE e.dept.mgr MEMBER OF d.emps and d.deptno=?1
```

ObjectGrid query EXISTS predicate

The EXISTS predicate tests for the presence or absence of a condition that specified by a subselect.

```
EXISTS ( subselect )
```

The result of EXISTS is true if the subselect returns at least one value, otherwise the result is false.

To negate an EXISTS predicate, precede the predicate with the NOT logical operator.

Example

Return departments that have at least one employee that earns more than 1000000:

```
SELECT OBJECT(d) FROM DeptBean d  
WHERE EXISTS ( SELECT e FROM IN (d.emps) e WHERE e.salary > 1000000 )
```

Return departments that have no employees:

```
SELECT OBJECT(d) FROM DeptBean d  
WHERE NOT EXISTS ( SELECT e FROM IN (d.emps) e)
```

You can also rewrite the previous query like in the following example:

```
SELECT OBJECT(d) FROM DeptBean d WHERE SIZE(d.emps)=0
```

ObjectGrid query ORDER BY clause

The ORDER BY clause specifies an ordering of the objects in the result collection. An example follows:

```
ORDER BY [ order_element ,]* order_element order_element ::= { path-expression } [  
ASC | DESC ]
```

The path expression must specify a single-valued field that is a primitive type of byte, short, int, long, float, double, char, or a wrapper type of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Character, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp and java.util.Calendar. The ASC order element

specifies that the results are displayed in ascending order, which is the default. A DESC order element specifies that the results are displayed in descending order.

Example

Return department objects. Display the department numbers in decreasing order:

```
SELECT OBJECT(d) FROM DeptBean d ORDER BY d.deptno DESC
```

Return employee objects, sorted by department number and name:

```
SELECT OBJECT(e) FROM EmpBean e ORDER BY e.dept.deptno ASC, e.name DESC
```

ObjectGrid query aggregation functions

Aggregation functions operate on a set of values to return a single scalar value. You can use these functions in the select and subselect methods. The following example illustrates an aggregation:

```
SELECT SUM (e.salary) FROM EmpBean e WHERE e.dept.deptno =20
```

This aggregation computes the total salary for department 20.

The aggregation functions are: AVG, COUNT, MAX, MIN, and SUM. The syntax of an aggregation function is illustrated in the following example:

```
aggregation-function ( [ ALL | DISTINCT ] expression )
```

or:

```
COUNT( [ ALL | DISTINCT ] identification-variable )
```

The DISTINCT option eliminates duplicate values before applying the function. The ALL option is the default option, and does not eliminate duplicate values. Null values are ignored in computing the aggregate function except when you use the COUNT(identification-variable) function, which returns a count of all the elements in the set.

Defining return type

The MAX and MIN functions can apply to any numeric, string or date-time data type and return the corresponding data type. The SUM and AVG functions take a numeric type as input. The AVG function returns a double type. The SUM function returns a long type if the input type is an integer type, except when the input is a Java BigInteger type, then the function returns a Java BigInteger type. The SUM function returns a double type if the input type is not an integer type, except when the input is a Java BigDecimal type, then the function returns a Java BigDecimal type. The COUNT function can take any data type except collections, and returns a long type.

When applied to an empty set, the SUM, AVG, MAX, and MIN functions can return a null value. The COUNT function returns zero (0) when it is applied to an empty set.

Using GROUP BY and HAVING clauses

The set of values that is used for the aggregate function is determined by the collection that results from the FROM and WHERE clause of the query. You can divide the set into groups and apply the aggregation function to each group. To perform this action, use a GROUP BY clause in the query. The GROUP BY clause defines grouping members, which comprise a list of path expressions. Each path expression specifies a field that is a primitive type of byte, short, int, long, float, double, boolean, char, or a wrapper type of Byte, Short, Integer, Long, Float, Double, BigDecimal, String, Boolean, Character, java.util.Date, java.sql.Date, java.sql.Time, java.sql.Timestamp, java.util.Calendar or a Java SE 5 enum.

The following example illustrates the use of the GROUP BY clause in a query that computes the average salary for each department:

```
SELECT e.dept.deptno, AVG ( e.salary) FROM EmpBean e GROUP BY e.dept.deptno
```

In division of a set into groups, a NULL value is considered equal to another NULL value.

Groups can be filtered using a HAVING clause that tests group properties before involving aggregate functions or grouping members. This filtering is similar to how the WHERE clause filters tuples (that is, records of the return collection values) from the FROM clause. An example of the HAVING clause follows:

```
SELECT e.dept.deptno, AVG ( e.salary) FROM EmpBean e
GROUP BY e.dept.deptno
HAVING COUNT(e) > 3 AND e.dept.deptno > 5
```

This query returns the average salary for departments that have more than three employees and the department number is greater than five.

You can use a HAVING clause without a GROUP BY clause. In this case, the entire set is treated as a single group, to which the HAVING clause is applied.

ObjectGrid query Backus-Naur Form:

A summary of the ObjectGrid Query Backus-Naur Form (BNF) Notation follows.

Table 4. Key to BNF summary

Representation	Description
{...}	Grouping
[...]	Optional constructs
bold	Keywords
*	Zero or more
	Alternates

```
ObjectGrid QL ::=select_clause from_clause [where_clause] [group_by_clause]
[having_clause] [order_by_clause]
from_clause ::=FROM identification_variable_declaration
[,identification_variable_declaration]*
identification_variable_declaration ::=collection_member_declaration |
range_variable_declaration
collection_member_declaration ::=IN ( collection_valued_path_expression |
single_valued_navigation) [AS] identifier | [LEFT [OUTER]
| INNER] JOIN collection_valued_path_expression |
single_valued_navigation [AS] identifier
range_variable_declaration ::=abstract_schema_name [AS] identifier
```

```

single_valued_path_expression ::= {single_valued_navigation | identification_variable}.
    { state_field | state_field.value_object_attribute } | single_valued_navigation
single_valued_navigation ::= identification_variable.[ single_valued_association_field. ]*
    single_valued_association_field
collection_valued_path_expression ::= identification_variable.[
    single_valued_association_field. ]* collection_valued_association_field
select_clause ::= SELECT [DISTINCT] [ selection , ]* selection
selection ::= {single_valued_path_expression | identification_variable | OBJECT
    ( identification_variable ) | aggregate_functions } [[ AS ] id ]
order_by_clause ::= ORDER BY [ {identification_variable.[ single_valued_association_field.
    ]*state_field} [ASC|DESC],]* {identification_variable.[
    single_valued_association_field. ]*state_field} [ASC|DESC]
where_clause ::= WHERE conditional_expression
conditional_expression ::= conditional_term | conditional_expression OR conditional_term
conditional_term ::= conditional_factor | conditional_term AND conditional_factor
conditional_factor ::= [NOT] conditional_primary
conditional_primary ::= simple_cond_expression | (conditional_expression)
simple_cond_expression ::= comparison_expression | between_expression | like_expression |
    in_expression | null_comparison_expression | empty_collection_comparison_expression |
    exists_expression | collection_member_expression
between_expression ::= numeric_expression [NOT] BETWEEN numeric_expression
    AND numeric_expression | string_expression [NOT] BETWEEN
    string_expression AND string_expression | datetime_expression [NOT]
    BETWEEN datetime_expression AND datetime_expression
in_expression ::= identification_variable.[ single_valued_association_field. ]state_field
    [NOT] IN { (subselect) | ( atom ,)* atom }
atom ::= { string_literal | numeric_literal | input_parameter }
like_expression ::= string_expression [NOT] LIKE {string_literal | input_parameter}
    [ESCAPE {string_literal | input_parameter}]
null_comparison_expression ::= {single_valued_path_expression | input_parameter} IS
    [ NOT ] NULL
empty_collection_comparison_expression ::= collection_valued_path_expression IS
    [NOT] EMPTY
collection_member_expression ::= { single_valued_path_expression | input_parameter } [
    NOT ] MEMBER [ OF ] collection_valued_path_expression
exists_expression ::= EXISTS {(subselect)}
subselect ::= SELECT [{ ALL | DISTINCT }] subselection from_clause
    [where_clause] [group_by_clause] [having_clause]
subselection ::= {single_valued_path_expression | identification_variable |
    aggregate_functions }
group_by_clause ::= GROUP BY[single_valued_path_expression,]*
    single_valued_path_expression
having_clause ::= HAVING conditional_expression
comparison_expression ::= numeric_expression comparison_operator { numeric_expression
    | {SOME | ANY | ALL} (subselect) } | string_expression
    comparison_operator {
string_expression | {SOME | ANY | ALL}(subselect) } |
datetime_expression comparison_operator {
datetime_expression {SOME | ANY | ALL}(subselect) } |
boolean_expression {=|<>} {
boolean_expression {SOME | ANY | ALL}(subselect) } |
entity_expression {=|<>} {
entity_expression {SOME | ANY | ALL}(subselect) }
comparison_operator ::= = | > | >= | < | <= | <>
string_expression ::= string_primary | (subselect)
string_primary ::= state_field_path_expression | string_literal | input_parameter |
    functions_returning_strings
datetime_expression ::= datetime_primary | (subselect)
datetime_primary ::= state_field_path_expression | string_literal | long_literal
    | input_parameter | functions_returning_datetime
boolean_expression ::= boolean_primary | (subselect)

```

```

boolean_primary ::= state_field_path_expression | boolean_literal | input_parameter
entity_expression ::= single_valued_association_path_expression |
    identification_variable | input_parameter
numeric_expression ::= simple_numeric_expression |(subselect)
simple_numeric_expression ::= numeric_term | numeric_expression {+|-} numeric_term
numeric_term ::= numeric_factor | numeric_term {*/} numeric_factor
numeric_factor ::= {+|-} numeric_primary
numeric_primary ::= single_valued_path_expression | numeric_literal |
    ( numeric_expression ) | input_parameter | functions
aggregate_functions ::=
AVG([ALL|DISTINCT] identification_variable.
    [ single_valued_association_field. ]*state_field) |
COUNT([ALL|DISTINCT] {single_valued_path_expression |
    identification_variable}) |
MAX([ALL|DISTINCT] identification_variable.[
    single_valued_association_field. ]*state_field) |
MIN([ALL|DISTINCT] identification_variable.[
    single_valued_association_field. ]*state_field) |
SUM([ALL|DISTINCT] identification_variable.[
    single_valued_association_field. ]*state_field)
functions ::=
ABS (simple_numeric_expression) |
CONCAT (string_primary , string_primary) |
LOWER (string_primary) |
LENGTH(string_primary) |
LOCATE(string_primary, string_primary [, simple_numeric_expression]) |
MOD (simple_numeric_expression, simple_numeric_expression) |
SIZE (collection_valued_path_expression) |
SQRT (simple_numeric_expression) |
SUBSTRING (string_primary, simple_numeric_expression[, simple_numeric_expression]) |
UPPER (string_primary) |
TRIM ([[LEADING | TRAILING | BOTH] [trim_character]
    FROM] string_primary)

```

Programming for transactions

Applications that require transactions introduce such considerations as handling locks, handling collisions, and transaction isolation.

Transaction processing overview

WebSphere eXtreme Scale uses transactions as its mechanism for interaction with data.

To interact with data, the thread in your application needs its own session. When the application wants to use the ObjectGrid on a thread, call one of the ObjectGrid.getSession methods to obtain a session. With the session, the application can work with data that is stored in the ObjectGrid maps.

When an application uses a Session object, the session must be in the context of a transaction. A transaction begins and commits or begins and rolls back using the begin, commit, and rollback methods on the Session object. Applications can also work in auto-commit mode, in which the Session automatically begins and commits a transaction whenever an operation is performed on the map. Auto-commit mode cannot group multiple operations into a single transaction, so it is the slower option if you are creating a batch of multiple operations into a single transaction. However, for transactions that contain only one operation, auto-commit is the faster option.

When your application is finished with the Session, use the optional `Session.close()` method to close the session. Closing the Session releases it from the heap and allows subsequent calls to the `getSession()` method to be reused, improving performance.

Data access and transactions:

After an application has a reference to an ObjectGrid instance or a client connection to a remote data grid, you can access and interact with data in your data grid. With the ObjectGridManager API, you can create a local instance or establish a client connection to a distributed instance. To create a local instance, use one of the `createObjectGrid` methods. To establish a client connection with a remote data grid, use the `getObjectGrid` method.

A thread in an application needs its own Session. When you want your application to use the ObjectGrid on a thread, call one of the `getSession` methods to obtain a Session. After the application is finished with the Session, call the `Session.close()` method. This method closes the session, returning it to the pool and releasing its resources. Closing a session is optional, but improves the performance of subsequent calls to the `getSession()` method. If the application is using a dependency injection framework such as Spring, you can inject a Session into an application bean when necessary.

After you obtain a Session, the application can access data stored in maps in the ObjectGrid. If the ObjectGrid uses entities, you can use the EntityManager API, which you can obtain with the `Session.getEntityManager` method. Because it is closer to Java specifications, the EntityManager interface is simpler than the map-based API. However, the EntityManager API carries a performance overhead because it tracks changes in objects. The map-based API is obtained by using the `Session.getMap` method.

WebSphere eXtreme Scale uses transactions. When an application interacts with a Session, it must be in the context of a transaction. A transaction is begun and committed or rolled back using the `Session.begin`, `Session.commit`, and `Session.rollback` methods on the Session object. Applications can also work in auto-commit mode, where the Session automatically begins and commits a transaction whenever the application interacts with Maps. However, the auto-commit mode is slower.

The logic of using transactions

Transactions may seem to be slow. You must use transactions for the following reasons:

1. To allow rollback of changes if an exception occurs or business logic needs to undo state changes.
2. To hold locks on data and release locks within the lifetime of a transaction, allowing a set of changes to be made atomically, that is, all changes or no changes to data.
3. To produce an atomic unit of replication.

You can customize how much transaction support is needed. Your application can turn off rollback support and locking but does so at a cost to the application. The application must handle the lack of these features.

For example, an application can turn off locking by configuring the BackingMap locking strategy to be NONE. This strategy is fast, but concurrent transactions can now modify the same data with no protection from each other. The application is responsible for all locking and data consistency when NONE is used.

An application can also change the way objects are copied when accessed by the transaction. The application can specify how objects are copied with the `ObjectMap.setCopyMode` method. With this method, you can turn off CopyMode. Turning off CopyMode is normally used for read-only transactions if different values can be returned for the same object within a transaction. Different values can be returned for the same object within a transaction.

For example, if the transaction called the `ObjectMap.get` method for the object at T1, it got the value at that point in time. If it calls the `get` method again within that transaction at a later time T2, another thread might have changed the value. Because the value was changed by another thread, the application sees a different value. If the application modifies an object retrieved using a NONE CopyMode value, it is changing the committed copy of that object directly. Rolling back the transaction has no meaning in this mode. You are changing the only copy in the ObjectGrid. Although using the NONE CopyMode is fast, be aware of its consequences. An application that uses a NONE CopyMode must never roll back the transaction. If the application rolls back the transaction, the indexes are not updated with the changes *and* the changes are not replicated if replication is turned on. The default values are easy to use and less prone to errors. If you start trading performance in exchange for less reliable data, the application needs to be aware of what it is doing to avoid unintended problems.

CAUTION:

Be careful when you are changing either the locking or the CopyMode values. If you change the values, unpredictable application behavior occurs.

Stored data interaction

After you obtain a session, you can use the following code fragment to use the Map API for inserting data.

```
Session session = ...;
ObjectMap personMap = session.getMap("PERSON");
session.begin();
Person p = new Person();
p.name = "John Doe";
personMap.insert(p.name, p);
session.commit();
```

The same example using the EntityManager API follows. This code sample assumes that the Person object is mapped to an Entity.

```
Session session = ...;
EntityManager em = session.getEntityManager();
session.begin();
Person p = new Person();
p.name = "John Doe";
em.persist(p);
session.commit();
```

The pattern is designed to obtain references to the ObjectMaps for the Maps that the thread works with, start a transaction, work with the data, then commit the transaction.

The ObjectMap interface has the typical Map operations such as put, get, and remove. However, use the more specific operation names such as: get, getForUpdate, insert, update, and remove. These method names convey the intent more precisely than the traditional Map APIs.

You can also use the indexing support, which is flexible.

The following is an example for updating an Object:

```
session.begin();
Person p = (Person)personMap.getForUpdate("John Doe");
p.name = "John Doe";
p.age = 30;
personMap.update(p.name, p);
session.commit();
```

The application normally uses the getForUpdate method rather than a simple get to lock the record. The update method must be called to actually provide the updated value to the Map. If update is not called then the Map is unchanged. The following is the same fragment using the EntityManager API:

```
session.begin();
Person p = (Person)em.findForUpdate(Person.class, "John Doe");
p.age = 30;
session.commit();
```

The EntityManager API is simpler than the Map approach. In this case, eXtreme Scale finds the Entity and returns a managed object to the application. The application modifies the object and commits the transaction, and eXtreme Scale tracks changes to managed objects automatically at commit time and performs the necessary updates.

Transactions and partitions

WebSphere eXtreme Scale transactions can update a single partition. Transactions from a client can read from multiple partitions, but they can update one partition only. If an application attempts to update two partitions, then the transaction fails and is rolled back. A transaction that is using an embedded ObjectGrid (grid logic) has no routing capability and can see data in the local partition only. This business logic can always get a second session that is a true client session to access other partitions. However, this transaction would be an independent transaction.

Queries and partitions

If a transaction has already searched for an Entity, the transaction is associated with the partition for that Entity. Any queries that run on a transaction that is associated with an Entity are routed to the associated partition.

If a query is run on a transaction before it is associated with a partition, you must set the partition ID to use for the query. The partition ID is an integer value. The query is then routed to that partition.

Queries only search within a single partition. However, you can use the DataGrid APIs to run the same query in parallel on all partitions or a subset of partitions. Use the DataGrid APIs to find an entry that might be in any partition.

The REST data service allows any HTTP client to access a WebSphere eXtreme Scale grid, and is compatible with WCF Data Services in the Microsoft .NET Framework 3.5 SP1. For more information, see Configuring REST data services.

Transactions:

Transactions have many advantages for data storage and manipulation. You can use transactions to protect the data grid from concurrent changes, to apply multiple changes as a concurrent unit, to replicate data and to implement a life cycle for locks on changes.

When a transaction starts, WebSphere eXtreme Scale allocates a special difference map to hold the current changes or copies of key and value pairs that the transaction uses. Typically, when a key and value pair is accessed, the value is copied before the application receives the value. The difference map tracks all changes for operations such as insert, update, get, remove, and so on. Keys are not copied because they are assumed to be immutable. If an ObjectTransformer object is specified, then this object is used for copying the value. If the transaction is using optimistic locking, then before images of the values are also tracked for comparison when the transaction commits.

If a transaction is rolled back, then the difference map information is discarded, and locks on entries are released. When a transaction commits, the changes are applied to the maps and locks are released. If optimistic locking is being used, then eXtreme Scale compares the before image versions of the values with the values that are in the map. These values must match for the transaction to commit. This comparison enables a multiple version locking scheme, but at a cost of two copies being made when the transaction accesses the entry. All values are copied again and the new copy is stored in the map. WebSphere eXtreme Scale performs this copy to protect itself against the application changing the application reference to the value after a commit.

You can avoid using several copies of the information. The application can save a copy by using pessimistic locking instead of optimistic locking as the cost of limiting concurrency. The copy of the value at commit time can also be avoided if the application agrees not to change a value after a commit.

Advantages of transactions

Use transactions for the following reasons:

By using transactions, you can:

- Roll back changes if an exception occurs or business logic needs to undo state changes.
- To apply multiple changes as an atomic unit at commit time.
- Hold and release locks on data to apply multiple changes as an atomic unit at commit time.
- Protect a thread from concurrent changes.
- Implement a life cycle for locks on changes.
- Produce an atomic unit of replication.

Transaction size

Larger transactions are more efficient, especially for replication. However, larger transactions can adversely impact concurrency because the locks on entries are held for a longer period of time. If you use larger transactions, you can increase

replication performance. This performance increase is important when you are pre-loading a Map. Experiment with different batch sizes to determine what works best for your scenario.

Larger transactions also help with loaders. If a loader is being used that can perform SQL batching, then significant performance gains are possible depending on the transaction and significant load reductions on the database side. This performance gain depends on the Loader implementation.

Automatic commit mode

If no transaction is actively started, then when an application interacts with an ObjectMap object, an automatic begin and commit operation is done on behalf of the application. This automatic begin and commit operation works, but prevents rollback and locking from working effectively. Synchronous replication speed is impacted because of the very small transaction size. If you are using an entity manager application, then do not use automatic commit mode because objects that are looked up with the EntityManager.find method immediately become unmanaged on the method return and become unusable.

External transaction coordinators

Typically, transactions begin with the session.begin method and end with the session.commit method. However, when eXtreme Scale is embedded, the transactions might be started and ended by an external transaction coordinator. If you are using an external transaction coordinator, you do not need to call the session.begin method and end with the session.commit method.

Java EE transaction integration

eXtreme Scale includes a Java Connector Architecture (JCA) 1.5 compliant resource adapter that supports both client connections to a remote data grid and local transaction management. Java Platform, Enterprise Edition (Java EE) applications such as servlets, JavaServer Pages (JSP) files and Enterprise JavaBeans (EJB) components can demarcate eXtreme Scale transactions using the standard javax.resource.cci.LocalTransaction interface or the eXtreme Scale session interface.

When the running in WebSphere Application Server with last participant support enabled in the application, you can enlist the eXtreme Scale transaction in a global transaction with other two-phase commit transactional resources.

CopyMode attribute:

You can tune the number of copies by defining the CopyMode attribute of the BackingMap or ObjectMap objects in the ObjectGrid descriptor XML file.

You can tune the number of copies by defining the CopyMode attribute of the BackingMap or ObjectMap objects. The copy mode has the following values:

- COPY_ON_READ_AND_COMMIT
- COPY_ON_READ
- NO_COPY
- COPY_ON_WRITE
- COPY_TO_BYTES
- COPY_TO_BYTES_RAW

The `COPY_ON_READ_AND_COMMIT` value is the default. The `COPY_ON_READ` value copies on the initial data retrieved, but does not copy at commit time. This mode is safe if the application does not modify a value after committing a transaction. The `NO_COPY` value does not copy data, which is only safe for read-only data. If the data never changes then you do not need to copy it for isolation reasons.

Be careful when you use the `NO_COPY` attribute value with maps that can be updated. WebSphere eXtreme Scale uses the copy on first touch to allow the transaction rollback. The application only changed the copy, and as a result, eXtreme Scale discards the copy. If the `NO_COPY` attribute value is used, and the application modifies the committed value, completing a rollback is not possible. Modifying the committed value leads to problems with indexes, replication, and so on because the indexes and replicas update when the transaction commits. If you modify committed data and then roll back the transaction, which does not actually roll back at all, then the indexes are not updated and replication does not take place. Other threads can see the uncommitted changes immediately, even if they have locks. Use the `NO_COPY` attribute value for read-only maps or for applications that complete the appropriate copy before modifying the value. If you use the `NO_COPY` attribute value and call IBM support with a data integrity problem, you are asked to reproduce the problem with the copy mode set to `COPY_ON_READ_AND_COMMIT`.

The `COPY_TO_BYTES` value stores values in the map in a serialized form. At read time, eXtreme Scale inflates the value from a serialized form and at commit time it stores the value to a serialized form. With this method, a copy occurs at both read and commit time.

The default copy mode for a map can be configured on the `BackingMap` object. You can also change the copy mode on maps before you start a transaction by using the `ObjectMap.setCopyMode` method.

An example of a backing map snippet from an `objectgrid.xml` file that shows how to set the copy mode for a given backing map follows. This example assumes that you are using `cc` as the `objectgrid/config` namespace.

```
<cc:backingMap name="RuntimeLifespan" copyMode="NO_COPY"/>
```

Lock manager:

When you configure a locking strategy, a lock manager is created for the backing map to maintain cache entry consistency.

Lock manager configuration

When either a `PESSIMISTIC` or an `OPTIMISTIC` lock strategy is used, a lock manager is created for the `BackingMap`. The lock manager uses a hash map to track entries that are locked by one or more transactions. If many map entries exist in the hash map, more lock buckets can result in better performance. The risk of Java synchronization collisions is lower as the number of buckets grows. More lock buckets also lead to more concurrency. The previous examples show how an application can set the number of lock buckets to use for a given `BackingMap` instance.

To avoid a `java.lang.IllegalStateException` exception, the `setNumberOfLockBuckets` method must be called before calling the `initialize` or `getSession` methods on the `ObjectGrid` instance. The `setNumberOfLockBuckets` method parameter is a Java

primitive integer that specifies the number of lock buckets to use. Using a prime number can allow for a uniform distribution of map entries over the lock buckets. A good starting point for best performance is to set the number of lock buckets to about 10 percent of the expected number of BackingMap entries.

Locking strategies:

Locking strategies include pessimistic, optimistic and none. To choose a locking strategy, you must consider issues such as the percentage of each type of operations you have, whether or not you use a loader and so on.

Locks are bound by transactions. You can specify the following locking settings:

- **No locking:** Running without the locking setting is the fastest. If you are using read-only data, then you might not need locking.
- **Pessimistic locking:** Acquires locks on entries, then and holds the locks until commit time. This locking strategy provides good consistency at the expense of throughput.
- **Optimistic locking:** Takes a before image of every record that the transaction touches and compares the image to the current entry values when the transaction commits. If the entry values change, then the transaction rolls back. No locks are held until commit time. This locking strategy provides better concurrency than the pessimistic strategy, at the risk of the transaction rolling back and the memory cost of making the extra copy of the entry.

Set the locking strategy on the BackingMap. You cannot change the locking strategy for each transaction. An example XML snippet that shows how to set the locking mode on a map using the XML file follows, assuming cc is the namespace for the objectgrid/config namespace:

```
<cc:backingMap name="RuntimeLifespan" lockStrategy="PESSIMISTIC" />
```

Pessimistic locking

Use the pessimistic locking strategy for read and write maps when other locking strategies are not possible. When an ObjectGrid map is configured to use the pessimistic locking strategy, a pessimistic transaction lock for a map entry is obtained when a transaction first gets the entry from the BackingMap. The pessimistic lock is held until the application completes the transaction. Typically, the pessimistic locking strategy is used in the following situations:

- When the BackingMap is configured with or without a loader and versioning information is not available.
- When the BackingMap is used directly by an application that needs help from the eXtreme Scale for concurrency control.
- When versioning information is available, but update transactions frequently collide on the backing entries, resulting in optimistic update failures.

Because the pessimistic locking strategy has the greatest impact on performance and scalability, this strategy should only be used for read and write maps when other locking strategies are not viable. For example, these situations might include when optimistic update failures occur frequently, or when recovery from optimistic failure is difficult for an application to handle.

Optimistic locking

The optimistic locking strategy assumes that no two transactions might attempt to update the same map entry while running concurrently. Because of this belief, the

lock mode does not need to be held for the life cycle of the transaction because it is unlikely that more than one transaction might update the map entry concurrently. The optimistic locking strategy is typically used in the following situations:

- When a BackingMap is configured with or without a loader and versioning information is available.
- When a BackingMap has mostly transactions that perform read operations. Insert, update, or remove operations on map entries do not occur often on the BackingMap.
- When a BackingMap is inserted, updated, or removed more frequently than it is read, but transactions rarely collide on the same map entry.

Like the pessimistic locking strategy, the methods on the ObjectMap interface determine how eXtreme Scale automatically attempts to acquire a lock mode for the map entry that is being accessed. However, the following differences between the pessimistic and optimistic strategies exist:

- Like the pessimistic locking strategy, an S lock mode is acquired by the get and getAll methods when the method is invoked. However, with optimistic locking, the S lock mode is not held until the transaction is completed. Instead, the S lock mode is released before the method returns to the application. The purpose of acquiring the lock mode is so that eXtreme Scale can ensure that only committed data from other transactions is visible to the current transaction. After eXtreme Scale has verified that the data is committed, the S lock mode is released. At commit time, an optimistic versioning check is performed to ensure that no other transaction has changed the map entry after the current transaction released its S lock mode. If an entry is not fetched from the map before it is updated, invalidated, or deleted, the eXtreme Scale run time implicitly fetches the entry from the map. This implicit get operation is performed to get the current value at the time the entry was requested to be modified.
- Unlike pessimistic locking strategy, the getForUpdate and getAllForUpdate methods are handled exactly like the get and getAll methods when the optimistic locking strategy is used. That is, an S lock mode is acquired at the start of the method and the S lock mode is released before returning to the application.

All other ObjectMap methods are handled exactly like they are handled for the pessimistic locking strategy. That is, when the commit method is invoked, an X lock mode is obtained for any map entry that is inserted, updated, removed, touched, or invalidated and the X lock mode is held until the transaction completes commit processing.

The optimistic locking strategy assumes that no concurrently running transactions attempt to update the same map entry. Because of this assumption, the lock mode does not need to be held for the life of the transaction because it is unlikely that more than one transaction might update the map entry concurrently. However, because a lock mode was not held, another concurrent transaction might potentially update the map entry after the current transaction has released its S lock mode.

To handle this possibility, eXtreme Scale gets an X lock at commit time and performs an optimistic versioning check to verify that no other transaction has changed the map entry after the current transaction read the map entry from the BackingMap. If another transaction changes the map entry, the version check fails and an OptimisticCollisionException exception occurs. This exception forces the current transaction to be rolled back and the application must try the entire

transaction again. The optimistic locking strategy is very useful when a map is mostly read and it is unlikely that updates for the same map entry might occur.

No locking

When a BackingMap is configured to use no locking strategy, no transaction locks for a map entry are obtained.

Using no locking strategy is useful when an application is a persistence manager such as an Enterprise JavaBeans (EJB) container or when an application uses Hibernate to obtain persistent data. In this scenario, the BackingMap is configured without a loader and the persistence manager uses the BackingMap as a data cache. In this scenario, the persistence manager provides concurrency control between transactions that are accessing the same Map entries.

WebSphere eXtreme Scale does not need to obtain any transaction locks for the purpose of concurrency control. This situation assumes that the persistence manager does not release its transaction locks before updating the ObjectGrid map with committed changes. If the persistence manager releases its locks, then a pessimistic or optimistic lock strategy must be used. For example, suppose that the persistence manager of an EJB container is updating an ObjectGrid map with data that was committed in the EJB container-managed transaction. If the update of the ObjectGrid map occurs before the persistence manager transaction locks are released, then you can use the no lock strategy. If the ObjectGrid map update occurs after the persistence manager transaction locks are released, then you must use either the optimistic or pessimistic lock strategy.

Another scenario where no locking strategy can be used is when the application uses a BackingMap directly and a Loader is configured for the map. In this scenario, the loader uses the concurrency control support that is provided by a relational database management system (RDBMS) by using either Java database connectivity (JDBC) or Hibernate to access data in a relational database. The loader implementation can use either an optimistic or pessimistic approach. A loader that uses an optimistic locking or versioning approach helps to achieve the greatest amount of concurrency and performance. For more information about implementing an optimistic locking approach, see the *OptimisticCallback* section in the information about loader considerations in the *Administration Guide*. If you are using a loader that uses pessimistic locking support of an underlying backend, you might want to use the `forUpdate` parameter that is passed on the `get` method of the Loader interface. Set this parameter to true if the `getForUpdate` method of the ObjectMap interface was used by the application to get the data. The loader can use this parameter to determine whether to request an upgradeable lock on the row that is being read. For example, DB2[®] obtains an upgradeable lock when an SQL select statement contains a FOR UPDATE clause. This approach offers the same deadlock prevention that is described in “Pessimistic locking” on page 221.

Distributing transactions:

Use Java Message Service (JMS) for distributed transaction changes between different tiers or in environments on mixed platforms.

JMS is an ideal protocol for distributed changes between different tiers or in environments on mixed platforms. For example, some applications that use eXtreme Scale might be deployed on IBM WebSphere Application Server Community Edition, Apache Geronimo, or Apache Tomcat, whereas other applications might run on WebSphere Application Server Version 6.x. JMS is ideal

for distributed changes between eXtreme Scale peers in these different environments. The high availability manager message transport is very fast, but can only distribute changes to Java virtual machines that are in a single core group. JMS is slower, but allows larger and more diverse sets of application clients to share an ObjectGrid. JMS is ideal when sharing data in an ObjectGrid between a fat Swing client and an application deployed on WebSphere Extended Deployment.

The built-in Client Invalidation Mechanism and Peer-to-Peer Replication are examples of JMS-based transactional changes distribution. See the information about configuring peer-to-peer replication with JMS in the *Administration Guide* for more information.

Implementing JMS

JMS is implemented for distributing transaction changes by using a Java object that behaves as an ObjectGridEventListener. This object can propagate the state in the following four ways:

1. Invalidate: Any entry that is evicted, updated or deleted is removed on all peer Java virtual machines when they receive the message.
2. Invalidate conditional: The entry is evicted only if the local version is the same or older than the version on the publisher.
3. Push: Any entry that was evicted, updated, deleted or inserted is added or overwritten on all peer Java virtual machines when they receive the JMS message.
4. Push conditional: The entry is only updated or added on the receive side if the local entry is less recent than the version that is being published.

Listen for changes for publishing

The plug-in implements the ObjectGridEventListener interface to intercept the transactionEnd event. When eXtreme Scale invokes this method, the plug-in attempts to convert the LogSequence list for each map that is touched by the transaction to a JMS message and then publish it. The plug-in can be configured to publish changes for all maps or a subset of maps. LogSequence objects are processed for the maps that have publishing enabled. The LogSequenceTransformer ObjectGrid class serializes a filtered LogSequence for each map to a stream. After all LogSequences are serialized to the stream, then a JMS ObjectMessage is created and published to a well-known topic.

Listen for JMS messages and apply them to the local ObjectGrid

The same plug-in also starts a thread that spins in a loop, receiving all messages that are published to the well known topic. When a message arrives, it passes the message contents to the LogSequenceTransformer class where it is converted to a set of LogSequence objects. Then, a no-write-through transaction is started. Each LogSequence object is provided to the Session.processLogSequence method, which updates the local Maps with the changes. The processLogSequence method understands the distribution mode. The transaction is committed and the local cache now reflects the changes. For more information about using JMS to distribute transaction changes, see the information about distributing changes between peer Java Virtual Machines in the *Administration Guide*.

Single-partition and cross-data-grid transactions:

The major distinction between WebSphere eXtreme Scale and traditional data storage solutions like relational databases or in-memory databases is the use of partitioning, which allows the cache to scale linearly. The important types of transactions to consider are single-partition and every-partition (cross-data-grid) transactions.

In general, interactions with the cache can be categorized as single-partition transactions or cross-data-grid transactions, as discussed in the following section.

Single-partition transactions

Single-partition transactions are the preferable method for interacting with caches that are hosted by WebSphere eXtreme Scale. When a transaction is limited to a single partition, then by default it is limited to a single Java virtual machine, and therefore a single server computer. A server can complete M number of these transactions per second, and if you have N computers, you can complete $M*N$ transactions per second. If your business increases and you need to perform twice as many of these transactions per second, you can double N by buying more computers. Then you can meet capacity demands without changing the application, upgrading hardware, or even taking the application offline.

In addition to letting the cache scale so significantly, single-partition transactions also maximize the availability of the cache. Each transaction only depends on one computer. Any of the other $(N-1)$ computers can fail without affecting the success or response time of the transaction. So if you are running 100 computers and one of them fails, only 1 percent of the transactions in flight at the moment that server failed are rolled back. After the server fails, WebSphere eXtreme Scale relocates the partitions that are hosted by the failed server to the other 99 computers. During this brief period, before the operation completes, the other 99 computers can still complete transactions. Only the transactions that would involve the partitions that are being relocated are blocked. After the failover process is complete, the cache can continue running, fully operational, at 99 percent of its original throughput capacity. After the failed server is replaced and returned to the data grid, the cache returns to 100 percent throughput capacity.

Cross-data-grid transactions

In terms of performance, availability and scalability, cross-data-grid transactions are the opposite of single-partition transactions. Cross-data-grid transactions access every partition and therefore every computer in the configuration. Each computer in the data grid is asked to look up some data and then return the result. The transaction cannot complete until every computer has responded, and therefore the throughput of the entire data grid is limited by the slowest computer. Adding computers does not make the slowest computer faster and therefore does not improve the throughput of the cache.

Cross-data-grid transactions have a similar effect on availability. Extending the previous example, if you are running 100 servers and one server fails, then 100 percent of the transactions that are in progress at the moment that server failed are rolled back. After the server fails, WebSphere eXtreme Scale starts to relocate the partitions that are hosted by that server to the other 99 computers. During this time, before the failover process completes, the data grid cannot process any of these transactions. After the failover process is complete, the cache can continue running, but at reduced capacity. If each computer in the data grid serviced 10 partitions, then 10 of the remaining 99 computers receive at least one extra partition as part of the failover process. Adding an extra partition increases the

workload of that computer by at least 10 percent. Because the throughput of the data grid is limited to the throughput of the slowest computer in a cross-data-grid transaction, on average, the throughput is reduced by 10 percent.

Single-partition transactions are preferable to cross-data-grid transactions for scaling out with a distributed, highly available, object cache like WebSphere eXtreme Scale. Maximizing the performance of these kinds of systems requires the use of techniques that are different from traditional relational methodologies, but you can turn cross-data-grid transactions into scalable single-partition transactions.

Best practices for building scalable data models

The best practices for building scalable applications with products like WebSphere eXtreme Scale include two categories: foundational principles and implementation tips. Foundational principles are core ideas that need to be captured in the design of the data itself. An application that does not observe these principles is unlikely to scale well, even for its mainline transactions. Implementation tips are applied for problematic transactions in an otherwise well-designed application that observes the general principles for scalable data models.

Foundational principles

Some of the important means of optimizing scalability are basic concepts or principles to keep in mind.

Duplicate instead of normalizing

The key thing to remember about products like WebSphere eXtreme Scale is that they are designed to spread data across a large number of computers. If the goal is to make most or all transactions complete on a single partition, then the data model design needs to ensure that all the data the transaction might need is located in the partition. Most of the time, the only way to achieve this is by duplicating data.

For example, consider an application like a message board. Two very important transactions for a message board are showing all the posts from a given user and all the posts on a given topic. First consider how these transactions would work with a normalized data model that contains a user record, a topic record, and a post record that contains the actual text. If posts are partitioned with user records, then displaying the topic becomes a cross-grid transaction, and vice versa. Topics and users cannot be partitioned together because they have a many-to-many relationship.

The best way to make this message board scale is to duplicate the posts, storing one copy with the topic record and one copy with the user record. Then, displaying the posts from a user is a single-partition transaction, displaying the posts on a topic is a single-partition transaction, and updating or deleting a post is a two-partition transaction. All three of these transactions will scale linearly as the number of computers in the data grid increases.

Scalability rather than resources

The biggest obstacle to overcome when considering denormalized data models is the impact that these models have on resources. Keeping two, three, or more copies of some data can seem to use too many resources to be practical. When you are confronted with this scenario, remember the following facts: Hardware resources get cheaper every year. Second, and

more importantly, WebSphere eXtreme Scale eliminates most hidden costs associated with deploying more resources.

Measure resources in terms of cost rather than computer terms such as megabytes and processors. Data stores that work with normalized relational data generally need to be located on the same computer. This required collocation means that a single larger enterprise computer needs to be purchased rather than several smaller computers. With enterprise hardware, it is not uncommon for one computer to be capable of completing one million transactions per second to cost much more than the combined cost of 10 computers capable of doing 100,000 transactions per second each.

A business cost in adding resources also exists. A growing business eventually runs out of capacity. When you run out of capacity, you either need to shut down while moving to a bigger, faster computer, or create a second production environment to which you can switch. Either way, additional costs will come in the form of lost business or maintaining almost twice the capacity needed during the transition period.

With WebSphere eXtreme Scale, the application does not need to be shut down to add capacity. If your business projects that you need 10 percent more capacity for the coming year, then increase the number of computers in the data grid by 10 percent. You can increase this percentage without application downtime and without purchasing excess capacity.

Avoid data transformations

When you are using WebSphere eXtreme Scale, data should be stored in a format that is directly consumable by the business logic. Breaking the data down into a more primitive form is costly. The transformation needs to be done when the data is written and when the data is read. With relational databases this transformation is done out of necessity, because the data is ultimately persisted to disk quite frequently, but with WebSphere eXtreme Scale, you do not need to perform these transformations. For the most part data is stored in memory and can therefore be stored in the exact form that the application needs.

Observing this simple rule helps denormalize your data in accordance with the first principle. The most common type of transformation for business data is the JOIN operations that are necessary to turn normalized data into a result set that fits the needs of the application. Storing the data in the correct format implicitly avoids performing these JOIN operations and produces a denormalized data model.

Eliminate unbounded queries

No matter how well you structure your data, unbounded queries do not scale well. For example, do not have a transaction that asks for a list of all items sorted by value. This transaction might work at first when the total number of items is 1000, but when the total number of items reaches 10 million, the transaction returns all 10 million items. If you run this transaction, the two most likely outcomes are the transaction timing out, or the client encountering an out-of-memory error.

The best option is to alter the business logic so that only the top 10 or 20 items can be returned. This logic alteration keeps the size of the transaction manageable no matter how many items are in the cache.

Define schema

The main advantage of normalizing data is that the database system can take care of data consistency behind the scenes. When data is denormalized for scalability, this automatic data consistency management no longer exists. You must implement a data model that can work in the application layer or as a plug-in to the distributed data grid to guarantee data consistency.

Consider the message board example. If a transaction removes a post from a topic, then the duplicate post on the user record needs to be removed. Without a data model, it is possible a developer would write the application code to remove the post from the topic and forget to remove the post from the user record. However, if the developer were using a data model instead of interacting with the cache directly, the `removePost` method on the data model could pull the user ID from the post, look up the user record, and remove the duplicate post behind the scenes.

Alternately, you can implement a listener that runs on the actual partition that detects the change to the topic and automatically adjusts the user record. A listener might be beneficial because the adjustment to the user record could happen locally if the partition happens to have the user record, or even if the user record is on a different partition, the transaction takes place between servers instead of between the client and server. The network connection between servers is likely to be faster than the network connection between the client and the server.

Avoid contention

Avoid scenarios such as having a global counter. The data grid will not scale if a single record is being used a disproportionate number of times compared to the rest of the records. The performance of the data grid will be limited by the performance of the computer that holds the given record.

In these situations, try to break the record up so it is managed per partition. For example consider a transaction that returns the total number of entries in the distributed cache. Instead of having every insert and remove operation access a single record that increments, have a listener on each partition track the insert and remove operations. With this listener tracking, insert and remove can become single-partition operations.

Reading the counter will become a cross-data-grid operation, but for the most part, it was already as inefficient as a cross-data-grid operation because its performance was tied to the performance of the computer hosting the record.

Implementation tips

You can also consider the following tips to achieve the best scalability.

Use reverse-lookup indexes

Consider a properly denormalized data model where customer records are partitioned based on the customer ID number. This partitioning method is the logical choice because nearly every business operation performed with the customer record uses the customer ID number. However, an important transaction that does not use the customer ID number is the login transaction. It is more common to have user names or e-mail addresses for login instead of customer ID numbers.

The simple approach to the login scenario is to use a cross-data-grid transaction to find the customer record. As explained previously, this approach does not scale.

The next option might be to partition on user name or e-mail. This option is not practical because all the customer ID based operations become cross-data-grid transactions. Also, the customers on your site might want to change their user name or e-mail address. Products like WebSphere eXtreme Scale need the value that is used to partition the data to remain constant.

The correct solution is to use a reverse lookup index. With WebSphere eXtreme Scale, a cache can be created in the same distributed grid as the cache that holds all the user records. This cache is highly available, partitioned and scalable. This cache can be used to map a user name or e-mail address to a customer ID. This cache turns login into a two partition operation instead of a cross-grid operation. This scenario is not as good as a single-partition transaction, but the throughput still scales linearly as the number of computers increases.

Compute at write time

Commonly calculated values like averages or totals can be expensive to produce because these operations usually require reading a large number of entries. Because reads are more common than writes in most applications, it is efficient to compute these values at write time and then store the result in the cache. This practice makes read operations both faster and more scalable.

Optional fields

Consider a user record that holds a business, home, and telephone number. A user could have all, none or any combination of these numbers defined. If the data were normalized then a user table and a telephone number table would exist. The telephone numbers for a given user could be found using a JOIN operation between the two tables.

De-normalizing this record does not require data duplication, because most users do not share telephone numbers. Instead, empty slots in the user record must be allowed. Instead of having a telephone number table, add three attributes to each user record, one for each telephone number type. This addition of attributes eliminates the JOIN operation and makes a telephone number lookup for a user a single-partition operation.

Placement of many-to-many relationships

Consider an application that tracks products and the stores in which the products are sold. A single product is sold in many stores, and a single store sells many products. Assume that this application tracks 50 large retailers. Each product is sold in a maximum of 50 stores, with each store selling thousands of products.

Keep a list of stores inside the product entity (arrangement A), instead of keeping a list of products inside each store entity (arrangement B). Looking at some of the transactions this application would have to perform illustrates why arrangement A is more scalable.

First look at updates. With arrangement A, removing a product from the inventory of a store locks the product entity. If the data grid holds 10000 products, only 1/10000 of the grid needs to be locked to perform the update. With arrangement B, the data grid only contains 50 stores, so 1/50

of the grid must be locked to complete the update. So even though both of these could be considered single-partition operations, arrangement A scales out more efficiently.

Now, considering reads with arrangement A, looking up the stores at which a product is sold is a single-partition transaction that scales and is fast because the transaction only transmits a small amount of data. With arrangement B, this transaction becomes a cross-data-grid transaction because each store entity must be accessed to see if the product is sold at that store, which reveals an enormous performance advantage for arrangement A.

Scaling with normalized data

One legitimate use of cross-data-grid transactions is to scale data processing. If a data grid has 5 computers and a cross-data-grid transaction is dispatched that sorts through about 100,000 records on each computer, then that transaction sorts through 500,000 records. If the slowest computer in the data grid can perform 10 of these transactions per second, then the data grid is capable of sorting through 5,000,000 records per second. If the data in the grid doubles, then each computer must sort through 200,000 records, and each transaction sorts through 1,000,000 records. This data increase decreases the throughput of the slowest computer to 5 transactions per second, thereby reducing the throughput of the data grid to 5 transactions per second. Still, the data grid sorts through 5,000,000 records per second.

In this scenario, doubling the number of computer allows each computer to return to its previous load of sorting through 100,000 records, allowing the slowest computer to process 10 of these transactions per second. The throughput of the data grid stays the same at 10 requests per second, but now each transaction processes 1,000,000 records, so the grid has doubled its capacity in terms of processing records to 10,000,000 per second.

Applications such as a search engine that need to scale both in terms of data processing to accommodate the increasing size of the Internet and throughput to accommodate growth in the number of users, you must create multiple data grids, with a round robin of the requests between the grids. If you need to scale up the throughput, add computers and add another data grid to service requests. If data processing needs to be scaled up, add more computers and keep the number of data grids constant.

Developing eXtreme Scale client components to use transactions

The WebSphere eXtreme Scale resource adapter provides client connection management and local transaction support. With this support, Java Platform, Enterprise Edition (Java EE) applications can look up eXtreme Scale client connections and demarcate local transactions with Java EE local transactions or the eXtreme Scale APIs.

Before you begin

Create an eXtreme Scale connection factory resource reference.

About this task

There are several options for working with eXtreme Scale data access APIs. In all cases, the eXtreme Scale connection factory must be injected into the application component, or looked up in Java Naming Directory Interface (JNDI). After the

connection factory is looked up, you can demarcate transactions and create connections to access the eXtreme Scale APIs.

You can optionally cast the `javax.resource.cci.ConnectionFactory` instance to a `com.ibm.websphere.xs.ra.XSConnectionFactory` that provides additional options for retrieving connection handles. The resulting connection handles must be cast to the `com.ibm.websphere.xs.ra.XSConnection` interface, which provides the `getSession` method. The `getSession` method returns a `com.ibm.websphere.objectgrid.Session` object handle that allows applications to use any of the eXtreme Scale data access APIs, such as the ObjectMap API and EntityManager API.

The Session handle and any derived objects are valid for the life of the `XSConnection` handle.

The following procedures can be used to demarcate eXtreme Scale transactions. You cannot mix each of the procedures. For example, you cannot mix global transaction demarcation and local transaction demarcation in the same application component context.

Procedure

- Use autocommit, local transactions. Use the following steps to automatically commit data access operations or operations that do not support an active transaction:
 1. Retrieve a `com.ibm.websphere.xs.ra.XSConnection` connection outside of the context of a global transaction.
 2. Retrieve and use the `com.ibm.websphere.objectgrid.Session` session to interact with the data grid.
 3. Invoke any data access operation that supports autocommit transactions.
 4. Close the connection.
- Use an ObjectGrid session to demarcate a local transaction. Use the following steps to demarcate an ObjectGrid transaction using the Session object:
 1. Retrieve a `com.ibm.websphere.xs.ra.XSConnection` connection.
 2. Retrieve the `com.ibm.websphere.objectgrid.Session` session.
 3. Use the `Session.begin()` method to start the transaction.
 4. Use the session to interact with the data grid.
 5. Use the `Session.commit()` or `rollback()` methods to end the transaction.
 6. Close the connection.
- Use a `javax.resource.cci.LocalTransaction` transaction to demarcate a local transaction. Use the following steps to demarcate an ObjectGrid transaction using the `javax.resource.cci.LocalTransaction` interface:
 1. Retrieve a `com.ibm.websphere.xs.ra.XSConnection` connection.
 2. Retrieve the `javax.resource.cci.LocalTransaction` transaction using the `XSConnection.getLocalTransaction()` method.
 3. Use the `LocalTransaction.begin()` method to start the transaction.
 4. Retrieve and use the `com.ibm.websphere.objectgrid.Session` session to interact with the data grid.
 5. Use the `LocalTransaction.commit()` or `rollback()` methods to end the transaction.
 6. Close the connection.
- Enlist the connection in a global transaction. This procedure also applies to container-managed transactions:

1. Begin the global transaction through the javax.transaction.UserTransaction interface or with a container-managed transaction.
2. Retrieve a com.ibm.websphere.xs.ra.XSConnection connection.
3. Retrieve and use the com.ibm.websphere.objectgrid.Session session.
4. Close the connection.
5. Commit or roll back the global transaction.

Example

See the following code example, which demonstrates the previous steps for demarcating eXtreme Scale transactions.

```
// (C) Copyright IBM Corp. 2001, 2012.
// All Rights Reserved. Licensed Materials - Property of IBM.
package com.ibm.ws.xs.ra.test.ee;

import javax.naming.InitialContext;
import javax.resource.cci.Connection;
import javax.resource.cci.ConnectionFactory;
import javax.resource.cci.LocalTransaction;
import javax.transaction.Status;
import javax.transaction.UserTransaction;

import junit.framework.TestCase;

import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.xs.ra.XSConnection;

/**
 * This sample requires that it runs in a J2EE context in your
 * application server. For example, using the JUnitEE framework servlet.
 *
 * The code in these test methods would typically reside in your own servlet,
 * EJB, or other web component.
 *
 * The sample depends on a configured WebSphere eXtreme Scale connection
 * factory registered at of JNDI Name of "eis/embedded/wxscf" that defines
 * a connection to a grid containing a Map with the name "Map1".
 *
 * The sample does a direct lookup of the JNDI name and does not require
 * resource injection.
 */
public class DocSampleTests extends TestCase {
    public final static String CF_JNDI_NAME = "eis/embedded/wxscf";
    public final static String MAP_NAME = "Map1";

    Long          key = null;
    Long          value = null;
    InitialContext ctx = null;
    ConnectionFactory cf = null;

    public DocSampleTests() {
    }
    public DocSampleTests(String name) {
        super(name);
    }
    protected void setUp() throws Exception {
        ctx = new InitialContext();
        cf = (ConnectionFactory)ctx.lookup(CF_JNDI_NAME);
        key = System.nanoTime();
        value = System.nanoTime();
    }
}
/**
 * This example runs when not in the context of a global transaction
 * and uses autocommit.
 */
public void testLocalAutocommit() throws Exception {
    Connection conn = cf.getConnection();
    try {
        Session session = ((XSConnection)conn).getSession();
        ObjectMap map = session.getMap(MAP_NAME);
        map.insert(key, value); // Or various data access operations
    }
}
```

```

    }
    finally {
        conn.close();
    }
}

/**
 * This example runs when not in the context of a global transaction
 * and demarcates the transaction using session.begin()/session.commit()
 */
public void testLocalSessionTransaction() throws Exception {
    Session session = null;
    Connection conn = cf.getConnection();
    try {
        session = ((XSConnection)conn).getSession();
        session.begin();
        ObjectMap map = session.getMap(MAP_NAME);
        map.insert(key, value); // Or various data access operations
        session.commit();
    }
    finally {
        if (session != null && session.isTransactionActive()) {
            try { session.rollback(); }
            catch (Exception e) { e.printStackTrace(); }
        }
        conn.close();
    }
}

/**
 * This example uses the LocalTransaction interface to demarcate
 * transactions.
 */
public void testLocalTranTransaction() throws Exception {
    LocalTransaction tx = null;
    Connection conn = cf.getConnection();
    try {
        tx = conn.getLocalTransaction();
        tx.begin();
        Session session = ((XSConnection)conn).getSession();
        ObjectMap map = session.getMap(MAP_NAME);
        map.insert(key, value); // Or various data access operations
        tx.commit(); tx = null;
    }
    finally {
        if (tx != null) {
            try { tx.rollback(); }
            catch (Exception e) { e.printStackTrace(); }
        }
        conn.close();
    }
}

/**
 * This example depends on an externally managed transaction,
 * the externally managed transaction might typically be present in
 * an EJB with its transaction attributes set to REQUIRED or REQUIRES_NEW.
 * NOTE: If there is NO global transaction active, this example runs in auto-commit
 * mode because it doesn't verify a transaction exists.
 */
public void testGlobalTransactionContainerManaged() throws Exception {
    Connection conn = cf.getConnection();
    try {
        Session session = ((XSConnection)conn).getSession();
        ObjectMap map = session.getMap(MAP_NAME);
        map.insert(key, value); // Or various data access operations
    }
    catch (Throwable t) {
        t.printStackTrace();
        UserTransaction tx = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
        if (tx.getStatus() != Status.STATUS_NO_TRANSACTION) {
            tx.setRollbackOnly();
        }
    }
    finally {
        conn.close();
    }
}
}

```

```

/**
 * This example demonstrates starting a new global transaction using the
 * UserTransaction interface. Typically the container starts the global
 * transaction (for example in an EJB with a transaction attribute of
 * REQUIRES_NEW), but this sample will also start the global transaction
 * using the UserTransaction API if it is not currently active.
 */
public void testGlobalTransactionTestManaged() throws Exception {
    boolean        started = false;
    UserTransaction tx = (UserTransaction)ctx.lookup("java:comp/UserTransaction");
    if (tx.getStatus() == Status.STATUS_NO_TRANSACTION) {
        tx.begin();
        started = true;
    }
    // else { called with an externally/container managed transaction }
    Connection     conn = null;
    try {
        conn = cf.getConnection(); // Get connection after the global tran starts
        Session     session = ((XSConnection)conn).getSession();
        ObjectMap   map = session.getMap(MAP_NAME);
        map.insert(key, value); // Or various data access operations
        if (started) {
            tx.commit(); started = false; tx = null;
        }
    }
    finally {
        if (started) {
            try { tx.rollback(); }
            catch (Exception e) { e.printStackTrace(); }
        }
        if (conn != null) { conn.close(); }
    }
}
}
}

```

Using locking

Locks have life cycles and different types of locks are compatible with others in various ways. Locks must be handled in the correct order to avoid deadlock scenarios.

Locks:

Locks have life cycles and different types of locks are compatible with others in various ways. Locks must be handled in the correct order to avoid deadlock scenarios.

Shared, upgradeable, and exclusive locks

When an application calls any method of the ObjectMap interface, uses the find methods on an index, or does a query, eXtreme Scale automatically attempts to acquire a lock for the map entry that is being accessed. WebSphere eXtreme Scale uses the following lock modes based on the method the application calls in the ObjectMap interface.

- The get and getAll methods on the ObjectMap interface, index methods, and queries acquire an *S lock*, or a shared lock mode for the key of a map entry. The duration that the S lock is held depends on the transaction isolation level used. An S lock mode allows concurrency between transactions that attempt to acquire an S or an upgradeable lock (U lock) mode for the same key, but blocks other transactions that attempt to get an exclusive lock (X lock) mode for the same key.
- The getForUpdate and getAllForUpdate methods acquire a *U lock*, or an upgradeable lock mode for the key of a map entry. The U lock is held until the transaction completes. A U lock mode allows concurrency between transactions that acquire an S lock mode for the same key, but blocks other transactions that attempt to acquire a U lock or X lock mode for the same key.

- The put, putAll, remove, removeAll, insert, update, and touch acquire an *X lock*, or exclusive lock mode for the key of a map entry. The X lock is held until the transaction completes. An X lock mode ensures that only one transaction is inserting, updating, or removing a map entry of a given key value. An X lock blocks all other transactions that attempt to acquire a S, U, or X lock mode for the same key.
- The global invalidate and global invalidateAll methods acquire an X lock for each map entry that is invalidated. The X lock is held until the transaction completes. No locks are acquired for the local invalidate and local invalidateAll methods because none of the BackingMap entries are invalidated by local invalidate method calls.

From the preceding definitions, it is obvious that an S lock mode is weaker than a U lock mode because it allows more transactions to run concurrently when accessing the same map entry. The U lock mode is slightly stronger than the S lock mode because it blocks other transactions that are requesting either a U or X lock mode. The S lock mode only blocks other transactions that are requesting an X lock mode. This small difference is important in preventing some deadlocks from occurring. The X lock mode is the strongest lock mode because it blocks all other transactions attempting to get an S, U, or X lock mode for the same map entry. The net effect of an X lock mode is to ensure that only one transaction can insert, update, or remove a map entry and to prevent updates from being lost when more than one transaction is attempting to update the same map entry.

The following table is a lock mode compatibility matrix that summarizes the described lock modes, which you can use to determine which lock modes are compatible with each other. To read this matrix, the row in the matrix indicates a lock mode that is already granted. The column indicates the lock mode that is requested by another transaction. If Yes is displayed in the column, then the lock mode requested by the other transaction is granted because it is compatible with the lock mode that is already granted. No indicates that the lock mode is not compatible and the other transaction must wait for the first transaction to release the lock that it owns.

Table 5. Lock mode compatibility matrix

Lock	Lock type S (shared)	Lock type U (upgradeable)	Lock type X (exclusive)	Strength
S (shared)	Yes	Yes	No	weakest
U (upgradeable)	Yes	No	No	normal
X (exclusive)	No	No	No	strongest

Locking deadlocks

Consider the following sequence of lock mode requests:

1. X lock is granted to transaction 1 for key1.
2. X lock is granted to transaction 2 for key2.
3. X lock requested by transaction 1 for key2. (Transaction 1 blocks waiting for lock owned by transaction 2.)
4. X lock requested by transaction 2 for key1. (Transaction 2 blocks waiting for lock owned by transaction 1.)

The preceding sequence is the classic deadlock example of two transactions that attempt to acquire more than a single lock, and each transaction acquires the locks in a different order. To prevent this deadlock, each transaction must obtain the multiple locks in the same order. If the OPTIMISTIC lock strategy is used and the

flush method on the ObjectMap interface is never used by the application, then lock modes are requested by the transaction only during the commit cycle. During the commit cycle, eXtreme Scale determines the keys for the map entries that need to be locked and requests the lock modes in key sequence (deterministic behavior). With this method, eXtreme Scale prevents the large majority of the classic deadlocks. However, eXtreme Scale does not and cannot prevent all possible deadlock scenarios. A few scenarios exist that the application needs to consider. Following are the scenarios that the application must be aware of and take preventative action against.

One scenario exists where eXtreme Scale is able to detect a deadlock without having to wait for a lock wait timeout to occur. If this scenario does occur, a `com.ibm.websphere.objectgrid.LockDeadlockException` exception results. Consider the following code snippet:

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
sess.begin();
Person p = (IPerson)person.get("Lynn");
// Lynn had a birthday, so we make her 1 year older.
p.setAge( p.getAge() + 1 );
person.put( "Lynn", p );
sess.commit();
```

In this situation, Lynn's boyfriend wants to make her older than she is now, and both Lynn and her boyfriend run this transaction concurrently. In this situation, both transactions own an S lock mode on the Lynn entry of the PERSON map as a result of the `person.get("Lynn")` method invocation. As a result of the `person.put("Lynn", p)` method call, both transactions attempt to upgrade the S lock mode to an X lock mode. Both transactions block waiting for the other transaction to release the S lock mode it owns. As a result, a deadlock occurs because a circular wait condition exists between the two transactions. A circular wait condition results when more than one transaction attempts to promote a lock from a weaker to a stronger mode for the same map entry. In this scenario, a `LockDeadlockException` exception results instead of a `LockTimeoutException` exception.

The application can prevent the `LockDeadlockException` exception for the preceding example by using the optimistic lock strategy instead of the pessimistic lock strategy. Using the optimistic lock strategy is the preferred solution when the map is mostly read and updates to the map are infrequent. If the pessimistic lock strategy must be used, the `getForUpdate` method can be used instead of the `get` method in the above example or a transaction isolation level of `TRANSACTION_READ_COMMITTED` can be used.

For more information, see the topic on locking strategies in the *Product Overview*.

Using the `TRANSACTION_READ_COMMITTED` transaction isolation level prevents the S lock that is acquired by the `get` method from being held until the transaction completes. If the key is never invalidated in the transactional cache, repeatable reads are still guaranteed.

See the topic on map entry locking in the *Administration Guide* for more information.

An alternative to changing the transaction isolation level is to use the `getForUpdate` method. The first transaction to call the `getForUpdate` method acquires a U lock mode instead of an S lock. This lock mode causes the second transaction to block when it calls the `getForUpdate` method because only one

transaction is granted a U lock mode. Because the second transaction is blocked, it does not own any lock mode on the Lynn map entry. The first transaction does not block when it attempts to upgrade the U lock mode to an X lock mode as a result of the put method call from the first transaction. This feature demonstrates why U lock mode is called the *upgradeable* lock mode. When the first transaction is completed, the second transaction unblocks and is granted the U lock mode. An application can prevent the lock promotion deadlock scenario by using the getForUpdate method instead of the get method when pessimistic lock strategy is being used.

Important: This solution does not prevent read-only transactions from being able to read a map entry. Read-only transactions call the get method, but never call the put, insert, update, or remove methods. Concurrency is just as high as when the regular get method is used. The only reduction in concurrency occurs when the getForUpdate method is called by more than one transaction for the same map entry.

You must be aware when a transaction calls the getForUpdate method on more than one map entry to ensure that the U locks are acquired in the same order by each transaction. For example, suppose that the first transaction calls the getForUpdate method for the key 1 and the getForUpdate method for key 2. Another concurrent transaction calls the getForUpdate method for the same keys, but in reverse order. This sequence causes the classic deadlock because multiple locks are obtained in different orders by different transactions. The application still needs to ensure that every transaction accesses multiple map entries in key sequence to ensure that deadlock does not occur. Because the U lock is obtained at the time that the getForUpdate method is called rather than at commit time, the eXtreme Scale cannot order the lock requests like it does during the commit cycle. The application must control the lock ordering in this case.

Using the flush method on the ObjectMap interface before a commit can introduce additional lock ordering considerations. The flush method is typically used to force changes made to the map out to the backend through the Loader plug-in. In this situation, the backend uses its own lock manager to control concurrency, so the lock wait condition and deadlock can occur in backend rather than in the eXtreme Scale lock manager. Consider the following transaction:

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
boolean activeTran = false;
try
{
    sess.begin();
    activeTran = true;
    Person p = (IPerson)person.get("Lynn");
    p.setAge( p.getAge() + 1 );
    person.put( "Lynn", p );
    person.flush();
    ...
    p = (IPerson)person.get("Tom");
    p.setAge( p.getAge() + 1 );
    sess.commit();
    activeTran = false;
}
finally
{
    if ( activeTran ) sess.rollback();
}
```

Suppose that another transaction also updated the Tom person, called the flush method, and then updated the Lynn person. If this situation occurred, the following interleaving of the two transactions results in a database deadlock condition:

```
X lock is granted to transaction 1 for "Lynn" when flush is executed.
X lock is granted to transaction 2 for "Tom" when flush is executed..
X lock requested by transaction 1 for "Tom" during commit processing.
(Transaction 1 blocks waiting for lock owned by transaction 2.)
X lock requested by transaction 2 for "Lynn" during commit processing.
(Transaction 2 blocks waiting for lock owned by transaction 1.)
```

This example demonstrates that the use of the flush method can cause a deadlock to occur in the database rather than in eXtreme Scale. This deadlock example can occur regardless of what lock strategy is used. The application must take care to prevent this kind of deadlock from occurring when using the flush method and when a Loader is plugged into the BackingMap. The preceding example also illustrates another reason why eXtreme Scale has a lock wait timeout mechanism. A transaction that is waiting for a database lock might be waiting while it owns an eXtreme Scale map entry lock. Consequently, problems at database level can cause excessive wait times for an eXtreme Scale lock mode and result in a LockTimeoutException exception.

Implementing exception handling in locking scenarios:

To prevent locks from being held for excessive amounts of time when a LockTimeoutException exception or a LockDeadlockException exception occurs, an application must ensure that it catches unexpected exceptions and calls the rollback method when something unexpected occurs.

Procedure

1. Catch the exception, and display resulting message.

```
try {
    ...
} catch (ObjectGridException oe) {
    System.out.println(oe);
}
```

The following exception displays as a result:

```
com.ibm.websphere.objectgrid.plugins.LockDeadlockException: Message
```

This message represents the string that is passed as a parameter when the exception is created and thrown.

2. Roll back the transaction after an exception:

```
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
boolean activeTran = false;
try
{
    sess.begin();
    activeTran = true;
    Person p = (IPerson)person.get("Lynn");
    // Lynn had a birthday, so we make her 1 year older.
    p.setAge( p.getAge() + 1 );
    person.put( "Lynn", p );
    sess.commit();
    activeTran = false;
}
}
```



```

finally
{
    if ( activeTran ) sess.rollback();
}

```

The `finally` block in the snippet of code ensures that a transaction is rolled back when an unexpected exception occurs. It not only handles a `LockDeadlockException` exception, but any other unexpected exception that might occur. The `finally` block handles the case where an exception occurs during a commit method invocation. This example is not the only way to deal with unexpected exceptions, and there might be cases where an application wants to catch some of the unexpected exceptions that can occur and display one of its application exceptions. You can add catch blocks as appropriate, but the application must ensure that the snippet of code does not exit without completing the transaction.

Configuring a locking strategy:

You can define an optimistic, a pessimistic, or no locking strategy on each `BackingMap` in the WebSphere eXtreme Scale configuration.

About this task

Each `BackingMap` instance can be configured to use one of the following locking strategies:

1. Optimistic locking mode
2. Pessimistic locking mode
3. None

The default lock strategy is `OPTIMISTIC`. Use optimistic locking when data is changed infrequently. Locks are only held for a short duration while data is being read from the cache and copied to the transaction. When the transaction cache is synchronized with the main cache, any cache objects that have been updated are checked against the original version. If the check fails, then the transaction is rolled back and an `OptimisticCollisionException` exception results.

The `PESSIMISTIC` lock strategy acquires locks for cache entries and should be used when data is changed frequently. Any time a cache entry is read, a lock is acquired and conditionally held until the transaction completes. The duration of some locks can be tuned using transaction isolation levels for the session.

If locking is not required because the data is never updated or is only updated during quiet periods, you can disable locking by using the `NONE` lock strategy. This strategy is very fast because a lock manager is not required. The `NONE` lock strategy is ideal for look-up tables or read-only maps.

For more information about locking strategies, see “Locking strategies” on page 221 the information about locking strategies in the *Product Overview*.

Procedure

- **Configure an optimistic locking strategy**

- Programmatically using the `setLockStrategy` method:

```

import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.LockStrategy;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;

```

```

...
ObjectGrid og =
    ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("test");
BackingMap bm = og.defineMap("optimisticMap");
bm.setLockStrategy( LockStrategy.OPTIMISTIC );

```

- Using the lockStrategy attribute in the .:

```

<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="test">
            <backingMap name="optimisticMap"
                lockStrategy="OPTIMISTIC"/>
        </objectGrid>
    </objectGrids>
</objectGridConfig>

```

- **Configure a pessimistic locking strategy**

- Programmatically using the setLockStrategy method:

```

specify pessimistic strategy programmatically
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.LockStrategy;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
...
ObjectGrid og =
    ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("test");
BackingMap bm = og.defineMap("pessimisticMap");
bm.setLockStrategy( LockStrategy.PESSIMISTIC);

```

- Using the lockStrategy attribute in the .

```

specify pessimistic strategy using XML
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="test">
            <backingMap name="pessimisticMap"
                lockStrategy="PESSIMISTIC"/>
        </objectGrid>
    </objectGrids>
</objectGridConfig>

```

- **Configure a no locking strategy**

- Programmatically using the setLockStrategy method:

```

import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.LockStrategy;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
...
ObjectGrid og =
    ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("test");
BackingMap bm = og.defineMap("noLockingMap");
bm.setLockStrategy( LockStrategy.NONE);

```

- Using the lockStrategy attribute in the .:

```

<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
    xmlns="http://ibm.com/ws/objectgrid/config">
    <objectGrids>
        <objectGrid name="test">

```

```

        <backingMap name="noLockingMap"
            lockStrategy="NONE"/>
    </objectGrid>
</objectGrids>
</objectGridConfig>

```

What to do next

To avoid a `java.lang.IllegalStateException` exception, you must call the `setLockStrategy` method before calling the `initialize` or `getSession` methods on the `ObjectGrid` instance.

Configuring the lock timeout value:

The lock timeout value on a `BackingMap` instance is used to ensure that an application does not wait endlessly for a lock mode to be granted because of a deadlock condition that occurs due to an application error.

Before you begin

To configure the lock timeout value, the locking strategy must be set to either `OPTIMISTIC` or `PESSIMISTIC`. See “Configuring a locking strategy” on page 239 for more information.

About this task

When a `LockTimeoutException` exception occurs, the application must determine if the timeout is occurring because the application is running slower than expected, or if the timeout occurred because of a deadlock condition. If an actual deadlock condition occurred, then increasing the lock wait timeout value does not eliminate the exception. Increasing the timeout results in the exception taking longer to occur. However, if increasing the lock wait timeout value does eliminate the exception, then the problem occurred because the application was running slower than expected. The application in this case must determine why performance is slow.

To prevent deadlocks from occurring, the lock manager has a default timeout value of 15 seconds. If the timeout limit is exceeded, a `LockTimeoutException` exception occurs. If your system is heavily loaded, the default timeout value might cause the `LockTimeoutException` exceptions to occur when no deadlock exists. In this situation, you can increase the lock timeout value programmatically or in the `ObjectGrid` descriptor XML file.

Procedure

- Configure a lock timeout value programmatically on a `BackingMap` instance with the `setLockTimeout` method.

The following example illustrates how to set the lock wait timeout value for the `map1` backing map to 60 seconds:

```

import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.LockStrategy;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
...
ObjectGrid og =
    ObjectGridManagerFactory.getObjectGridManager().createObjectGrid("test");
BackingMap bm = og.defineMap("map1");
bm.setLockStrategy( LockStrategy.PESSIMISTIC );
bm.setLockTimeout( 60 );

```

To avoid a `java.lang.IllegalStateException` exception, call both the `setLockStrategy` method and the `setLockTimeout` method before calling either the `initialize` or `getSession` methods on the `ObjectGrid` instance. The `setLockTimeout` method parameter is a Java primitive integer that specifies the number of seconds that eXtreme Scale waits for a lock mode to be granted. If a transaction waits longer than the lock wait timeout value configured for the `BackingMap`, a `com.ibm.websphere.objectgrid.LockTimeoutException` exception results.

- Configure the lock timeout value using the `lockTimeout` attribute in the `ObjectGrid` descriptor XML file `ObjectGrid` descriptor XML file.

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="test">
      <backingMap name="optimisticMap"
        lockStrategy="OPTIMISTIC"
        lockTimeout="60"/>
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

- Override the lock wait timeout for a single `ObjectMap` instance. Use the `ObjectMap.setLockTimeout` method to override the lock timeout value for a specific `ObjectMap` instance. The lock timeout value affects all transactions started after the new timeout value is set. This method can be useful when lock collisions are possible or expected in select transactions.

Map entry locks with query and indexes:

This topic describes how eXtreme Scale Query APIs and the `MapRangeIndex` indexing plug-in interact with locks and some best practices to increase concurrency and decrease deadlocks when using the pessimistic locking strategy for maps.

Overview

The `ObjectGrid` Query API allows `SELECT` queries over `ObjectMap` cache objects and entities. When a query is run, the query engine uses a `MapRangeIndex` when possible to find matching keys that match values in the query's `WHERE` clause or to bridge relationships. When an index isn't available, the query engine will scan each entry in one or more maps to find the appropriate entries. Both the query engine and index plugins will acquire locks to verify consistent data, depending on the locking strategy, transaction isolation level, and transaction state.

Locking with the `HashIndex` plug-in

The eXtreme Scale `HashIndex` plug-in allows finding keys based on a single attribute stored in the cache entry value. The index stores the indexed value in a separate data structure from the cache map. The index validates the keys against map entries before returning to the user to try to achieve an accurate result set. When the pessimistic lock strategy is used and the index is used against a local `ObjectMap` instance (versus a client/server `ObjectMap`), the index will acquire locks for each matching entry. When using optimistic locking or a remote `ObjectMap`, the locks are always immediately released.

The type of lock that is acquired depends upon the `forUpdate` argument passed to the `ObjectMap.getIndex` method. The `forUpdate` argument specifies the type of lock that the index should acquire. If `false`, a shareable (S) lock is acquired and if `true`, an upgradeable (U) lock is acquired.

If the lock type is shareable, the transaction isolation setting for the session is applied and affects the duration of the lock. See the transaction isolation topic for details on how transaction isolation is used to add concurrency to applications.

Shared locks with query

The eXtreme Scale query engine acquires S locks when needed to introspect the cache entries to discover if they satisfy the query's filter criteria. When using repeatable read transaction isolation with pessimistic locking, the S locks are only retained for the elements that are included in the query result and are released for any entries that are not included in the result. If using a lower transaction isolation level or optimistic locking, the S locks are not retained.

Shared locks with client to server query

When using the eXtreme Scale query from a client, the query typically runs on the server unless all of the maps or entities referenced in the query are local to the client (for example: a client-replicated map or a query result entity). All queries that run in a read/write transaction will retain S locks as described in the previous section. If the transaction is not a read/write transaction, then a session is not retained on the server and the S locks are released.

A read/write transaction is only routed to a primary partition and a session is maintained on the server for the client session. A transaction can be promoted to read/write under the following conditions:

1. Any map configured to use pessimistic locking is accessed using the `ObjectMap.get` and `getAll` API methods or the `EntityManager.find` methods.
2. The transaction is flushed, causing updates to be sent to the server.
3. Any map configured to use optimistic locking is accessed using the `ObjectMap.getForUpdate` or `EntityManager.findForUpdate` method.

Upgradeable locks with query

Shareable locks are useful when concurrency and consistency is important. It guarantees that an entry's value does not change for the life of the transaction. No other transaction can change the value while any other S locks are held, and only one other transaction can establish an intent to update the entry. See the Pessimistic Locking Mode topic for details on the S, U and X locking modes.

Upgradeable locks are used to identify the intent to update a cache entry when using the pessimistic lock strategy. It allows synchronization between transactions that want to modify a cache entry. Transactions can still view the entry using an S lock, but other transactions are prevented from acquiring a U lock or an X lock. In many scenarios, acquiring a U lock without first acquiring an S lock is necessary to avoid deadlocks. See the Pessimistic Locking Mode topic for common deadlock examples.

The `ObjectQuery` and `EntityManager Query` interfaces provide the `setForUpdate` method to identify the intended use for the query result. Specifically, the query engine acquires U locks instead of S locks for each map entry involved in the query result:

```
ObjectMap orderMap = session.getMap("Order");
ObjectQuery q = session.createQuery("SELECT o FROM Order o WHERE o.orderDate=?1");
q.setParameter(1, "20080101");
q.setForUpdate(true);
session.begin();
// Run the query. Each order has U lock
Iterator result = q.getResultIterator();
// For each order, update the status.
while(result.hasNext()) {
    Order o = (Order) result.next();
    o.status = "shipped";
    orderMap.update(o.getId(), o);
}
// When committed, the
session.commit();

Query q = em.createQuery("SELECT o FROM Order o WHERE o.orderDate=?1");
q.setParameter(1, "20080101");
q.setForUpdate(true);
emTran.begin();
// Run the query. Each order has U lock
Iterator result = q.getResultIterator();
// For each order, update the status.
while(result.hasNext()) {
    Order o = (Order) result.next();
    o.status = "shipped";
}
tmTran.commit();
```

When the **setForUpdate** attribute is enabled, the transaction is automatically converted to a read/write transaction and the locks are held on the server as expected. If the query cannot use any indexes, then the map must be scanned which will result in temporary U locks for map entries that do not satisfy the query result, and hold U locks for entries that are included in the result.

Transaction isolation

For transactions, you can configure each backing map configuration with one of three lock strategies: pessimistic, optimistic or none. When you are using pessimistic and optimistic locking, eXtreme Scale uses shared (S), upgradeable (U) and exclusive (X) locks to maintain consistency. This locking behavior is most notable when using pessimistic locking, because optimistic locks are not held. You can use one of three transaction isolation levels to tune the locking semantics that eXtreme Scale uses to maintain consistency in each cache map: repeatable read, read committed and read uncommitted.

Transaction isolation overview

Transaction isolation defines how the changes that are made by one operation become visible to other concurrent operations.

WebSphere eXtreme Scale supports three transaction isolation levels with which you can further tune the locking semantics that eXtreme Scale uses to maintain consistency in each cache map: repeatable read, read committed and read uncommitted. The transaction isolation level is set on the `Session` interface using the `setTransactionIsolation` method. The transaction isolation can be changed any time during the life of the session, if a transaction is not currently in progress.

The product enforces the various transaction isolation semantics by adjusting the way in which shared (S) locks are requested and held. Transaction isolation has no effect on maps configured to use the optimistic or none locking strategies or when upgradeable (U) locks are acquired.

Repeatable read with pessimistic locking

The repeatable read transaction isolation level is the default. This isolation level prevents dirty reads and non-repeatable reads, but does not prevent phantom reads. A dirty read is a read operation that occurs on data that has been modified by a transaction but has not been committed. A non-repeatable read might occur when read locks are not acquired when performing a read operation. A phantom read can occur when two identical read operations are performed, but two different sets of results are returned because an update has occurred on the data between the read operations. The product achieve a repeatable read by holding onto any S locks until the transaction that owns the lock completes. Because an X lock is not granted until all S locks are released, all transactions holding the S lock are guaranteed to see the same value when re-read.

```
map = session.getMap("Order");
session.setTransactionIsolation(Session.TRANSACTION_REPEATABLE_READ);
session.begin();

// An S lock is requested and held and the value is copied into
// the transactional cache.
Order order = (Order) map.get("100");
// The entry is evicted from the transactional cache.
map.invalidate("100", false);

// The same value is requested again. It already holds the
// lock, so the same value is retrieved and copied into the
// transactional cache.
Order order2 (Order) = map.get("100");

// All locks are released after the transaction is synchronized
// with cache map.
session.commit();
```

Phantom reads are possible when you are using queries or indexes because locks are not acquired for ranges of data, only for the cache entries that match the index or query criteria. For example:

```
session1.setTransactionIsolation(Session.TRANSACTION_REPEATABLE_READ);
session1.begin();

// A query is run which selects a range of values.
ObjectQuery query = session1.createObjectQuery
    ("SELECT o FROM Order o WHERE o.itemName='Widget'");

// In this case, only one order matches the query filter.
// The order has a key of "100".
// The query engine automatically acquires an S lock for Order "100".
Iterator result = query.getResultIterator();

// A second transaction inserts an order that also matches the query.
Map orderMap = session2.getMap("Order");
orderMap.insert("101", new Order("101", "Widget"));

// When the query runs again in the current transaction, the
// new order is visible and will return both Orders "100" and "101".
result = query.getResultIterator();
```

```
// All locks are released after the transaction is synchronized
// with cache map.
session.commit();
```

Read committed with pessimistic locking

The read committed transaction isolation level can be used with eXtreme Scale, which prevents dirty reads, but does not prevent non-repeatable reads or phantom reads, so eXtreme Scale continues to use S locks to read data from the cache map, but immediately releases the locks.

```
map1 = session1.getMap("Order");
session1.setTransactionIsolation(Session.TRANSACTION_READ_COMMITTED);
session1.begin();
```

```
// An S lock is requested but immediately released and
//the value is copied into the transactional cache.
```

```
Order order = (Order) map1.get("100");
```

```
// The entry is evicted from the transactional cache.
map1.invalidate("100", false);
```

```
// A second transaction updates the same order.
// It acquires a U lock, updates the value, and commits.
// The ObjectGrid successfully acquires the X lock during
// commit since the first transaction is using read
// committed isolation.
```

```
Map orderMap2 = session2.getMap("Order");
session2.begin();
order2 = (Order) orderMap2.getForUpdate("100");
order2.quantity=2;
orderMap2.update("100", order2);
session2.commit();
```

```
// The same value is requested again. This time, they
// want to update the value, but it now reflects
// the new value
Order order1Copy (Order) = map1.getForUpdate("100");
```

Read uncommitted with pessimistic locking

The read uncommitted transaction isolation level can be used with eXtreme Scale, which is a level that allows dirty reads, non-repeatable reads and phantom reads.

Optimistic collision exception

You can receive an `OptimisticCollisionException` directly, or receive it with an `ObjectGridException`.

The following code is an example of how to catch the exception and then display its message:

```
try {
...
} catch (ObjectGridException oe) {
    System.out.println(oe);
}
```

Exception cause

`OptimisticCollisionException` is created in a situation in which two different clients try to update the same map entry at relatively the same time. For example, if one

client attempts to commit a session and update the map entry after another client reads the data before the commit, that data is then incorrect. The exception is created when the other client attempts to commit the incorrect data.

Retrieving the key that triggered the exception

It might be useful, when troubleshooting such an exception, to retrieve the key corresponding to the entry that triggered the exception. The benefit of the `OptimisticCollisionException` is it contains the `getKey` method, which returns the object representing that key. The following example demonstrates how to retrieve and print the key when catching `OptimisticCollisionException`:

```
try {
    ...
} catch (OptimisticCollisionException oce) {
    System.out.println(oce.getKey());
}
```

ObjectGridException causes an OptimisticCollisionException

`OptimisticCollisionException` might be the cause of `ObjectGridException` displaying. If this is the case, you can use the following code to determine the exception type and print out the key. The following code uses the `findRootCause` utility method as described in the section below.

```
try {
    ...
} catch (ObjectGridException oe) {
    Throwable root = findRootCause( oe );
    if (root instanceof OptimisticCollisionException) {
        OptimisticCollisionException oce = (OptimisticCollisionException)root;
        System.out.println(oce.getKey());
    }
}
```

General exception handling technique

Knowing the root cause of a `Throwable` object is helpful in isolating the source of an error. The following example demonstrates how an exception handler uses a utility method to find the root cause of the `Throwable` object.

Example:

```
static public Throwable findRootCause( Throwable t )
{
    // Start with Throwable that occurred as the root.
    Throwable root = t;

    // Follow cause chain until last Throwable in chain is found.
    Throwable cause = root.getCause();
    while ( cause != null )
    {
        root = cause;
        cause = root.getCause();
    }

    // Return last Throwable in the chain as the root cause.
    return root;
}
```

Running parallel business logic on the data grid (DataGrid API)

The DataGrid API provides a simple programming interface to run business logic over all or a subset of the data grid in parallel with where the data is located.

DataGrid APIs and partitioning:

With the DataGrid APIs, a client can send requests to one partition, a subset of partitions, or all the partitions in a data grid. The client can specify a list of keys, and WebSphere eXtreme Scale determines the set of partitions that are hosting the keys. The request is then sent to all the partitions in the set in parallel and the client waits for the results. The client can also send requests without specifying keys, therefore, requests are sent to all partitions.

Agents that are deployed to the data grid do not work in client mode. These agents work directly against the primary shard. Working directly against the primary shard results in maximum performance, allowing tens of thousands or more transactions per second because the agent works with the data at full memory speeds. Working directly with the primary shard also means that an agent can only see data that is within that shard. This provides some interesting opportunities that cannot be done on a client.

A typical eXtreme Scale client must be able to determine the partition from the transaction, because the client needs to route the request. If an agent is directly attached to a shard, then no routing is needed. All requests go against that shard. Because the agent is directly attached to a shard, data in other maps in the shard can be accessed without worrying about common partitioning keys, and so on, because no routing occurs.

DataGrid agents and entity-based Maps:

A map contains key objects and value objects. The key object is a generated tuple as is the value. An agent is normally provided with the application specified key objects.

The key object is a generated tuple as is the value. An agent is normally provided with the application specified key objects. This will be the key objects used by the application or Tuples if it is an entity Map. An application using Entities will not want to deal with Tuples directly and would prefer to work with the Java objects mapped to the Entity.

Therefore, an Agent class can implement the EntityAgentMixin interface. This forces the class to implement one more method, getClassForEntity(). This returns the entity class to use with the agent on the server side. The keys are converted to this Entity before invoking the process and reduce methods.

This is a different semantic to a non EntityAgentMixin agent where those methods are provided with just the keys. An agent implementing EntityAgentMixin receives the Entity object which includes keys and values in one object.

Note: If the entity does not exist on the server, the keys are the raw Tuple format of the key instead of the managed entity.

DataGrid API example:

The DataGrid APIs support two common grid programming patterns: parallel map and parallel reduce.

Parallel Map

The parallel map allows the entries for a set of keys to be processed and returns a result for each entry processed. The application makes a list of keys and receives a Map of key/result pairs after invoking a Map operation. The result is the result of applying a function to the entry of each key. The function is supplied by the application.

MapGridAgent call flow

When the `AgentManager.callMapAgent` method is invoked with a collection of keys, the `MapGridAgent` instance is serialized and sent to each primary partition that the keys resolve to. This means that any instance data stored in the agent can be sent to the server. Each primary partition therefore has one instance of the agent. The process method is invoked for each instance one time for each key that resolves to the partition. The result of each process method is then serialized back to the client and returned to the caller in a Map instance, where the result is represented as the value in the map.

When the `AgentManager.callMapAgent` method is invoked without a collection of keys, the `MapGridAgent` instance is serialized and sent to every primary partition. This means that any instance data stored in the agent can be sent to the server. Each primary partition therefore has one instance (partition) of the agent. The `processAllEntries` method is invoked for each partition. The result of each `processAllEntries` method is then serialized back to the client and returned to the caller in a Map instance. The following example assumes that a `Person` entity exists with the following shape:

```
import com.ibm.websphere.projector.annotations.Entity;
import com.ibm.websphere.projector.annotations.Id;
@Entity
public class Person
{
    @Id String ssn;
    String firstName;
    String surname;
    int age;
}
```

The application supplied function is written as a class that implements the `MapAgentGrid` interface. An example agent that shows a function to return the age of a `Person` multiplied by two.

```
public class DoublePersonAgeAgent implements MapGridAgent, EntityAgentMixin
{
    private static final long serialVersionUID = -2006093916067992974L;

    int lowAge;
    int highAge;

    public Object process(Session s, ObjectMap map, Object key)
    {
        Person p = (Person)key;
        return new Integer(p.age * 2);
    }

    public Map processAllEntries(Session s, ObjectMap map)
    {
        EntityManager em = s.getEntityManager();
        Query q = em.createQuery("select p from Person p where p.age > ?1 and p.age < ?2");
        q.setParameter(1, lowAge);
        q.setParameter(2, highAge);
        Iterator iter = q.getResultIterator();
        Map<Person, Integer> rc = new HashMap<Person, Integer>();
        while(iter.hasNext())
        {
            Person p = (Person)iter.next();
            rc.put(p, (Integer)process(s, map, p));
        }
    }
}
```

```

    return rc;
}
public Class getClassForEntity()
{
    return Person.class;
}
}

```

The previous example shows the Map agent for doubling a Person. The first process method is supplied with the Person to work with and returns double the age of that entry. The second process method is called for each partition and finds all Person objects with an age between lowAge and highAge and returns their ages doubled.

```

Session s = grid.getSession();
ObjectMap map = s.getMap("Person");
AgentManager amgr = map.getAgentManager();

DoublePersonAgeAgent agent = new DoublePersonAgeAgent();

// make a list of keys
ArrayList<Person> keyList = new ArrayList<Person>();
Person p = new Person();
p.ssn = "1";
keyList.add(p);
p = new Person ();
p.ssn = "2";
keyList.add(p);

// get the results for those entries
Map<Tuple, Object> = amgr.callMapAgent(agent, keyList);
// Close the session (optional in Version 7.1.1 and later) for improved performance
s.close();

```

The previous example shows a client obtaining a Session and a reference to the Person Map. The agent operation is performed against a specific Map. The AgentManager interface is retrieved from that Map. An instance of the agent to invoke is created and any necessary state is added to the object by setting attributes, there are none in this case. A list of keys are then constructed. A Map with the values for person 1 doubled, and the same values for person 2 are returned.

The agent is then invoked for that set of keys. The agents process method is invoked on each partition with some of the specified keys in the grid in parallel. A Map is returned providing the merged results for the specified key. In this case, a Map with the values holding the age for person 1 doubled and the same for person 2 is returned.

If the key does not exist, the agent is still invoked. This invocation gives the agent the opportunity to create the map entry. If you are using an EntityAgentMixin, the key to process is not the entity, but the actual Tuple key value for the entity. If the keys are unknown, then you can ask all partitions to find Person objects of a certain shape and return their ages doubled. An example follows:

```

Session s = grid.getSession();
ObjectMap map = s.getMap("Person");
AgentManager amgr = map.getAgentManager();

DoublePersonAgeAgent agent = new DoublePersonAgeAgent();
agent.lowAge = 20;
agent.highAge = 9999;

Map m = amgr.callMapAgent(agent);

```

The previous example shows the `AgentManager` being obtained for the `Person` `Map`, and the agent constructed and initialized with the low and high ages for `Persons` of interest. The agent is then invoked using the `callMapAgent` method. Notice, no keys are supplied. As a result, the `ObjectGrid` invokes the agent on every partition in the grid in parallel and returns the merged results to the client. This set of returns contains all `Person` objects in the grid with an age between low and high and calculates the age of those `Person` objects doubled. This example shows how the grid APIs can be used to run a query to find entities that match a certain query. The agent is serialized and transported by the `ObjectGrid` to the partitions with the needed entries. The results are similarly serialized for transport back to the client. Care needs to be taken with the `Map` APIs. If the `ObjectGrid` was hosting terabytes of objects and running on many servers, then potentially this processing would overwhelm client machines. Use `Map` APIs to process a small subset. If a large subset needs processing, use a `reduce` agent to do the processing out in the data grid rather than on a client.

Parallel Reduction or aggregation agents

This style of programming processes a subset of the entries and calculates a single result for the group of entries. Examples of such a result would be:

- Minimum value
- Maximum value
- Some other business-specific function

A `reduce` agent is coded and invoked in a similar manner to the `Map` agents.

ReduceGridAgent call flow

When the `AgentManager.callReduceAgent` method is invoked with a collection of keys, the `ReduceGridAgent` instance is serialized and sent to each primary partition that the keys resolve to. This means that any instance data stored in the agent can be sent to the server. Each primary partition therefore has one instance of the agent. The `reduce(Session s, ObjectMap map, Collection keys)` method is invoked once per instance (partition) with the subset of keys that resolves to the partition. The result of each `reduce` method is then serialized back to the client. The `reduceResults` method is invoked on the client `ReduceGridAgent` instance with the collection of each result from each remote `reduce` invocation. The result from the `reduceResults` method is returned to the caller of the `callReduceAgent` method.

When the `AgentManager.callReduceAgent` method is invoked without a collection of keys, the `ReduceGridAgent` instance is serialized and sent to each primary partition. This means that any instance data stored in the agent can be sent to the server. Each primary partition therefore has one instance of the agent. The `reduce(Session s, ObjectMap map)` method is invoked once per instance (partition). The result of each `reduce` method is then serialized back to the client. The `reduceResults` method is invoked on the client `ReduceGridAgent` instance with the collection of each result from each remote `reduce` invocation. The result from the `reduceResults` method is returned to the caller of the `callReduceAgent` method. Here is an example of a `reduce` agent that simply adds the ages of the matching entries.

```
package com.ibm.ws.objectgrid.test.agent.jdk5;

import java.util.Collection;
import java.util.Iterator;

import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.datagrid.EntryErrorValue;
import com.ibm.websphere.objectgrid.datagrid.ReduceGridAgent;
```

```

import com.ibm.websphere.objectgrid.query.ObjectQuery;
import com.ibm.websphere.samples.objectgrid.entityxmlgen.PersonFeature1Entity.PersonKey;

public class SumAgeReduceAgent implements ReduceGridAgent {
    private static final long serialVersionUID = 252108071723284899L;

    /**
     * Invoked on the server if a collection of keys is passed to
     * AgentManager.callReduceAgent(). This is invoked on each primary shard
     * where the key applies.
     */
    public Object reduce(Session s, ObjectMap map, Collection keyList) {
        try {
            int sum = 0;
            Iterator<PersonKey> iter = keyList.iterator();
            while (iter.hasNext()) {
                Object nextKey = iter.next();
                PersonKey pk = (PersonKey) nextKey;
                Person p = (Person) map.get(pk);
                sum += p.age;
            }

            return sum;
        } catch (Exception e) {
            throw new RuntimeException(e.getMessage(), e);
        }
    }

    /**
     * Invoked on the server if a collection of keys is NOT passed to
     * AgentManager.callReduceAgent(). This is invoked on every primary shard.
     */
    public Object reduce(Session s, ObjectMap map) {
        ObjectQuery q = s
            .createQuery("select p from Person p where p.age > -1");
        Iterator<Person> iter = q.getResultIterator();
        int sum = 0;
        while (iter.hasNext()) {
            Object nextKey = iter.next();
            Person p = (Person) nextKey;
            sum += p.age;
        }
        return sum;
    }

    /**
     * Invoked on the client to reduce the results from all partitions.
     */
    public Object reduceResults(Collection results) {
        // If we encounter an EntryErrorValue, then throw a RuntimeException
        // to indicate that there was at least one failure and include each
        // EntryErrorValue
        // as part of the thrown exception.
        Iterator<Integer> iter = results.iterator();
        int sum = 0;
        while (iter.hasNext()) {
            Object nextResult = iter.next();
            if (nextResult instanceof EntryErrorValue) {
                EntryErrorValue eev = (EntryErrorValue) nextResult;
                throw new RuntimeException(
                    "Error encountered on one of the partitions: "
                    + nextResult, eev.getException());
            }

            sum += ((Integer) nextResult).intValue();
        }
        return new Integer(sum);
    }
}

```

The previous example shows the agent. The agent has three important parts. The first allows a specific set of entries to be processed without a query. It iterates over the set of entries, adding the ages. The sum is returned from the method. The second uses a query to select the entries to be aggregated. It then sums all the matching Person ages. The third method is used to aggregate the results from each partition to a single result. The ObjectGrid performs the entry aggregation in parallel across the grid. Each partition produces an intermediate result that must be aggregated with other partition intermediate results. This third method performs that task. In the following example the agent is invoked, and the ages of all Persons with ages 10 - 20 exclusively are aggregated:

```

Session s = grid.getSession();
ObjectMap map = s.getMap("Person");
AgentManager amgr = map.getAgentManager();

```

```

SumAgeReduceAgent agent = new SumAgeReduceAgent();

Person p = new Person();
p.ssn = "1";
ArrayList<Person> list = new ArrayList<Person>();
list.add(p);
p = new Person ();
p.ssn = "2";
list.add(p);
Integer v = (Integer)amgr.callReduceAgent(agent, list);
// Close the session (optional in Version 7.1.1 and later) for improved performance
s.close();

```

Agent functions

The agent is free to do ObjectMap or EntityManager operations within the local shard where it is running. The agent receives a Session and can add, update, query, read, or remove data from the partition the Session represents. Some applications query only data from the grid, but you can also write an agent to increment all the Person ages by 1 that match a certain query. There is a transaction on the Session when the agent is called, and is committed when the agent returns unless an exception is thrown

Error handling

If a map agent is invoked with an unknown key then the value that is returned is an error object that implements the EntryErrorValue interface.

Transactions

A map agent runs in a separate transaction from the client. Agent invocations may be grouped into a single transaction. If an agent fails and throws an exception, the transaction is rolled back. Any agents that ran successfully in a transaction rolls back with the failed agent. The AgentManager reruns the rolled-back agents that ran successfully in a new transaction.

Configuring clients programmatically

You can configure a WebSphere eXtreme Scale client based on your requirements such as the need to override settings.

Override plug-ins

You can override the following plug-ins on a client:

- **ObjectGrid plug-ins**
 - TransactionCallback plug-in
 - ObjectGridEventListener plug-in
- **BackingMap plug-ins**
 - Evictor plug-in
 - MapEventListener plug-in
 - numberOfBuckets attribute
 - ttlEvictorType attribute
 - timeToLive attribute

Configure the client programmatically

You can also override client-side ObjectGrid settings programmatically. Create an ObjectGridConfiguration object that is similar in structure to the server-side ObjectGrid instance. The following code creates a client-side ObjectGrid instance that is functionally equivalent to the client override in the previous section which uses an XML file.

```
client-side override programmatically
ObjectGridConfiguration companyGridConfig = ObjectGridConfigFactory
    .createObjectGridConfiguration("CompanyGrid");
Plugin txCallbackPlugin = ObjectGridConfigFactory.createPlugin(
    PluginType.TRANSACTION_CALLBACK, "com.company.MyClientTxCallback");
companyGridConfig.addPlugin(txCallbackPlugin);

Plugin ogEventListenerPlugin = ObjectGridConfigFactory.createPlugin(
    PluginType.OBJECTGRID_EVENT_LISTENER, "");
companyGridConfig.addPlugin(ogEventListenerPlugin);

BackingMapConfiguration customerMapConfig = ObjectGridConfigFactory
    .createBackingMapConfiguration("Customer");
customerMapConfig.setNumberOfBuckets(1429);
Plugin evictorPlugin = ObjectGridConfigFactory.createPlugin(PluginType.EVICTOR,
    "com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor");
customerMapConfig.addPlugin(evictorPlugin);

companyGridConfig.addBackingMapConfiguration(customerMapConfig);

BackingMapConfiguration orderLineMapConfig = ObjectGridConfigFactory
    .createBackingMapConfiguration("OrderLine");
orderLineMapConfig.setNumberOfBuckets(701);
orderLineMapConfig.setTimeToLive(800);
orderLineMapConfig.setTtlEvictorType(TTLType.LAST_ACCESS_TIME);

companyGridConfig.addBackingMapConfiguration(orderLineMapConfig);

List ogConfigs = new ArrayList();
ogConfigs.add(companyGridConfig);

Map overrideMap = new HashMap();
overrideMap.put(CatalogServerProperties.DEFAULT_DOMAIN, ogConfigs);

ogManager.setOverrideObjectGridConfigurations(overrideMap);
ClientClusterContext client = ogManager.connect(catalogServerEndpoints, null, null);
ObjectGrid companyGrid = ogManager.getObjectGrid(client, objectGridName);
```

The ogManager instance of the ObjectGridManager interface checks for overrides only in the ObjectGridConfiguration and BackingMapConfiguration objects that you include in the overrideMap Map. For instance, the previous code overrides the number of buckets on the OrderLine Map. However, the Order map remains unchanged on the client side because no configuration for that map is included.

Disable the client near cache

The near cache is enabled by default when locking is configured as optimistic or none. Clients do not maintain a near cache when the locking setting is configured as pessimistic. To disable the near cache, you must set the numberOfBuckets attribute to 0 in the client override ObjectGrid descriptor file.

Enabling client-side map replication

You can also enable replication of maps on the client side to make data available faster.

With eXtreme Scale, you can replicate a server map to one or more clients by using asynchronous replication. A client can request a local read-only copy of a server side map by using the ClientReplicableMap.enableClientReplication method.

```
void enableClientReplication(Mode mode, int[] partitions,
    ReplicationMapListener listener) throws ObjectGridException;
```


The first parameter is the replication mode. This mode can be a continuous replication or a snapshot replication. The second parameter is an array of partition IDs that represent the partitions from which to replicate the data. If the value is null or an empty array, the data is replicated from all the partitions. The last parameter is a listener to receive client replication events. See `ClientReplicableMap` and `ReplicationMapListener` in the API documentation for details.

After the replication is enabled, then the server starts to replicate the map to the client. The client is eventually only a few transactions behind the server at any point in time.

Accessing data with the REST data service

Develop applications that perform operations using REST data service protocols.

Operations with the REST data service

After you start the eXtreme Scale REST data service, you can use any HTTP client to interact with it. A Web browser, PHP client, Java client or WCF Data Services client can be used to issue any of the supported request operations.

The REST service implements a subset of the Microsoft Atom Publishing Protocol: Data Services URI and Payload Extensions specification, Version 1.0 which is part of OData protocol. This topic describes which of the features of the specification are supported and how they are mapped to eXtreme Scale.

Service root URI

Microsoft WCF Data Services typically defines a service per data source or entity model. The eXtreme Scale REST data service defines a service per defined `ObjectGrid`. Each `ObjectGrid` that is defined in the eXtreme Scale `ObjectGrid` client override XML file is automatically exposed as a separate REST service root.

The URI for the service root is:

```
http://host:port/contextroot/restservice/gridname
```

Where:

- *contextroot* is defined when you deploy the REST data service application, and depends on the application server
- *gridname* is the name of the `ObjectGrid`

Request types

The following list describes the Microsoft WCF Data Services request types which the eXtreme Scale REST data service supports. For details about each request type that WCF Data Services supports, see: MSDN: Request Types.

Insert request types

Clients can insert resources using the POST HTTP verb with the following limitations:

- `InsertEntity` Request: Supported.
- `InsertLink` request: Supported.
- `InsertMediaResource` request: Not supported due to media resource support restriction.

For additional information, see: MSDN: Insert Request Types.

Update request types

Clients can update resources using the PUT and MERGE HTTP verbs with the following limitations:

- UpdateEntity Request: Supported.
- UpdateComplexType Request: Not Supported due to complex type restriction.
- UpdatePrimitiveProperty Request: Supported.
- UpdateValue Request: Supported.
- UpdateLink Request: Supported.
- UpdateMediaResource Request: Not supported due to media resource support restriction.

For additional information, see: MSDN: Insert Request types.

Delete request types

Clients can delete resources using the DELETE HTTP verb with the following limitations:

- DeleteEntity Request: Supported.
- DeleteLink Request: Supported.
- DeleteValue request: Supported.

For additional information, see: MSDN: Delete Request Types.

Retrieve request types

Clients can retrieve resources using the GET HTTP verb with the following limitations:

- RetrieveEntitySet Request: Supported.
- RetrieveEntity Request: Supported.
- RetrieveComplexType Request: Not supported due to complex type restriction.
- RetrievePrimitiveProperty Request: Supported.
- RetrieveValue Request: Supported.
- RetrieveServiceMetadata Request: Supported.
- RetrieveServiceDocument Request: Supported.
- RetrieveLink Request: Supported.
- Retrieve Request Containing a Customizable Feed Mapping: Not supported
- RetrieveMediaResource: Not supported due to media resource restriction.

For additional information, see: MSDN: Retrieve Request Types.

System query options

Queries are supported which allow clients to identify a collection of entities or a single entity. System query options are specified in a data service URI and are supported with the following limitations:

- \$expand: Supported
- \$filter: Supported.
- \$orderby: Supported.

- \$format: Not supported. The acceptable format is identified in the HTTP Accept request header.
- \$skip: Supported
- \$top: Supported

For additional information, see: MSDN: System Query Options.

Partition routing

Partition routing is based on the root entity. A request URI infers a root entity if its resource path starts with a root entity or with an entity that has a direct or indirect association to the entity. In a partitioned environment, any request that cannot infer a root entity will be rejected. Any request that infers a root entity will be routed to the correct partition.

For additional information on defining a schema with associations and root entities, see Scalable data model in eXtreme Scale and Partitioning.

Invoke request

Invoke requests are not supported. For additional information, see MSDN: Invoke Request.

Batch request

Clients can batch multiple Change Sets or Query Operations within a single request. This can reduce the number of round trips to the server and allows multiple requests to participate in a single transaction. For additional information, see MSDN: Batch Request.

Tunneled requests

Tunneled requests are not supported. For additional information, see MSDN: Tunneled Requests.

Optimistic concurrency in the REST data service

The eXtreme Scale REST data service uses an optimistic locking model by using native HTTP headers: If-Match, If-None-Match, and ETag. These headers are sent in request and response messages to relay an entity's version information from the server to client and client to server.

For more details on optimistic concurrency, refer to MSDN Library: Optimistic Concurrency (ADO.NET).

The eXtreme Scale REST data service enables optimistic concurrency for an entity if a version attribute is defined in the entity schema for that entity. A version property can be defined in the entity schema by a @Version annotation for Java classes or a <version/> attribute for entities defined using an entity descriptor XML file. The eXtreme Scale REST data service automatically propagates the value of the version property to the client in the ETag header for single entity responses using an m:etag attribute in the payload for multiple entity XML responses, and an etag attribute in the payload for multiple entity JSON responses.

For more details on defining an eXtreme Scale entity schema, see "Defining an entity schema" on page 166.

Request protocols for the REST data service

In general, the protocols for interacting with the REST service are the same as those described in the WCF Data Services AtomPub protocol. However, eXtreme Scale does provide additional details, from eXtreme Scale Entity Model perspective. Users are expected to be familiar with the WCF Data Services protocols before reading this section. Alternatively, users can read this section with the WCF Data Services protocol section.

Examples are provided to illustrate the request and response. These examples apply to both the eXtreme Scale REST data service and WCF Data Services. Because Web browsers can only retrieve data, the CUD (create, update and delete) operations must be performed by another client such as Java, JavaScript, RUBY or PHP.

Retrieve requests with the REST data service

A RetrieveEntity Request is used by a client to retrieve an eXtreme Scale entity. The response payload contains the entity data in AtomPub or JSON format. Also, the system operator \$expand can be used to expand the relations. The relations are represented in line within the data service response as an Atom Feed Document, which is a to-many relation, or an Atom Entry Document which is a to-one relation.

Tip: For more details on the RetrieveEntity protocol defined in WCF Data Services, refer to MSDN: RetrieveEntity Request.

Retrieving an entity

The following RetrieveEntity example retrieves a Customer entity with key.

AtomPub

- Method
GET
- Request URI:
`http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/`
`Customer('ACME')`
- Request Header:
Accept: application/atom+xml
- Request Payload:
None
- Response Header:
Content-Type: application/atom+xml
- Response Payload:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<entry xml:base = "http://localhost:8080/wxsrestservice/
restservice" xmlns:d= "http://schemas.microsoft.com/ado/2007/
08/dataservices" xmlns:m = "http://schemas.microsoft.com/ado/2007/
08/dataservices/metadata" xmlns = "http://www.w3.org/2005/Atom">

<category term = "NorthwindGridModel.Customer" scheme = "http://
schemas.microsoft.com/ado/2007/08/dataservices/scheme"/>
<id>http://localhost:8080/wxsrestservice/restservice/
NorthwindGrid/Customer('ACME')</id>
  <title type = "text"/>
  <updated>2009-12-16T19:52:10.593Z</updated>
  <author>
```

```

        <name/>
    </author>
    <link rel = "edit" title = "Customer" href = "Customer(
    'ACME')"/>
    <link rel = "http://schemas.microsoft.com/ado/2007/08/
    dataservices/related/
orders" type = "application/atom+xml;type=feed" title =
"orders" href = "Customer('ACME')/orders"/>
    <content type = "application/xml">
        <m:properties>
            <d:customerId>ACME</d:customerId>
            <d:city m:null = "true"/>
            <d:companyName>RoaderRunner</d:companyName>
            <d:contactName>ACME</d:contactName>
            <d:country m:null = "true"/>
            <d:version m:type = "Edm.Int32">3</d:version>
        </m:properties>
    </content>
</entry>

```

- Response Code:
200 OK

JSON

- Method
GET
- Request URI:
http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
Customer('ACME')
- Request Header:
Accept: application/json
- Request Payload:
None
- Response Header:
Content-Type: application/json
- Response Payload:

```

{"d":{"__metadata":{"uri":"http://localhost:8080/wxsrestservice/
restservice/NorthwindGrid/Customer('ACME')",
"type":"NorthwindGridModel.Customer"},
"customerId":"ACME",
"city":null,
"companyName":"RoaderRunner",
"contactName":"ACME",
"country":null,
"version":3,
"orders":{"__deferred":{"uri":"http://localhost:8080/
wxsrestservice/restservice/
NorthwindGrid/Customer('ACME')/orders"}}}}

```
- Response Code:
200 OK

Queries

A query can also be used with a RetrieveEntitySet or RetrieveEntity request. A query is specified by the system \$filter operator.

For details on the \$filter operator, refer to MSDN: Filter System Query Option (\$filter)

The OData protocol supports several common expressions. The eXtreme Scale REST data service supports a subset of the expressions defined in the specification:

- Boolean expressions:
 - eq, ne, lt, le, gt, ge
 - negate
 - not
 - parenthesis
 - and, or
- Arithmetic expressions:
 - add
 - sub
 - mul
 - div
- Primitive literals
 - String
 - date-time
 - decimal
 - single
 - double
 - int16
 - int32
 - int64
 - binary
 - null
 - byte

The following expressions are *not* available:

- Boolean expressions:
 - isof
 - cast
- Method call expressions
- Arithmetic expressions:
 - mod
- Primitive literals:
 - Guid
- Member expressions

For a complete list and description of the expressions that are available in Microsoft WCF Data Services, see section 2.2.3.6.1.1 : Common Expression Syntax.

The following example demonstrates a RetrieveEntity request with a query. In this example, all customers whose contact name is “RoadRunner” are retrieved. The only customer which matches this filter is Customer('ACME') as shown in the response payload.

Restriction: This query will only work for non-partitioned entities. If Customer is partitioned, then the key belonging to the customer is required.

AtomPub

- Method: GET
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/ Customer?$filter=contactName eq 'RoadRunner'`
- Request Header: Accept: application/atom+xml
- Input Payload: None
- Response Header: Content-Type: application/atom+xml
- Response Payload:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<feed
  xmlns:base="http://localhost:8080/wxsrestservice/restservice"
  xmlns:d="http://schemas.microsoft.com/ado/2007/08/
    dataservices"
  xmlns:m="http://schemas.microsoft.com/ado/2007/08/
    dataservices/metadata"
  xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Customer</title>
  <id> http://localhost:8080/wxsrestservice/restservice/
    NorthwindGrid/Customer </id>
  <updated>2009-09-16T04:59:28.656Z</updated>
  <link rel="self" title="Customer" href="Customer" />
  <entry>
    <category term="NorthwindGridModel.Customer"
      scheme="http://schemas.microsoft.com/ado/2007/08/
        dataservices/scheme" />
    <id>
      http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
      Customer('ACME')</id>
    <title type="text" />
    <updated>2009-09-16T04:59:28.656Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Customer" href="Customer('ACME')" />
    <link
      rel="http://schemas.microsoft.com/ado/2007/08/dataservices/
        related/orders"
      type="application/atom+xml;type=feed" title="orders"
      href="Customer('ACME')/orders" />
    <content type="application/xml">
      <m:properties>
        <d:customerId>ACME</d:customerId>
        <d:city m:null = "true"/>
        <d:companyName>RoadRunner</d:companyName>
        <d:contactName>ACME</d:contactName>
        <d:country m:null = "true"/>
        <d:version m:type = "Edm.Int32">3</d:version>
      </m:properties>
    </content>
  </entry>
</feed>
```

- Response Code: 200 OK

JSON

- Method: GET
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/ Customer?$filter=contactName eq 'RoadRunner'`
- Request Header: Accept: application/json
- Request Payload: None

- Response Header: Content-Type: application/json
- Response Payload:


```
{
  "d": [
    {
      "__metadata": {
        "uri": "http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('ACME')",
        "type": "NorthwindGridModel.Customer",
        "customerId": "ACME",
        "city": null,
        "companyName": "RoaderRunner",
        "contactName": "ACME",
        "country": null,
        "version": 3,
        "orders": {
          "__deferred": {
            "uri": "http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('ACME')/orders"
          }
        }
      }
    }
  ]
}
```
- Response Code: 200 OK

System operator \$expand

The system operator \$expand can be used to expand associations. The associations are represented in line in the data service response. Multi-valued (to-many) associations are represented as an Atom Feed Document or JSON array. Single-valued (to-one) associations, are represented as n Atom Entry Document or JSON object.

For more details on the \$expand system operator, refer to Expand System Query Option (\$expand).

Here is an example of using the \$expand system operator. In this example, we retrieve the entity Customer('IBM') which has an Orders 5000, 5001 and others associated with it. The \$expand clause is set to "orders", so the order collection is expanded as inline in the response payload. Only orders 5000 and 5001 are displayed here.

AtomPub

- Method: GET
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')?$expand=orders`
- Request Header: Accept: application/atom+xml
- Request Payload: None
- Response Header: Content-Type: application/atom+xml
- Response Payload:


```
<?xml version="1.0" encoding="utf-8"?>
<entry xml:base = "http://localhost:8080/wxsrestservice/restservice"
  xmlns:d = "http://schemas.microsoft.com/ado/2007/08/dataservices"
  xmlns:m = "http://schemas.microsoft.com/ado/2007/08/dataservices/
  metadata" xmlns = "http://www.w3.org/2005/Atom">
<category term = "NorthwindGridModel.Customer" scheme = "http://schemas.
microsoft.com/ado/2007/08/dataservices/scheme"/>
  <id>http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
  Customer('IBM')</id>
  <title type = "text"/>
  <updated>2009-12-16T22:50:18.156Z</updated>
  <author>
    <name/>
  </author><link rel = "edit" title = "Customer" href =
  "Customer('IBM')"/>
  <link rel = "http://schemas.microsoft.com/ado/2007/08/dataservices/
```



```

related/orders" type = "application/atom+xml;type=feed" title =
"orders" href = "Customer('IBM')/orders">
  <m:inline>
    <feed>
      <title type = "text">orders</title>
      <id>http://localhost:8080/wxsrestservice/restservice/
NorthwindGrid/Customer('IBM')/orders</id>
      <updated>2009-12-16T22:50:18.156Z</updated>
      <link rel = "self" title = "orders" href = "Customer
('IBM')/orders"/>
      <entry>
        <category term = "NorthwindGridModel.Order" scheme =
"http://schemas.microsoft.com/ado/2007/08/
dataservices/scheme"/>
        <id>http://localhost:8080/wxsrestservice/restservice/
NorthwindGrid/Order(orderId=5000,customer_customerId=
'IBM')</id>
        <title type = "text"/>
        <updated>2009-12-16T22:50:18.156Z</updated>
        <author>
          <name/>
        </author>
        <link rel = "edit" title = "Order" href =
"Order(orderId=5000,customer_customerId='IBM')"/>
        <link rel = "http://schemas.microsoft.com/ado/2007/08/
dataservices/related/customer" type = "application/
atom+xml;type=entry" title = "customer" href =
"Order(orderId=5000,customer_customerId='IBM')/customer"/>
        <link rel = "http://schemas.microsoft.com/ado/2007/08/
dataservices/related/orderDetails" type = "application/
atom+xml;type=feed" title = "orderDetails" href =
"Order(orderId=5000,customer_customerId='IBM')/orderDetails"/>
        <content type = "application/xml">
          <m:properties>
            <d:orderId m:type = "Edm.Int32">5000</d:orderId>
            <d:customer_customerId>IBM</d:customer_customerId>
            <d:orderDate m:type = "Edm.DateTime">
2009-12-16T19:46:29.562</d:orderDate>
            <d:shipCity>Rochester</d:shipCity>
            <d:shipCountry m:null = "true"/>
            <d:version m:type = "Edm.Int32">0</d:version>
          </m:properties>
        </content>
      </entry>
      <entry>
        <category term = "NorthwindGridModel.Order" scheme =
"http://schemas.microsoft.com/ado/2007/08/
dataservices/scheme"/>
        <id>http://localhost:8080/wxsrestservice/restservice/
NorthwindGrid/Order(orderId=5001,customer_customerId=
'IBM')</id>
        <title type = "text"/>
        <updated>2009-12-16T22:50:18.156Z</updated>
        <author>
          <name/></author>
        <link rel = "edit" title = "Order" href = "Order(
orderId=5001,customer_customerId='IBM')"/>
        <link rel = "http://schemas.microsoft.com/ado/2007/
08/dataservices/related/customer" type =
"application/atom+xml;type=entry" title =
"customer" href = "Order(orderId=5001,customer_customerId=
'IBM')/customer"/>
        <link rel = "http://schemas.microsoft.com/ado/2007/08/
dataservices/related/orderDetails" type =
"application/atom+xml;type=feed" title =
"orderDetails" href = "Order(orderId=5001,
customer_customerId='IBM')/orderDetails"/>

```

```

        <content type = "application/xml">
            <m:properties>
                <d:orderId m:type = "Edm.Int32">5001</d:orderId>
                <d:customer_customerId>IBM</d:customer_customerId>
                <d:orderDate m:type = "Edm.DateTime">2009-12-16T19:
50:11.125</d:orderDate>
                <d:shipCity>Rochester</d:shipCity>
                <d:shipCountry m:null = "true"/>
                <d:version m:type = "Edm.Int32">0</d:version>
            </m:properties>
        </content>
    </entry>
</feed>
</m:inline>
</link>
<content type = "application/xml">
    <m:properties>
        <d:customerId>IBM</d:customerId>
        <d:city m:null = "true"/>
        <d:companyName>IBM Corporation</d:companyName>
        <d:contactName>John Doe</d:contactName>
        <d:country m:null = "true"/>
        <d:version m:type = "Edm.Int32">4</d:version>
    </m:properties>
</content>
</entry>

```

- Response Code: 200 OK

JSON

- Method: GET
- Request URI: [http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer\('IBM'\)?\\$expand=orders](http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')?$expand=orders)
- Request Header: Accept: application/json
- Request Payload: None
- Response Header: Content-Type: application/json
- Response Payload:

```

{"d":{"__metadata":{"uri":"http://localhost:8080/wxsrestservice/
restservice/NorthwindGrid/Customer('IBM')",
"type":"NorthwindGridModel.Customer"},
"customerId":"IBM",
"city":null,
"companyName":"IBM Corporation",
"contactName":"John Doe",
"country":null,
"version":4,
"orders":[{"__metadata":{"uri":"http://localhost:8080/
wxsrestservice/restservice/NorthwindGrid/Order(
orderId=5000,customer_customerId='IBM')",
"type":"NorthwindGridModel.Order"},
"orderId":5000,
"customer_customerId":"IBM",
"orderDate":"\\/Date(1260992789562)\\/\"",
"shipCity":"Rochester",
"shipCountry":null,
"version":0,
"customer":{"__deferred":{"uri":"http://localhost:8080/
wxsrestservice/restservice/NorthwindGrid/Order(
orderId=5000,customer_customerId='IBM')/customer"}},
"orderDetails":{"__deferred":{"uri":"http://localhost:
8080/wxsrestservice/restservice/NorthwindGrid/
Order(orderId=5000,customer_customerId='IBM')/
orderDetails"}}},
{"__metadata":{"uri":"http://localhost:8080/wxsrestservice/

```

```

    restservice/NorthwindGrid/Order(orderId=5001,
    customer_customerId='IBM')", "type":
    "NorthwindGridModel.Order"},
  "orderId":5001,
  "customer_customerId":"IBM",
  "orderDate":"\\/Date(126099301125)\\/",
  "shipCity":"Rochester",
  "shipCountry":null,
  "version":0,
  "customer":{"__deferred":{"uri":"http://localhost:
    8080/wxsrestservice/restservice/
    NorthwindGrid/Order(orderId=5001,customer_customerId
    ='IBM')/customer"}},
  "orderDetails":{"__deferred":{"uri":"http://localhost:8080/
    wxsrestservice/restservice/NorthwindGrid/Order(
    orderId=5001, customer_customerId='IBM')/
    orderDetails"}}}]}}

```

- Response Code: 200 OK

Retrieving non-entities with REST data services

The REST data service allows you to retrieve more than only entities, such as entity collections and properties.

Retrieve an entity collection

A RetrieveEntitySet Request can be used by a client to retrieve a set of eXtreme Scale entities. The entities are represented as an Atom Feed Document or JSON array in the response payload. For more details on the RetrieveEntitySet protocol defined in WCF Data Services, refer to MSDN: RetrieveEntitySet Request.

The following RetrieveEntitySet request example retrieves all the Order entities associated with the Customer('IBM') entity. Only orders 5000 and 5001 are displayed here.

AtomPub

- Method: GET
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')/orders`
- Request Header: Accept: application/atom+xml
- Request Payload: None
- Response Header: Content-Type: application/atom+xml
- Response Payload:

```

<?xml version="1.0" encoding="utf-8"?>
<feed xml:base = "http://localhost:8080/wxsrestservice/restservice"
  xmlns:d = "http://schemas.microsoft.com/ado/2007/08/dataservices"
  xmlns:m = "http://schemas.microsoft.com/ado/2007/08/dataservices/
  metadata" xmlns = "http://www.w3.org/2005/Atom">
  <title type = "text">Order</title>
  <id>http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
  Order</id>
  <updated>2009-12-16T22:53:09.062Z</updated>
  <link rel = "self" title = "Order" href = "Order"/>
  <entry>
    <category term = "NorthwindGridModel.Order" scheme = "http://
    schemas.microsoft.com/
    ado/2007/08/dataservices/scheme"/>
    <id>http://localhost:8080/wxsrestservice/restservice/
    NorthwindGrid/Order(orderId=5000,customer_customerId=
    'IBM')</id>
    <title type = "text"/>

```

```

    <updated>2009-12-16T22:53:09.062Z</updated>
    <author>
      <name/>
    </author>
    <link rel = "edit" title = "Order" href = "Order(orderId=5000,
customer_customerId='IBM')"/>
    <link rel = "http://schemas.microsoft.com/ado/2007/08/
dataservices/related/customer"
type = "application/atom+xml;type=entry"
title = "customer" href = "Order(orderId=5000,
customer_customerId='IBM')/customer"/>
    <link rel = "http://schemas.microsoft.com/ado/2007/08/
dataservices/related/orderDetails"
type = "application/atom+xml;type=feed"
title = "orderDetails" href = "Order(orderId=5000,
customer_customerId='IBM')/
orderDetails"/>
    <content type = "application/xml">
      <m:properties>
        <d:orderId m:type = "Edm.Int32">5000</d:orderId>
        <d:customer_customerId>IBM</d:customer_customerId>
        <d:orderDate m:type = "Edm.DateTime">2009-12-16T19:
46:29.562</d:orderDate>
        <d:shipCity>Rochester</d:shipCity>
        <d:shipCountry m:null = "true"/>
        <d:version m:type = "Edm.Int32">0</d:version>
      </m:properties>
    </content>
  </entry>
  <entry>
    <category term = "NorthwindGridModel.Order" scheme = "http://
schemas.microsoft.com/ado/2007/08/dataservices/scheme"/>
    <id>http://localhost:8080/wxsrestservice/restservice/
NorthwindGrid/Order(orderId=5001, customer_customerId='IBM')
</id>
    <title type = "text"/>
    <updated>2009-12-16T22:53:09.062Z</updated>
    <author>
      <name/>
    </author>
    <link rel = "edit" title = "Order" href = "Order(orderId=5001,
customer_customerId='IBM')"/>
    <link rel = "http://schemas.microsoft.com/ado/2007/08/
dataservices/related/customer"
type = "application/atom+xml;type=entry"
title = "customer" href = "Order(orderId=5001,
customer_customerId='IBM')/customer"/>
    <link rel = "http://schemas.microsoft.com/ado/2007/08/
dataservices/related/orderDetails"
type = "application/atom+xml;type=feed"
title = "orderDetails" href = "Order(orderId=5001,
customer_customerId='IBM')/orderDetails"/>
    <content type = "application/xml">
      <m:properties>
        <d:orderId m:type = "Edm.Int32">5001</d:orderId>
        <d:customer_customerId>IBM</d:customer_customerId>
        <d:orderDate m:type = "Edm.DateTime">2009-12-16T19:50:
11.125</d:orderDate>
        <d:shipCity>Rochester</d:shipCity>
        <d:shipCountry m:null = "true"/>
        <d:version m:type = "Edm.Int32">0</d:version>
      </m:properties>
    </content>
  </entry>
</feed>

```

- Response Code: 200 OK

JSON

- Method: GET
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')/orders`
- Request Header: `Accept: application/json`
- Request Payload: None
- Response Header: `Content-Type: application/json`
- Response Payload:

```
{ "d": [ { "__metadata": { "uri": "http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=5000,customer_customerId='IBM')", "type": "NorthwindGridModel.Order" }, "orderId": 5000, "customer_customerId": "IBM", "orderDate": "\\Date(1260992789562)\\/", "shipCity": "Rochester", "shipCountry": null, "version": 0, "customer": { "__deferred": { "uri": "http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=5000,customer_customerId='IBM')/customer" } }, "orderDetails": { "__deferred": { "uri": "http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=5000,customer_customerId='IBM')/orderDetails" } } }, { "__metadata": { "uri": "http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=5001,customer_customerId='IBM')", "type": "NorthwindGridModel.Order" }, "orderId": 5001, "customer_customerId": "IBM", "orderDate": "\\Date(1260993011125)\\/", "shipCity": "Rochester", "shipCountry": null, "version": 0, "customer": { "__deferred": { "uri": "http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=5001,customer_customerId='IBM')/customer" } }, "orderDetails": { "__deferred": { "uri": "http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=5001,customer_customerId='IBM')/orderDetails" } } } ] }
```
- Response Code: 200 OK

Retrieve a property

A `RetrievePrimitiveProperty` request can be used to get the value of a property of an eXtreme Scale entity instance. The property value is represented as XML format for AtomPub requests and a JSON object for JSON requests in the response payload. For more details on `RetrievePrimitiveProperty` request, refer to MSDN: `RetrievePrimitiveProperty Request`.

The following `RetrievePrimitiveProperty` request example retrieves the `contactName` property of the `Customer('IBM')` entity.

AtomPub

- Method: GET
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')/contactName`
- Request Header: `Accept: application/xml`

- Request Payload: None
- Response Header: Content-Type: application/atom+xml
- Response Payload:


```
<contactName xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices">
  John Doe
</contactName>
```
- Response Code: 200 OK

JSON

- Method: GET
- Request URI: http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/ Customer('IBM')/contactName
- Request Header: Accept: application/json
- Request Payload: None
- Response Header: Content-Type: application/json
- Response Payload: {"d":{"contactName":"John Doe"}}
- Response Code: 200 OK

Retrieve a property value

A RetrieveValue request can be used to get the raw value of a property on an eXtreme Scale entity instance. The property value is represented as a raw value in the response payload. If the entity type is one of the following, then the media type of the response is "text/plain." Otherwise the response' media type is "application/octet-stream." These types are:

- Java primitive types and their respective wrappers
- java.lang.String
- byte[]
- Byte[]
- char[]
- Character[]
- enums
- java.math.BigInteger
- java.math.BigDecimal
- java.util.Date
- java.util.Calendar
- java.sql.Date
- java.sql.Time
- java.sql.Timestamp

For more details on the RetrieveValue request, refer to MSDN: RetrieveValue Request.

The following RetrieveValue request example retrieves the raw value of the contactName property of the Customer('IBM') entity.

- Request Method: GET
- Request URI: http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/ Customer('IBM')/contactName/\$value
- Request Header: Accept: text/plain

- Request Payload: None
- Response Header: Content-Type: text/plain
- Response Payload: John Doe
- Response Code: 200 OK

Retrieve a link

A RetrieveLink Request can be used to get the link(s) representing a to-one association or to-many association. For the to-one association, the link is from one eXtreme Scale Entity instance to another, and the link is represented in the response payload. For the to-many association, the links are from one eXtreme Scale Entity instance to all others in a specified eXtreme Scale entity collection, and the response is represented as a set of links in the response payload. For more details on RetrieveLink request, refer to MSDN: RetrieveLink Request.

Here is a RetrieveLink request example. In this example, we retrieve the association between entity Order(orderId=5000,customer_customerId='IBM') and its customer. The response shows the Customer entity URI.

AtomPub

- Method: GET
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=5000,customer_customerId='IBM')/$links/customer`
- Request Header: Accept: application/xml
- Request Payload: None
- Response Header: Content-Type: application/xml
- Response Payload:


```
<?xml version="1.0" encoding="utf-8"?>
<uri>http://localhost:8080/wxsrestservice/restservice/
  NorthwindGrid/Customer('IBM')</uri>
```
- Response Code: 200 OK

JSON

- Method: GET
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=5000,customer_customerId='IBM')/$links/customer`
- Request Header: Accept: application/json
- Request Payload: None
- Response Header: Content-Type: application/json
- Response Payload: `{"d":{"uri":"http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')"}}`

Retrieve service metadata

A RetrieveServiceMetadata Request can be used to get the conceptual schema definition language (CSDL) document, which describes the data model associated with the eXtreme Scale REST data service. For more details on RetrieveServiceMetadata request, refer to MSDN: RetrieveServiceMetadata Request.

Retrieve service document

A RetrieveServiceDocument Request can be used to retrieve the Service Document describing the collection of resources exposed by the eXtreme Scale REST data service. For more details on RetrieveServiceDocument request, refer to MSDN: RetrieveServiceDocument Request.

Insert requests with REST data services

An InsertEntity Request can be used to insert a new eXtreme Scale entity instance, potentially with new related entities, into the eXtreme Scale REST data service.

Insert entity request

An InsertEntity Request can be used to insert a new eXtreme Scale entity instance, potentially with new related entities, into the eXtreme Scale REST data service. When inserting an entity, the client may specify if the resource or entity should be automatically linked to other existing entities in the data service.

The client must include the required binding information in the representation of the associated relation in the request payload.

In addition to supporting the insertion of a new EntityType instance (E1), the InsertEntity request also allows inserting new entities related to E1 (described by an entity relation) in a single Request. For example, when inserting a Customer('IBM'), we can insert all the orders with Customer('IBM'). This form of an InsertEntity Request is also known as a *deep insert*. With a deep insert, the related entities must be represented using the inline representation of the relation associated with E1 that identifies the link to the to-be-inserted related entities.

The properties of the entity to be inserted are specified in the request payload. The properties are parsed by the REST data service and then set to the correspondent property on the entity instance. For the AtomPub format, the property is specified as a <d:PROPERTY_NAME> XML element. For JSON, the property is specified as a property of a JSON object.

If a property is missing in the request payload, then the REST data service sets the entity property value to the java default value. However, the database backend might reject such a default value, for example, if the column is not nullable in the database. Then a 500 response code will be returned to indicate an Internal Server error.

If there are duplicate properties specified in the payload, the last property will be used. All the previous values for the same property name are ignored by the REST data service.

If the payload contains a non-existent property, then the REST data service returns a 400 (Bad Request) response code to indicate the request sent by the client was syntactically incorrect.

If the key properties are missing, then the REST data service returns a response code of 400 (Bad Request) to indicate a missing key property.

If the payload contains a link to a related entity with a non-existent key, then the REST data service returns a 404 (Not Found) response code to indicate the linked entity cannot be found.

If the payload contains a link to a related entity with an incorrect association name, then the REST data service returns a 400 (Bad Request) response code to indicate the link cannot be found.

If the payload contains more than one link to a to-one relation, the last link will be used. All the previous links for the same association are ignored.

For more details on the InsertEntity request, see MSDN Library: InsertEntity Request.

An InsertEntity request inserts a Customer entity with key 'IBM'.

AtomPub

- Method: POST
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')`
- Request Header: Accept: application/atom+xml Content-Type: application/atom+xml
- Request Payload:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
  xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
  xmlns="http://www.w3.org/2005/Atom">
  <category term="NorthwindGridModel.Customer"
    scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <content type="application/xml">
    <m:properties>
      <d:customerId>Rational</d:customerId>
      <d:city>Rochester</d:city>
      <d:companyName>Rational</d:companyName>
      <d:contactName>John Doe</d:contactName>
      <d:country>USA</d:country>
    </m:properties>
  </content>
</entry>
```

- Response Header: Content-Type: application/atom+xml
- Response Payload:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
  xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
  xmlns="http://www.w3.org/2005/Atom">
  <category term="NorthwindGridModel.Customer"
    scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <content type="application/xml">
    <m:properties>
      <d:customerId>Rational</d:customerId>
      <d:city>Rochester</d:city>
      <d:companyName>Rational</d:companyName>
      <d:contactName>John Doe</d:contactName>
      <d:country>USA</d:country>
    </m:properties>
  </content>
</entry>
```

Response Header:

Content-Type: application/atom+xml

Response Payload:

```
<?xml version="1.0" encoding="utf-8"?>
<entry xml:base = "http://localhost:8080/wxsrestservice/restservice" xmlns:d =
  "http://schemas.microsoft.com/ado/2007/08/dataservices" xmlns:m =
  "http://schemas.microsoft.com/
```

```

ado/2007/08/dataservices/metadata" xmlns = "http://www.w3.org/2005/Atom">
  <category term = "NorthwindGridModel.Customer" scheme = "http://schemas.
    microsoft.com/ado/2007/08/dataservices/scheme"/>
  <id>http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/
    Customer('Rational')</id>
  <title type = "text"/>
  <updated>2009-12-16T23:25:50.875Z</updated>
  <author>
    <name/>
  </author>
  <link rel = "edit" title = "Customer" href = "Customer('Rational')"/>
  <link rel = "http://schemas.microsoft.com/ado/2007/08/dataservices/related/
    orders" type = "application/atom+xml;type=feed"
    title = "orders" href = "Customer('Rational')/orders"/>
  <content type = "application/xml">
    <m:properties>
      <d:customerId>Rational</d:customerId>
      <d:city>Rochester</d:city>
      <d:companyName>Rational</d:companyName>
      <d:contactName>John Doe</d:contactName>
      <d:country>USA</d:country>
      <d:version m:type = "Edm.Int32">0</d:version>
    </m:properties>
  </content>
</entry>

```

- Response Code: 201 Created

JSON

- Method: POST
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer`
- Request Header: Accept: application/json Content-Type: application/json
- Request Payload:


```

{"customerId":"Rational",
 "city":null,
 "companyName":"Rational",
 "contactName":"John Doe",
 "country": "USA",}

```
- Response Header: Content-Type: application/json
- Response Payload:


```

{"d":{"__metadata":{"uri":"http://localhost:8080/wxsrestservice/restservice/
  NorthwindGrid/Customer('Rational')",
  "type":"NorthwindGridModel.Customer"},
  "customerId":"Rational",
  "city":null,
  "companyName":"Rational",
  "contactName":"John Doe",
  "country":"USA",
  "version":0,
  "orders":{"__deferred":{"uri":"http://localhost:8080/wxsrestservice/restservice/
    NorthwindGrid/Customer('Rational')/orders"}}}}

```
- Response Code: 201 Created

Insert link request

An InsertLink Request can be used to create a new Link between two eXtreme Scale entity instances. The URI of the request must resolve to an eXtreme Scale to-many association. The payload of the request contains a single link which points to the to-many association target entity.

If the URI of the InsertLink request represents a to-one association, the REST data service returns a 400 (Bad request) response.

If the URI of the InsertLink request points to an association which does not exist, the REST data service returns a 404 (Not Found) response to indicate the link cannot be found.

If the payload contains a link with a key which does not exist, the REST data service returns a 404 (Not Found) response to indicate the linked entity cannot be found.

If the payload contains more than one link, the eXtreme Scale Rest Data Service will parse the first link. The remaining links are ignored.

For more details on InsertLink request, refer to: MSDN Library: InsertLink Request.

The following InsertLink request example creates a link from Customer('IBM') to Order(orderId=5000,customer_customerId='IBM').

AtomPub

- Method: POST
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')/$link/orders`
- Request Header: Content-Type: application/xml
- Request Payload:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<uri>http://host:1000/wxsrestservice/restservice/NorthwindGrid/Order(orderId=
5000,customer_customerId='IBM')</uri>
```
- Response Payload: None
- Response Code: 204 No Content

JSON

- Method: POST
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')/$links/orders`
- Request Header: Content-Type: application/json
- Request Payload:

```
{"uri": "http://host:1000/wxsrestservice/restservice/NorthwindGrid/Order(orderId
=5000,customer_customerId='IBM')"}}
```
- Response Payload: None
- Response Code: 204 No Content

Update requests with REST data services

The WebSphere eXtreme Scale REST data service supports update requests for entities, entity primitive properties, and so on.

Update an entity

An UpdateEntity Request can be used to update an existing eXtreme Scale entity. The client can use an HTTP PUT method to replace an existing eXtreme Scale entity, or use an HTTP MERGE method to merge the changes into an existing eXtreme Scale entity.

When updating the entity, the client can specify if the entity, in addition to being updated, must be automatically linked to other existing entities in the data service that are related through single valued to-one associations.

The property of the entity to be updated is in the request payload. The property is parsed by the REST data service and then set to the correspondent property on the entity. For the AtomPub format, the property is specified as a <d:PROPERTY_NAME> XML element. For JSON, the property is specified as a property of a JSON object.

If a property is missing in the request payload, the REST data service sets the entity property value to the Java default value for HTTP PUT method. However, the database backend might reject such a default value if, for example, the column is not nullable in the database. Then a 500 (Internal Server Error) response code is returned to indicate an Internal Server Error. If a property is missing in the HTTP MERGE request payload, the REST data service does not change the existing property value.

If there are duplicate properties specified in the payload, the last property is used. All the previous values with the same property name are ignored by the REST data service.

If the payload contains a non-existent property, the REST data service returns a 400 (Bad Request) response code to indicate the request sent by the client was syntactically incorrect.

As part of the serialization of a resource, if the payload of an Update request contains any of the key properties for the entity, the REST data service ignores those key values since entity keys are immutable.

For details on UpdateEntity request, refer to: MSDN Library: UpdateEntity Request.

An UpdateEntity request updates the city name of Customer('IBM') to 'Raleigh'.

AtomPub

- Method: PUT
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/ Customer('IBM')`
- Request Header: Content-Type: `application/atom+xml`
- Request Payload:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<entry xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices"
xmlns:m="http://schemas.microsoft.com/ado/2007/08/dataservices/metadata"
xmlns="http://www.w3.org/2005/Atom">
  <category term="NorthwindGridModel.Customer"
  scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
  <title />
  <updated>2009-07-28T21:17:50.609Z</updated>
  <author>
    <name />
  </author>
  <id />
  <content type="application/xml">
    <m:properties>
      <d:customerId>IBM</d:customerId>
      <d:city>Raleigh</d:city>
      <d:companyName>IBM Corporation</d:companyName>
      <d:contactName>Big Blue</d:contactName>
    </m:properties>
  </content>
</entry>
```

```
<d:country>USA</d:country>
</m:properties>
</content>
</entry>
```

- Response Payload: None
- Response Code: 204 No Content

JSON

- Method: PUT
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')`
- Request Header: Content-Type: application/json
- Request Payload:

```
{ "customerId": "IBM",
  "city": "Raleigh",
  "companyName": "IBM Corporation",
  "contactName": "Big Blue",
  "country": "USA", }
```
- Response Payload: None
- Response Code: 204 No Content

Update an entity primitive property

The UpdatePrimitiveProperty Request can update a property value of an eXtreme Scale entity. The property and value to be updated are in the request payload. The property cannot be a key property since eXtreme Scale does not allow clients to change entity keys.

For more details on the UpdatePrimitiveProperty request, refer to: MSDN Library: UpdatePrimitiveProperty Request.

Here is an UpdatePrimitiveProperty request example. In this example, we update the city name of Customer('IBM') to 'Raleigh'.

AtomPub

- Method: PUT
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')/city`
- Request Header: Content-Type: application/xml
- Request Payload:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<city xmlns:d="http://schemas.microsoft.com/ado/2007/08/dataservices">
  Raleigh
</city>
```
- Response Payload: None
- Response Code: 204 No Content

JSON

- Method: PUT
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')/city`
- Request Header: Content-Type: application/json
- Request Payload: `{"city": "Raleigh"}`

- Response Payload: None
- Response Code: 204 No Content

Update an entity primitive property value

The UpdateValue Request can update a raw property value of an eXtreme Scale entity. The value to be updated is represented as a raw value in the request payload. The property cannot be a key property since eXtreme Scale does not allow clients to change entity keys.

The content type of the request can be “text/plain” or “application/octet-stream” depending on the property type. For more information, see “Retrieving non-entities with REST data services” on page 265.

For more details on the UpdateValue request, refer to: MSDN Library: UpdateValue Request

Here is an UpdateValue request example. In this example, update the city name of Customer('IBM') to 'Raleigh'.

- Method: PUT
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')/city/$value`
- Request Header: Content-Type: text/plain
- Request Payload: Raleigh
- Response Payload: None
- Response Code: 204 No Content

Update a link

The UpdateLink request can be used to establish an association between two eXtreme Scale entity instances. The association can be a single valued (to-one) relation or a multi-valued (to-many) relation.

Updating a link between two eXtreme Scale entity instances can establish associations or remove associations. For example, if the client establishes a to-one association between an Order(orderId=5000,customer_customerId='IBM') entity and Customer('ALFKI') instance, it has to dissociate the Order(orderId=5000,customer_customerId='IBM') entity and entity from its currently associated Customer instance.

If either of the entity instances specified in the UpdateLink request cannot be found, the REST data service returns a 404 (Not Found) response.

If the URI of the UpdateLink request specifies a non-existent association, the REST data service returns a 404 (Not Found) response to indicate the link cannot be found.

If the URI specified in the UpdateLink request payload does not resolve to the same entity or the same key as specified in the URI, if exists, then the eXtreme Scale Rest Data Service returns a 400 (Bad Request) response.

If the UpdateLink request payload contains multiple links, then the REST data service parses the first link only. The rest of the links are ignored.

For more details on the UpdateLink request, refer to: MSDN Library: UpdateLink Request.

Here is an UpdateLink request example. In this example, we update the customer relation of Order(orderId=5000,customer_customerId='IBM') entity and from Customer('IBM') to Customer('IBM').

Remember: The previous example is for illustration only. Because all associations are typically key-associations for a partitioned grid, the link cannot be changed.

AtomPub

- Method: PUT
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(101)/$links/customer`
- Request Header: Content-Type: application/xml
- Request Payload:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<uri>
  http://host:1000/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')
</uri>
```
- Response Payload: None
- Response Code: 204 No Content

JSON

- Method: PUT
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(orderId=5000,customer_customerId='IBM')/$links/customer`
- Request Header: Content-Type: application/xml
- Request Payload: `{"uri": "http://host:1000/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')"}`
- Response Payload: None
- Response Code: 204 No Content

Delete requests with REST data services

The WebSphere eXtreme Scale REST data service can delete entities, property values and links.

Delete an entity

The DeleteEntity Request can delete an eXtreme Scale entity from the REST data service.

If any relation to the to-be-deleted entity has cascade-delete set, then the eXtreme Scale Rest data service will delete the related entity or entities. For more details on the DeleteEntity request, refer to MSDN Library: DeleteEntity Request.

The following DeleteEntity request deletes the customer with key 'IBM'.

- Method: DELETE
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')`
- Request Payload: None

- Response Payload: None
- Response Code: 204 No Content

Delete a property value

The DeleteValue Request sets an eXtreme Scale entity property to null.

Any property of an eXtreme Scale entity can be set to null with a DeleteValue request. To set a property to null, ensure all of the following:

- For any primitive number type and its wrapper, BigInteger, or BigDecimal, the property value is set to 0.
- For Boolean or boolean type, the property value is set to false.
- For char or Character type, the property value is set to character #X1 (NIL).
- For enum type, the property value is set to the enum value with ordinal 0.
- For all other types, the property value is set to null.

However, such a delete request could be rejected by the database backend if, for example, the property is not nullable in the database. In this case, the REST data service returns a 500 (Internal Server Error) response. For more details on the DeleteValue request, refer to: MSDN Library: DeleteValue Request.

Here is a DeleteValue request example. In this example, we set the contact name of Customer('IBM') to null.

- Method: DELETE
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Customer('IBM')/contactName`
- Request Payload: None
- Response Payload: None
- Response Code: 204 No Content

Delete a link

The DeleteLink request can remove an association between two eXtreme Scale entity instances. The association can be a to-one relation or a to-many relation. However, such a delete request could be rejected by the database backend if, for example, the foreign key constraint is set. In this case, the REST data service returns a 500 (Internal Server Error) response. For more details on the DeleteLink request, refer to: MSDN Library: DeleteLink Request.

The following DeleteLink request removes the association between Order(101) and its associated Customer.

- Method: DELETE
- Request URI: `http://localhost:8080/wxsrestservice/restservice/NorthwindGrid/Order(101)/$links/customer`
- Request Payload: None
- Response Payload: None
- Response Code: 204 No Content

System APIs and plug-ins

A plug-in is a component that provides a function to the pluggable components, which include ObjectGrid and BackingMap. To most effectively use eXtreme Scale as an in-memory data grid or database processing space, you should carefully determine how best you can maximize performance with available plug-ins.

Managing plug-in life cycles

You can manage plug-in life cycles with specialized methods from each plug-in, which are available to be invoked at designated functional points. Both initialize and destroy methods define the life cycle of plug-ins, which are controlled by their *owner* objects. An owner object is the object that actually uses the given plug-in. An owner can be a grid client, server, or a backing map.

About this task

Similarly all plug-ins can implement the optional mix-in interfaces appropriate for their owner object. Any ObjectGrid plug-in can implement the optional mix-in interface ObjectGridPlugin. Any BackingMap plug-in can implement the optional mix-in interface BackingMapPlugin. The optional mix-in interfaces require implementation of several additional methods beyond the initialize() and destroy() methods for the basic plug-ins. For more information about these interfaces, see the API documentation.

When owner objects are initializing, those objects set attributes on the plug-in, then invoke the initialize method of their owned plug-ins. During the destroy cycle of owner objects, the destroy method of plug-ins are consequently invoked also. For details on the specifics of initialize and destroy methods, along with other methods capable with each plug-in, refer to the topics relevant to each plug-in.

As an example, consider a distributed environment. Both the client-side ObjectGrids and the server-side ObjectGrids can have their own plug-ins. The life cycle of a client-side ObjectGrid, and therefore, its plug-in instances are independent from all server-side ObjectGrid and plug-in instances.

In such a distributed topology, assume that you have an ObjectGrid named myGrid defined in the objectGrid.xml file and configured with a customized ObjectGridEventListener named myObjectGridEventListener. The objectGridDeployment.xml file defines the deployment policy for the myGrid ObjectGrid. Both the objectGrid.xml and objectGridDeployment.xml files are used to start container servers. During the startup of the container server, the server-side myGrid ObjectGrid instance is initialized. Meanwhile, the initialize method of the myObjectGridEventListener instance that is owned by the myObjectGrid instance is invoked. After the container server is started, your application can connect to the server-side myGrid ObjectGrid instance and obtain a client-side instance.

When obtaining the client-side myGrid ObjectGrid instance, the client-side myGrid instance goes through its own initialization cycle and invokes the initialize method of its own client-side myObjectGridEventListener instance. This client-side myObjectGridEventListener instance is independent from the server-side myObjectGridEventListener instance. Its life cycle is controlled by its owner, which is the client-side myGrid ObjectGrid instance.

If your application disconnects or destroys the client-side myGrid ObjectGrid instance, then the destroy method that belongs to the client-side

myObjectGridEventListener instance is invoked automatically. However, this process has no impact on server-side myObjectGridEventListener instance. The destroy method of the server-side myObjectGridEventListener instance can only be invoked during the destroy life cycle of the server-side myGrid ObjectGrid instance, when stopping a container server. Specifically, when stopping a container server, the contained ObjectGrid instances are destroyed and the destroy method of all their owned plug-ins is invoked.

Although the previous example applies specifically to the case of a client and a server instance of an ObjectGrid, the owner of a plug-in can also be a BackingMap interface. In addition, carefully to determine your configurations for plug-ins that you might write, based on these life cycle considerations. Use the following topics to write plug-ins that provide extended life cycle management events that you can use to set up or remove resources in your environment:

Writing an ObjectGridPlugin plug-in

An ObjectGridPlugin is an optional mix-in interface that you can use to provide extended life cycle management events to all other ObjectGrid plug-ins.

About this task

Any ObjectGrid plug-in that implements the ObjectGridPlugin receives the extended set of life cycle events, and can provide more control, which you can use to set up or remove resources. In a container for a partitioned data grid, there will be one ObjectGrid instance (the plugin owner) for each partition managed by the container. When individual partitions are removed, the resources that are used by that ObjectGrid instance must also be removed. Therefore, you might need to close or end a resource, such as an open configuration file or a running thread that is managed by a plug-in, when the owning partition for that resource is removed.

The ObjectGridPlugin interface provides methods to set or modify the state of the plug-in, as well as methods to introspect the current state of the plug-in. All methods must be implemented correctly, and the WebSphere eXtreme Scale runtime environment verifies the method behavior under certain circumstances. For example, after calling the initialize() method, the eXtreme Scale runtime environment calls the isInitialized() method to ensure that the method successfully completed the appropriate initialization.

Procedure

1. Implement the ObjectGridPlugin interface so that the ObjectGridPlugin plug-in receives notifications about significant eXtreme Scale events. Three main categories of methods exist:

Properties methods

setObjectGrid()

getObjectGrid()

Purpose

Called to set the ObjectGrid instance the plug-in is used for.

Called to get or confirm the ObjectGrid instance the plug-in is used for.

Initialization methods

initialize()

isInitialized()

Purpose

Called to initialize the ObjectGridPlugin.

Called to get or confirm the initialization status of the plug-in.

Destruction methods

destroy()

Purpose

Called to destroy the ObjectGridPlugin.

Destruction methods

isDestroyed()

Purpose

Called to get or confirm the destroyed status of the plug-in.

See the API documentation for more information about these interfaces.

2. Configure an ObjectGridPlugin plug-in with XML. Use the `com.company.org.MyObjectGridPluginTxCallback` class, which implements the TransactionCallback interface and the ObjectGridPlugin interface.

In the following code example, the custom transaction callback, which will ultimately receive extended life cycle events, is generated and added to an ObjectGrid.

Important: The TransactionCallback interface already has an initialize method, a new initialize method is added as well as the destroy method and other ObjectGridPlugin methods. Each method is used, and the initialize methods only perform initialization one time. The following XML creates a configuration that uses the enhanced TransactionCallback interface.

The following text must be in the `myGrid.xml` file:

```
?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="myGrid">
      <bean id="TransactionCallback"
        className="com.company.org.MyObjectGridPluginTxCallback" />
      <backingMap name="Book"/>
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

Notice the bean declarations come before the backingMap declarations.

3. Provide the `myGrid.xml` file to the ObjectGridManager plug-in to facilitate the creation of this configuration.

Writing a BackingMapPlugin plug-in

A BackingMap plug-in implements the BackingMapPlugin mix-in interface, which you can use to receive extended capabilities for managing its life cycle.

About this task

Any existing BackingMap plug-in that also implements the BackingMapPlugin interface will automatically receive the extended set of lifecycle events during its construction and use.

The BackingMapPlugin interface provides methods to set or modify the state of the plug-in, as well as methods to introspect the current state of the plug-in.

All methods must be implemented correctly, and the WebSphere eXtreme Scale runtime environment verifies the method behavior under certain circumstances. For example, after calling the `initialize()` method, the eXtreme Scale runtime environment calls the `isInitialized()` method to ensure that the method successfully completed the appropriate initialization.

Procedure

1. Implement the BackingMapPlugin interface so that the BackingMapPlugin plug-in receives notifications about significant eXtreme Scale events. Three main categories of methods exist:

Properties methods

setBackingMap()

getBackingMap()

Purpose

Called to set the BackingMap instance the plug-in is used for.

Called to get or confirm the BackingMap instance the plug-in is used for.

Initialization methods

initialize()

isInitialized()

Purpose

Called to initialize the BackingMapPlugin plug-in.

Called to get or confirm the initialization status of the plug-in.

Destruction methods

destroy()

isDestroyed()

Purpose

Called to destroy the BackingMapPlugin plug-in.

Called to get or confirm the destroyed status of the plug-in.

See the API documentation for more information about these interfaces.

2. Configure a BackingMapPlugin plug-in with XML. Assume that the class name of an eXtreme Scale Loader plug-in is the `com.company.org.MyBackingMapPluginLoader` class, which implements the Loader interface and the BackingMapPlugin interface.

In the following code example, the custom transaction callback, which will ultimately receive extended life cycle events, is generated and added to a BackingMap.

You can also configure a BackingMapPlugin plug-in using XML. The following text must be in the `myGrid.xml` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridconfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config../objectGrid.xsd"
xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="myGrid">
      <backingMap name="Book" pluginCollectionRef="myPlugins" />
    </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
    <backingMapPluginCollection id="myPlugins">
      <bean id="Loader"
        className="com.company.org.MyBackingMapPluginLoader" />
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

3. Provide the `myGrid.xml` file to the ObjectGridManager plug-in to facilitate the creation of this configuration.

Results

The BackingMap instance that is created has a Loader that receives BackingMapPlugin life cycle events.

Plug-ins for multimaster replication

Consider transforming cached objects to increase the performance of your cache. You can use the ObjectTransformer plug-in when your processor usage is high. Up

to 60-70 percent of the total processor time is spent serializing and copying entries. By implementing the ObjectTransformer plug-in, you can serialize and deserialize objects with your own implementation. You can use a CollisionArbiter plug-in to define how change collisions are handled in your domains.

Developing custom arbiters for multi-master replication

Change collisions might occur if the same records can be changed simultaneously in two places. In a multi-master replication topology, catalog service domains detect collisions automatically. When a catalog service domain detects a collision, it invokes an arbiter. Typically, collisions are resolved with the default collision arbiter. However, an application can provide a custom collision arbiter.

Before you begin

- See “Planning multiple data center topologies” on page 104 for more information about planning and designing the multi-master replication topology.
- See Configuring multiple data center topologies for more information about setting up links between catalog service domains.

About this task

If a catalog service domain receives a replicated entry that collides with a collision record, the default arbiter uses the changes from the lexicographically lowest named catalog service domain. For example, if domain A and B generate a conflict for a record, then the change from domain B is ignored. Domain A keeps its version and the record in domain B is changed to match the record from domain A. Domain names are converted to uppercase for comparison.

An alternative option is for the multi-master replication topology to call on a custom collision plug-in to decide the outcome. These instructions outline how to develop a custom collision arbiter and configure a multi-master replication topology to use it.

Procedure

1. Develop a custom collision arbiter and integrate it into your application.

The class must implement the interface:

```
com.ibm.websphere.objectgrid.revision.CollisionArbiter
```

A collision plug-in has three choices for deciding the outcome of a collision. It can choose the local copy or the remote copy or it can provide a revised version of the entry. A catalog service domain provides the following information to a custom collision arbiter:

- The existing version of the record
- The collision version of the record
- A Session object that must be used to create the revised version of the collided entry

The plug-in method returns an object that indicates its decision. The method invoked by the domain to call the plug-in must return true or false, where false means to ignore the collision. When the collision is ignored, the local version remains unchanged and the arbiter forgets that it ever saw the existing version. The method returns a true value if the method used the provided session to create a new, merged version of the record, reconciling the change.

2. In the objectgrid.xml file, specify the custom arbiter plug-in.

The ID must be CollisionArbiter.

```

<dg:objectGrid name="revisionGrid" txTimeout="10">
  <dg:bean className="com.you.your_application.
    CustomArbiter" id="CollisionArbiter">
    <dg:property name="property" type="java.lang.String"
      value="propertyValue"/>
  </dg:bean>
</dg:objectGrid>

```

Plug-ins for versioning and comparing cache objects

Use the OptimisticCallback plug-in to customize versioning and comparison operations of cache objects when you are using the optimistic locking strategy.

You can provide a pluggable optimistic callback object that implements the `com.ibm.websphere.objectgrid.plugins.OptimisticCallback` interface. For entity maps, a high performance OptimisticCallback plug-in is automatically configured.

Purpose

Use the OptimisticCallback interface to provide optimistic comparison operations for the values of a map. An OptimisticCallback plug-in is required when you use the optimistic locking strategy. The product provides a default OptimisticCallback implementation. However, typically your application must plug in its own implementation of the OptimisticCallback interface.

Default implementation

The eXtreme Scale framework provides a default implementation of the OptimisticCallback interface that is used if the application does not plug in an application-provided OptimisticCallback object. The default implementation always returns the special value of `NULL_OPTIMISTIC_VERSION` as the version object for the value and never updates the version object. This action makes optimistic comparison a "no operation" function. In most cases, you do not want the "no operation" function to occur when you are using the optimistic locking strategy. Your applications must implement the OptimisticCallback interface and plug in their own OptimisticCallback implementations so that the default implementation is not used. However, at least one scenario exists where the default provided OptimisticCallback implementation is useful. Consider the following situation:

- A loader is plugged in for the backing map.
- The loader knows how to perform the optimistic comparison without assistance from an OptimisticCallback plug-in.

How can the loader perform optimistic versioning without assistance from an OptimisticCallback object? The loader has knowledge of the value class object and knows which field of the value object is used as an optimistic versioning value. For example, suppose the following interface is used for the value object for the employees map:

```

public interface Employee
{
    // Sequential sequence number used for optimistic versioning.
    public long getSequenceNumber();
    public void setSequenceNumber(long newSequenceNumber);
    // Other get/set methods for other fields of Employee object.
}

```

In this example, the loader knows that it can use the `getSequenceNumber` method to get the current version information for an Employee value object. The loader increments the returned value to generate a new version number before it updates

the persistent storage with the new Employee value. For a Java database connectivity (JDBC) loader, the current sequence number in the WHERE clause of an overqualified SQL UPDATE statement is used, and it uses the new generated sequence number to set the sequence number column to the new sequence number value. Another possibility is that the loader makes use of some backend-provided function that automatically updates a hidden column that can be used for optimistic versioning.

In some situations, a stored procedure or trigger can possibly be used to help maintain a column that holds versioning information. If the loader is using one of these techniques for maintaining optimistic versioning information, then the application does not need to provide an OptimisticCallback implementation. The default OptimisticCallback implementation is usable in this scenario because the loader can handle optimistic versioning without any assistance from an OptimisticCallback object.

Default implementation for entities

Entities are stored in the ObjectGrid using tuple objects. The default OptimisticCallback implementation behavior is similar to the behavior for non-entity maps. However, the version field in the entity is identified using the @Version annotation or the version attribute in the entity descriptor XML file.

The version attribute can be of the following types: int, Integer, short, Short, long, Long or java.sql.Timestamp. An entity must only have one version attribute defined. Only set the version attribute during construction. After the entity is persisted, the value of the version attribute should not be modified.

If a version attribute is not configured and the optimistic locking strategy is used, then the entire tuple is implicitly versioned using the entire state of the tuple, which is much more expensive

In the following example, the Employee entity has a long version attribute named SequenceNumber:

```
@Entity
public class Employee
{
    private long sequence;
        // Sequential sequence number used for optimistic versioning.
        @Version
        public long getSequenceNumber() {
            return sequence;
        }
        public void setSequenceNumber(long newSequenceNumber) {
            this.sequence = newSequenceNumber;
        }
        // Other get/set methods for other fields of Employee object.
    }
}
```

Writing an OptimisticCallback plug-in

An OptimisticCallback plug-in must to implement the OptimisticCallback interface and follow the common ObjectGrid plug-in conventions. See the OptimisticCallback interface in the API documentation for more information.

The following list provides a description or consideration for each of the methods in the OptimisticCallback interface:

NULL_OPTIMISTIC_VERSION

This special value is returned by the `getVersionedObjectForValue` method if the `OptimisticCallback` implementation does not require version checking. The built-in plugin implementation of the `com.ibm.websphere.objectgrid.plugins.builtins.NoVersioningOptimisticCallback` class uses this value because versioning is disabled when you are specifying this plug-in implementation.

getVersionedObjectForValue method

The `getVersionedObjectForValue` method might return a copy of the value or an attribute of the value that can be used for versioning purposes. This method is called whenever an object is associated with a transaction. When no Loader is plugged into a backing map, the backing map uses this value at commit time to perform an optimistic version comparison. The optimistic version comparison is used by the backing map to ensure that the version has not changed after this transaction first accessed the map entry that was modified by this transaction. If another transaction had already modified the version for this map entry, the version comparison fails and the backing map displays an `OptimisticCollisionException` exception to force the transaction to roll back. If a Loader is plugged in, the backing map does not use the optimistic versioning information. Instead, the Loader is responsible for performing the optimistic versioning comparison and updating the versioning information when necessary. The Loader typically gets the initial versioning object from the `LogElement` passed to the `batchUpdate` method on the loader, which is called when a flush operation occurs or a transaction is committed.

The following code shows the implementation used by the `EmployeeOptimisticCallbackImpl` object:

```
public Object getVersionedObjectForValue(Object value)
{
    if (value == null)
    {
        return null;
    }
    else
    {
        Employee emp = (Employee) value;
        return new Long( emp.getSequenceNumber() );
    }
}
```

As demonstrated in the previous example, the `sequenceNumber` attribute is returned in a `java.lang.Long` object as expected by the Loader, which implies that the same person that wrote the Loader either wrote the `EmployeeOptimisticCallbackImpl` implementation or worked closely with the person that implemented the `EmployeeOptimisticCallbackImpl` - for example, agreed on the value returned by the `getVersionedObjectForValue` method. The default `OptimisticCallback` plug-in returns the special value `NULL_OPTIMISTIC_VERSION` as the version object.

updateVersionedObjectForValue method

This method is called whenever a transaction has updated a value and a new versioned object is needed. If the `getVersionedObjectForValue` method returns an attribute of the value, this method typically updates the attribute value with a new version object. If `getVersionedObjectForValue` method returns a copy of the value,

this method typically does not complete any actions. The default `OptimisticCallback` plug-in does not complete any actions with this method because the default implementation of `getVersionedObjectForValue` always returns the special value `NULL_OPTIMISTIC_VERSION` as the version object. The following example shows the implementation used by the `EmployeeOptimisticCallbackImpl` object that is used in the `OptimisticCallback` section:

```
public void updateVersionedObjectForValue(Object value)
{
    if ( value != null )
    {
        Employee emp = (Employee) value;
        long next = emp.getSequenceNumber() + 1;
        emp.updateSequenceNumber( next );
    }
}
```

As demonstrated in the previous example, the `sequenceNumber` attribute increments by one so that the next time the `getVersionedObjectForValue` method is called, the `java.lang.Long` value that is returned has a long value that is the original sequence number value plus one, for example, is the next version value for this employee instance. This example implies that the same person that wrote the `Loader` either wrote `EmployeeOptimisticCallbackImpl` or worked closely with the person that implemented the `EmployeeOptimisticCallbackImpl`.

serializeVersionedValue method

This method writes the versioned value to the specified stream. Depending on the implementation, the versioned value can be used to identify optimistic update collisions. In some implementations, the versioned value is a copy of the original value. Other implementations might have a sequence number or some other object to indicate the version of the value. Because the actual implementation is unknown, this method is provided to perform the appropriate serialization. The default implementation calls the `writeObject` method.

inflateVersionedValue method

This method takes the serialized version of the versioned value and returns the actual versioned value object. Depending on the implementation, the versioned value can be used to identify optimistic update collisions. In some implementations, the versioned value is a copy of the original value. Other implementations might have a sequence number or some other object to indicate the version of the value. Because the actual implementation is unknown, this method is provided to perform the appropriate deserialization. The default implementation calls the `readObject` method.

Using application-provided `OptimisticCallback` object

You have two approaches to add an application-provided `OptimisticCallback` object into the `BackingMap` configuration: XML configuration and programmatic configuration.

Programmatically plug in an `OptimisticCallback` object

The following example demonstrates how an application can programmatically plug in an `OptimisticCallback` object for the employee backing map in the local `grid1` `ObjectGrid` instance:

```

import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid1" );
BackingMap bm = dg.defineMap("employees");
EmployeeOptimisticCallbackImpl cb = new EmployeeOptimisticCallbackImpl();
bm.setOptimisticCallback( cb );

```

XML configuration approach to plug in an OptimisticCallback object

The application can use an XML file to plug in its OptimisticCallback object as shown in the following example:

```

<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
  <objectGrid name="grid1">
    <backingMap name="employees" pluginCollectionRef="employees" lockStrategy="OPTIMISTIC" />
  </objectGrid>
</objectGrids>


<backingMapPluginCollections>
  <backingMapPluginCollection id="employees">
    <bean id="OptimisticCallback" className="com.xyz.EmployeeOptimisticCallbackImpl" />
  </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>

```

Plug-ins for serializing cached objects

WebSphere eXtreme Scale uses multiple Java processes to serialize the data, by converting the Java object instances to bytes and back to objects again, as needed, to move the data between client and server processes.

To serialize data in eXtreme Scale, you can use Java serialization, the ObjectTransformer plug-in, or the DataSerializer plug-ins.

 The ObjectTransformer interface has been replaced by the DataSerializer plug-ins, which you can use to efficiently store arbitrary data in WebSphere eXtreme Scale so that existing product APIs can efficiently interact with your data.

Serializer programming overview

You can use the DataSerializer plug-ins to write optimized serializers for storing Java objects and other data in binary form in the grid. The plug-in also provides methods that you can use to query attributes within the binary data without requiring the entire data object to be inflated.

The DataSerializer plug-ins include three main plug-ins and several optional mix-in interfaces. The MapSerializerPlugin plug-in includes metadata about the relationship between a map and other maps. It also includes a reference to a KeySerializerPlugin and ValueSerializerPlugin. The key and value serializer plug-ins include metadata and serialization code responsible for interacting with the respective key and value data for a map. A MapSerializerPlugin plug-in must include one or both key and value serializers.

The KeySerializerPlugin plug-in provides methods and metadata for serializing, inflating and introspecting keys. The ValueSerializer plug-in provides methods and metadata for serializing, inflating and introspecting values. Both interfaces have different requirements. For details on what methods are available on the

DataSerializer plug-ins, see the API documentation for the `com.ibm.websphere.objectgrid.plugins.io` package.

MapSerializerPlugin plug-in

The `MapSerializerPlugin` is the main plug-in point to the `BackingMap` interface, and it includes two nested plug-ins: the `KeySerializerPlugin` and `ValueSerializerPlugin` plug-ins. Since eXtreme Scale does not support nested or wired plug-ins, the `BasicMapSerializerPlugin` plug-in accesses these nested plug-ins artificially. When you use these plug-ins with the OSGi framework, the only proxy is the `MapSerializerPlugin` plug-in. All nested plug-ins must not be cached within other dependent plug-ins, such as loaders, unless those plug-ins also listen for `BackingMap` life cycle events. This is important when running in an OSGi framework, because references to those plug-ins can continue to be refreshed.

KeySerializerPlugin plug-in

The `KeySerializerPlugin` plug-in extends the `DataSerializer` interface and includes other mix-in interfaces and metadata that describes the key. Use this plug-in to serialize and inflate key data objects and attributes.

ValueSerializerPlugin plug-in

The `ValueSerializerPlugin` plug-in extends the `DataSerializer` interface, but exposes no additional methods. Use this plug-in to serialize and inflate value data objects and attributes.

Optional and mix-in interfaces

Optional and mix-in interfaces provide additional capabilities, such as:

Optimistic versioning

The `Versionable` interface allows the `ValueSerializerPlugin` plug-in to handle version checking and version updates when using optimistic locking. If the `Versioning` is not implemented and optimistic locking is enabled, then the version is the entire serialized form of the data object value.

Non-hashCode-based routing

The `Partitionable` interface allows `KeySerializerPlugin` implementations to route requests to explicit partitions. This is equivalent to the `PartitionableKey` interface, when used with the `ObjectMap` API without a `KeySerializerPlugin`. Without this feature, the key is routed to the partition based on the resulting `hashCode`.

UserReadable (toString) interface

The `UserReadable (toString)` interface allows all `DataSerializer` implementations to provide an alternative method to display data in log files and debuggers. With this capability, you can hide sensitive data such as passwords. If `DataSerializer` implementations do not implement this interface, then the runtime environment might call `toString()` directly on the object or include alternative representations, if appropriate.

Evolution support

The `Mergeable` interface can be implemented on `ValueSerializerPlugin` plug-in implementations to allow interoperability between multiple versions of objects when there are different `DataSerializer` versions updating data in the grid through its lifetime. The `Mergeable` methods allow the `DataSerializer` plug-in to retain any data that it might not otherwise understand.

Avoiding object inflation when updating and retrieving cache data

You can use the `DataSerializer` plug-ins to bypass automatic object inflation and manually retrieve attributes from data that has already been serialized. You can also use the `DataSerializer` to insert and update data in its serialized form. This usage can be useful when only part of the data needs to be accessed or when the data needs to be passed between systems.

About this task

This task uses the `COPY_TO_BYTES_RAW` copy mode with the `MapSerializerPlugin` and `ValueSerializerPlugin` plug-ins. The `MapSerializer` is the main plug-in point to the `BackingMap` interface. It includes two nested plug-ins, the `KeyDataSerializer` and `ValueDataSerializer`. Since the product does not support nested plug-ins, the `BaseMapSerializer` supports nested or wired plug-ins artificially. Therefore, when you use these APIs in the OSGi container, the `MapSerializer` is the only proxy. All nested plug-ins must not be cached within other dependent plug-ins, such as a loader, unless it also listens for `BackingMap` life cycle events, so that it can refresh its supporting references.

When `COPY_TO_BYTES_RAW` is set, all `ObjectMap` methods return `SerializedValue` objects, allowing the user to retrieve the serialized form or the Java object form of the value.

When using a `KeySerializerPlugin` plug-in, all methods that return keys, such as the `MapIndexPlugin` or `Loader` plug-ins return `SerializedKey` objects.

When the data is already in serialized form, the data is inserted using the same `SerializedKey` and `SerializedValue` objects. When the data is in `byte[]` format, the `DataObjectKeyFactory` and `DataObjectValueFactory` factories are used to create the appropriate key or value wrapper. The factories are available on the `DataObjectContext`, which can be accessed from the `SerializerAccessor` for the `BackingMap`, or from within the `DataSerializer` implementation.

The example in this topic demonstrates how to complete the following actions:

Procedure

1. Use the `DataSerializer` plug-ins to serialize and inflate data objects.
2. Retrieve serialized values.
3. Retrieve individual attributes from a serialized value.
4. Insert pre-serialized keys and values.

Example

Use this example to update and retrieve cache data:

```
import java.io.IOException;
import com.ibm.websphere.objectgrid.CopyMode;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.io.XsDataOutputStream;
import com.ibm.websphere.objectgrid.plugins.io.SerializerAccessor;
import com.ibm.websphere.objectgrid.plugins.io.ValueSerializerPlugin;
import com.ibm.websphere.objectgrid.plugins.io.dataobject.DataObjectContext;
import com.ibm.websphere.objectgrid.plugins.io.dataobject.SerializedKey;
import com.ibm.websphere.objectgrid.plugins.io.dataobject.SerializedValue;

/**
 * Use the DataSerializer to serialize an Order key.
 */
```

```

public byte[] serializeOrderKey(ObjectGrid grid, String key)
    throws IOException {
    SerializerAccessor sa = grid.getMap("Order").getSerializerAccessor();
    DataObjectContext dftObjCtx = sa.getDefaultContext();
    XsDataOutputStream out = dftObjCtx.getDataStreamManager()
        .createOutputStream();
    sa.getMapSerializerPlugin().getKeySerializerPlugin()
        .serializeDataObject(sa.getDefaultContext(), key, out);
    return out.toByteArray();
}

/**
 * Use the DataSerializer to serialize an Order value.
 */
public byte[] serializeOrderValue(ObjectGrid grid, Order value)
    throws IOException {
    SerializerAccessor sa = grid.getMap("Order").getSerializerAccessor();
    DataObjectContext dftObjCtx = sa.getDefaultContext();
    XsDataOutputStream out = dftObjCtx.getDataStreamManager()
        .createOutputStream();
    sa.getMapSerializerPlugin().getValueSerializerPlugin()
        .serializeDataObject(sa.getDefaultContext(), value, out);
    return out.toByteArray();
}

/**
 * Retrieve a single Order in serialized form.
 */
public byte[] fetchOrderRAWBytes(Session session, String key)
    throws ObjectGridException {
    ObjectMap map = session.getMap("Order");

    // Override the CopyMode to retrieve the serialized form of the value.
    // This process affects all API methods from this point on for the life
    // of the Session.
    map.setCopyMode(CopyMode.COPY_TO_BYTES_RAW, null);
    SerializedValue serValue = (SerializedValue) map.get(key);

    if (serValue == null)
        return null;

    // Retrieve the byte array and return it to the caller.
    return serValue.getInputStream().toByteArray();
}

/**
 * Retrieve one or more attributes from the Order without inflating the
 * Order object.
 */
public Object[] fetchOrderAttribute(Session session, String key,
    String... attributes) throws ObjectGridException, IOException {
    ObjectMap map = session.getMap("Order");

    // Override the CopyMode to retrieve the serialized form of the value.
    // This process affects all API methods from this point on for the life
    // of the Session.
    map.setCopyMode(CopyMode.COPY_TO_BYTES_RAW, null);
    SerializedValue serValue = (SerializedValue) map.get(key);

    if (serValue == null)
        return null;

    // Retrieve a single attribute from the byte buffer.
    ValueSerializerPlugin valSer = session.getObjectGrid()
        .getMap(map.getName()).getSerializerAccessor()
        .getMapSerializerPlugin().getValueSerializerPlugin();
    Object attrCtx = valSer.getAttributeContexts(attributes);
    return valSer.inflateDataObjectAttributes(serValue.getContext(),
        serValue.getInputStream(), attrCtx);
}

/**
 * Inserts a pre-serialized key and value into the Order map.
 */
public void insertRAWOrder(Session session, byte[] key, byte[] value)
    throws ObjectGridException {
    ObjectMap map = session.getMap("Order");

    // Get a referece to the default DataObjectContext for the map.
    DataObjectContext dftDtaObjCtx = session.getObjectGrid()
        .getMap(map.getName()).getSerializerAccessor()
        .getDefaultContext();


    // Wrap the key and value in a SerializedKey and SerializedValue
    // wrapper.
    SerializedKey serKey = dftDtaObjCtx.getKeyFactory().createKey(key);
    SerializedValue serValue = dftDtaObjCtx.getValueFactory().createValue(
        value);
}

```

```
        // Insert the serialized form of the key and value.
        map.insert(serKey, serValue);
    }
}
```

ObjectTransformer plug-in

With the ObjectTransformer plug-in, you can serialize, deserialize, and copy objects in the cache for increased performance.

 The ObjectTransformer interface has been replaced by the DataSerializer plug-ins, which you can use to efficiently store arbitrary data in WebSphere eXtreme Scale so that existing product APIs can efficiently interact with your data.

If you see performance issues with processor usage, add an ObjectTransformer plug-in to each map. If you do not provide an ObjectTransformer plug-in, up to 60-70 percent of the total processor time is spent serializing and copying entries.

Purpose

With the ObjectTransformer plug-in, your applications can provide custom methods for the following operations:

- Serialize or deserialize the key for an entry
- Serialize or deserialize the value for an entry
- Copy a key or value for an entry

If no ObjectTransformer plug-in is provided, you must be able to serialize the keys and values because the ObjectGrid uses a serialize and deserialize sequence to copy the objects. This method is expensive, so use an ObjectTransformer plug-in when performance is critical. The copying occurs when an application looks up an object in a transaction for the first time. You can avoid this copying by setting the copy mode of the Map to NO_COPY or reduce the copying by setting the copy mode to COPY_ON_READ. Optimize the copy operation when needed by the application by providing a custom copy method on this plug-in. Such a plug-in can reduce the copy overhead from 65–70 percent to 2/3 percent of total processor time.

The default copyKey and copyValue method implementations first attempt to use the clone method, if the method is provided. If no clone method implementation is provided, the implementation defaults to serialization.

Object serialization is also used directly when the eXtreme Scale is running in distributed mode. The LogSequence uses the ObjectTransformer plug-in to help serialize keys and values before transmitting the changes to peers in the ObjectGrid. You must take care when providing a custom serialization method instead of using the built-in Java developer kit serialization. Object versioning is a complex issue and you might encounter problems with version compatibility if you do not ensure that your custom methods are designed for versioning.

The following list describes how the eXtreme Scale tries to serialize both keys and values:

- If a custom ObjectTransformer plug-in is written and plugged in, eXtreme Scale calls methods in the ObjectTransformer interface to serialize keys and values and get copies of object keys and values.
- If a custom ObjectTransformer plug-in is not used, eXtreme Scale serializes and deserializes values according to the default. If the default plug-in is used, each object is implemented as externalizable or is implemented as serializable.

- If the object supports the Externalizable interface, the writeExternal method is called. Objects that are implemented as externalizable lead to better performance.
- If the object does not support the Externalizable interface and does implement the Serializable interface, the object is saved using the ObjectOutputStream method.

Using the ObjectTransformer interface

An ObjectTransformer object must implement the ObjectTransformer interface and follow the common ObjectGrid plug-in conventions.

Two approaches, programmatic configuration and XML configuration, are used to add an ObjectTransformer object into the BackingMap configuration as follows.

Programmatically plug in an ObjectTransformer object

The following code snippet creates the custom ObjectTransformer object and adds it to a BackingMap:

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
BackingMap backingMap = myGrid.getMap("myMap");
MyObjectTransformer myObjectTransformer = new MyObjectTransformer();
backingMap.setObjectTransformer(myObjectTransformer);
```

XML configuration approach to plug in an ObjectTransformer

Assume that the class name of the ObjectTransformer implementation is the com.company.org.MyObjectTransformer class. This class implements the ObjectTransformer interface. An ObjectTransformer implementation can be configured using the following XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="myGrid">
      <backingMap name="myMap" pluginCollectionRef="myMap" />
    </objectGrid>
  </objectGrids>

  <backingMapPluginCollections>
    <backingMapPluginCollection id="myMap">
      <bean id="ObjectTransformer" className="com.company.org.MyObjectTransformer" />
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

ObjectTransformer usage scenarios

You can use the ObjectTransformer plug-in in the following situations:

- Non-serializable object
- Serializable object but improve serialization performance
- Key or value copy

In the following example, ObjectGrid is used to store the Stock class:

```
/**
 * Stock object for ObjectGrid demo
 *
 */
public class Stock implements Cloneable {
    String ticket;
```

```

double price;
String company;
String description;
int serialNumber;
long lastTransactionTime;
/**
 * @return Returns the description.
 */
public String getDescription() {
    return description;
}
/**
 * @param description The description to set.
 */
public void setDescription(String description) {
    this.description = description;
}
/**
 * @return Returns the lastTransactionTime.
 */
public long getLastTransactionTime() {
    return lastTransactionTime;
}
/**
 * @param lastTransactionTime The lastTransactionTime to set.
 */
public void setLastTransactionTime(long lastTransactionTime) {
    this.lastTransactionTime = lastTransactionTime;
}
/**
 * @return Returns the price.
 */
public double getPrice() {
    return price;
}
/**
 * @param price The price to set.
 */
public void setPrice(double price) {
    this.price = price;
}
/**
 * @return Returns the serialNumber.
 */
public int getSerialNumber() {
    return serialNumber;
}
/**
 * @param serialNumber The serialNumber to set.
 */
public void setSerialNumber(int serialNumber) {
    this.serialNumber = serialNumber;
}
/**
 * @return Returns the ticket.
 */
public String getTicket() {
    return ticket;
}
/**
 * @param ticket The ticket to set.
 */
public void setTicket(String ticket) {
    this.ticket = ticket;
}
/**
 * @return Returns the company.
 */
public String getCompany() {
    return company;
}
/**
 * @param company The company to set.
 */
public void setCompany(String company) {
    this.company = company;
}
}
//clone
public Object clone() throws CloneNotSupportedException

```



```

    {
        return super.clone();
    }
}

```

You can write a custom object transformer class for the Stock class:

```

/**
 * Custom implementation of ObjectGrid ObjectTransformer for stock object
 */
public class MyStockObjectTransformer implements ObjectTransformer {
    /* (non-Javadoc)
    * @see com.ibm.websphere.objectgrid.plugins.ObjectTransformer#serializeKey
    * (java.lang.Object,
    * java.io.ObjectOutputStream)
    */
    public void serializeKey(Object key, ObjectOutputStream stream) throws IOException {
        String ticket= (String) key;
        stream.writeUTF(ticket);
    }

    /* (non-Javadoc)
    * @see com.ibm.websphere.objectgrid.plugins.
    ObjectTransformer#serializeValue(java.lang.Object,
    java.io.ObjectOutputStream)
    */
    public void serializeValue(Object value, ObjectOutputStream stream) throws IOException {
        Stock stock= (Stock) value;
        stream.writeUTF(stock.getTicket());
        stream.writeUTF(stock.getCompany());
        stream.writeUTF(stock.getDescription());
        stream.writeDouble(stock.getPrice());
        stream.writeLong(stock.getLastTransactionTime());
        stream.writeInt(stock.getSerialNumber());
    }

    /* (non-Javadoc)
    * @see com.ibm.websphere.objectgrid.plugins.
    ObjectTransformer#inflateKey(java.io.ObjectInputStream)
    */
    public Object inflateKey(ObjectInputStream stream) throws IOException, ClassNotFoundException {
        String ticket=stream.readUTF();
        return ticket;
    }

    /* (non-Javadoc)
    * @see com.ibm.websphere.objectgrid.plugins.
    ObjectTransformer#inflateValue(java.io.ObjectInputStream)
    */
    public Object inflateValue(ObjectInputStream stream) throws IOException, ClassNotFoundException {
        Stock stock=new Stock();
        stock.setTicket(stream.readUTF());
        stock.setCompany(stream.readUTF());
        stock.setDescription(stream.readUTF());
        stock.setPrice(stream.readDouble());
        stock.setLastTransactionTime(stream.readLong());
        stock.setSerialNumber(stream.readInt());
        return stock;
    }

    /* (non-Javadoc)
    * @see com.ibm.websphere.objectgrid.plugins.
    ObjectTransformer#copyValue(java.lang.Object)
    */
    public Object copyValue(Object value) {
        Stock stock = (Stock) value;
        try {
            return stock.clone();
        }
        catch (CloneNotSupportedException e)
        {
            // display exception message
        }
    }

    /* (non-Javadoc)
    * @see com.ibm.websphere.objectgrid.plugins.
    ObjectTransformer#copyKey(java.lang.Object)
    */
    public Object copyKey(Object key) {
        String ticket=(String) key;
        String ticketCopy= new String (ticket);
        return ticketCopy;
    }
}

```

Then, plug in this custom MyStockObjectTransformer class into the BackingMap:

```
ObjectGridManager ogf=ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogf.getObjectGrid("NYSE");
BackingMap bm = og.defineMap("NYSEStocks");
MyStockObjectTransformer ot = new MyStockObjectTransformer();
bm.setObjectTransformer(ot);
```

Plug-ins for providing event listeners

You can use the `ObjectGridEventListener`, `MapEventListener`, `ObjectGridLifecycleListener` and `BackingMapLifecycleListener` plug-ins to configure notifications for various events in the eXtreme Scale cache. Listener plug-ins are registered with an `ObjectGrid` or `BackingMap` instance like other eXtreme Scale plug-ins and add integration and customization points for applications and cache providers.

ObjectGridEventListener plug-in

An `ObjectGridEventListener` plug-in provides eXtreme Scale life cycle events for the `ObjectGrid` instance, shards, and transactions. Use the `ObjectGridEventListener` plug-in to receive notifications when significant events occur on an `ObjectGrid`. These events include `ObjectGrid` initialization, the beginning of a transaction, the ending a transaction, and destroying an `ObjectGrid`. To listen for these events, create a class that implements the `ObjectGridEventListener` interface and add it to the eXtreme Scale.

For more information about writing an `ObjectGridEventListener` plug-in, see “`ObjectGridEventListener` plug-in” on page 298. You can also refer to the API documentation for more information.

Adding and removing ObjectGridEventListener instances

An `ObjectGrid` can have multiple `ObjectGridEventListener` listeners. Add and remove the listeners using the `addEventListener`, and `removeEventListener` methods on the `ObjectGrid` interface. You can also declaratively register `ObjectGridEventListener` plug-ins with the `ObjectGrid` descriptor file. For examples, see “`ObjectGridEventListener` plug-in” on page 298.

MapEventListener plug-in

A `MapEventListener` plug-in provides callback notifications and significant cache state changes that occur for a `BackingMap` instance. For details on writing a `MapEventListener` plug-in, see “`MapEventListener` plug-in” on page 297. You can also refer to the API documentation for more information.

Adding and removing MapEventListener instances

An eXtreme Scale can have multiple `MapEventListener` listeners. Add and remove listeners with the `addMapEventListener`, and `removeMapEventListener` methods on the `BackingMap` interface. You can also declaratively register `MapEventListener` listeners with the `ObjectGrid` descriptor file. For examples, see “`MapEventListener` plug-in” on page 297.

BackingMapLifecycleListener plug-in

A `BackingMapLifecycleListener` plug-in provides callback notifications for life cycle state changes that occur for a `BackingMap` instance. The `BackingMap` instance proceeds through a predefined set of states during its life time.

Adding and removing BackingMapLifecycleListener instances

An eXtreme Scale server can have multiple BackingMapLifecycleListener listeners. Add and remove listeners with the `addMapEventListener` and `removeMapEventListener` methods on the BackingMap interface. Any BackingMap plug-ins that implement the BackingMapLifecycleListener interface are also automatically added as a BackingMapLifecycleListener for the ObjectGrid instance they are registered with. You can also declaratively register BackingMapLifecycleListener listeners with the ObjectGrid descriptor file. For examples, see BackingMapLifecycleListener plug-in.

ObjectGridLifecycleListener plug-in

An ObjectGridLifecycleListener plug-in provides callback notifications for life cycle state changes that occur for an ObjectGrid instance. The ObjectGrid instance proceeds through a predefined set of states during its life time.

Adding and removing ObjectGridLifecycleListener instances

An eXtreme Scale can have multiple ObjectGridLifecycleListener listeners. Add and remove listeners with the `addEventListener` and `removeEventListener` methods on the ObjectGrid interface. Any ObjectGrid plug-ins that implement the ObjectGridLifecycleListener interface are automatically added as an ObjectGridLifecycleListener for the ObjectGrid instance that they are registered with. You can also declaratively register ObjectGridLifecycleListener listeners with the ObjectGrid deployment descriptor file. For examples, see ObjectGridLifecycleListener plug-in.

MapEventListener plug-in

A MapEventListener plug-in provides callback notifications and significant cache state changes that occur for a BackingMap object: when a map has finished pre-loading or when an entry is evicted from the map. A particular MapEventListener plug-in is a custom class you write implementing the MapEventListener interface.

MapEventListener plug-in conventions

When you develop a MapEventListener plug-in, you must follow common plug-in conventions. For more information about plug-in conventions, see “Plug-ins overview” on page 117. For other types of listener plug-ins, see “Plug-ins for providing event listeners” on page 296.

After you write a MapEventListener implementation, you can plug it in to the BackingMap configuration programmatically or with an XML configuration.

Write a MapEventListener implementation

Your application can include an implementation of the MapEventListener plug-in. The plug-in must implement the MapEventListener interface to receive significant events about a map. Events are sent to the MapEventListener plug-in when an entry is evicted from the map and when the preload of a map completes.

Programmatically plug in a MapEventListener implementation

The class name for the custom MapEventListener is the `com.company.org.MyMapEventListener` class. This class implements the

MapEventListener interface. The following code snippet creates the custom MapEventListener object and adds it to a BackingMap object:

```
ObjectGridManager objectGridManager =
    ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
BackingMap myMap = myGrid.defineMap("myMap");
MyMapEventListener myListener = new MyMapEventListener();
myMap.addMapEventListener(myListener);
```

Plug in a MapEventListener implementation using XML

A MapEventListener implementation can be configured using XML. The following XML must be in the myGrid.xml file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridconfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config../objectGrid.xsd"
xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="myGrid">
      <backingMap name="myMap" pluginCollectionRef="myPlugins" />
    </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
    <backingMapPluginCollection id="myPlugins">
      <bean id="MapEventListener" className=
"com.company.org.MyMapEventListener" />
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

Providing this file to the ObjectGridManager instance facilitates the creation of this configuration. The following code snippet shows how to create an ObjectGrid instance using this XML file. The newly created ObjectGrid instance has a MapEventListener set on the myMap BackingMap.

```
ObjectGridManager objectGridManager =
    ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid =
    objectGridManager.createObjectGrid("myGrid", new URL("file:etc/test/myGrid.xml"),
    true, false);
```

ObjectGridEventListener plug-in

An ObjectGridEventListener plug-in provides WebSphere eXtreme Scale life cycle events for the ObjectGrid, shards and transactions. An ObjectGridEventListener plug-in provides notifications when an ObjectGrid is initialized or destroyed, and when a transaction is started or ended. ObjectGridEventListener plug-ins are custom classes you write implementing the ObjectGridEventListener interface. Optionally, the implementation includes ObjectGridEventGroup sub-interfaces and follow the common eXtreme Scale plug-in conventions.

Overview

An ObjectGridEventListener plug-in is useful when a Loader plug-in is available, and you must initialize Java Database Connectivity (JDBC) connections or connections to a back end when transactions start and end. Typically, an ObjectGridEventListener plug-in and a Loader plug-in are written together.

Writing an ObjectGridEventListener plug-in

An ObjectGridEventListener plug-in must implement the ObjectGridEventListener interface to receive notifications about significant eXtreme Scale events. To receive

additional event notifications, you can implement the following interfaces. These sub-interfaces are included in the `ObjectGridEventGroup` interface:

- `ShardEvents` interface
- `ShardLifecycle` interface
- `TransactionEvents` interface

For more information about these interfaces, see the API documentation.

Shard events

When the catalog service places partition primary or replica shards in a Java virtual machine (JVM), a new `ObjectGrid` instance is created in that JVM to host that shard. Some applications that need to start threads on the JVM host the primary need notification of these events. The `ObjectGridEventGroup.ShardEvents` interface declares the `shardActivate` and `shardDeactivate` methods. These methods are called only when a shard is activated as a primary and when the shard is deactivated from a primary. These two events allow the application to start additional threads when the shard is a primary and stop the threads when the shard returns to being a replica or is taken out of service.

An application can determine which partition has been activated by looking up a specific `BackingMap` in the `ObjectGrid` reference that is provided to the `shardActivate` method using the `ObjectGrid#getMap` method. The application can then see the partition number using the `BackingMap#getPartitionId()` method. The partitions are numbered from 0 to the number of partitions in the deployment descriptor minus one.

Shard life-cycle events

`ObjectGridEventListener.initialize` and `ObjectGridEventListener.destroy` method events are delivered using the `ObjectGridEventGroup.ShardLifecycle` interface.

Transaction events

`ObjectGridEventListener.transactionBegin` and `ObjectGridEventListener.transactionEnd` methods are delivered through the `ObjectGridEventGroup.TransactionEvents` interface.

If an `ObjectGridEventListener` plug-in implements the `ObjectGridEventListener` and `ShardLifecycle` interfaces, then shard life-cycle events are the only events that are delivered to the listener. After you implement any of the new `ObjectGridEventGroup` inner interfaces, eXtreme Scale only delivers those specific events by the new interfaces. With this implementation, code can be backwards compatible. If you are using the new inner interfaces, it can now receive just the specific events that are needed.

Using the `ObjectGridEventListener` plug-in

To use a custom `ObjectGridEventListener` plug-in, first create a class that implements the `ObjectGridEventListener` interface and any optional `ObjectGridEventGroup` sub-interfaces. Add the custom listener to an `ObjectGrid` to receive notification of significant events. You have two approaches to add an `ObjectGridEventListener` plug-in into the eXtreme Scale configuration: programmatic configuration and XML configuration.

Configure an ObjectGridEventListener plug-in programmatically

Assume that the class name of the eXtreme Scale event listener is the `com.company.org.MyObjectGridEventListener` class. This class implements the `ObjectGridEventListener` interface. The following code snippet creates the custom `ObjectGridEventListener` and adds it to an `ObjectGrid`.

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
MyObjectGridEventListener myListener = new MyObjectGridEventListener();
myGrid.addEventListener(myListener);
```

Configure an ObjectGridEventListener plug-in with XML

You can also configure an `ObjectGridEventListener` plug-in using XML. The following XML creates a configuration that is equivalent to the described programmatically created `ObjectGrid` event listener. The following text must be in the `myGrid.xml` file:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="myGrid">
      <bean id="ObjectGridEventListener"
        className="com.company.org.MyObjectGridEventListener" />
      <backingMap name="Book"/>
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

Notice the bean declarations come before the `backingMap` declarations. Provide this file to the `ObjectGridManager` plug-in to facilitate the creation of this configuration. The following code snippet demonstrates how to create an `ObjectGrid` instance using this XML file. The `ObjectGrid` instance that is created has an `ObjectGridEventListener` listener set on the `myGrid` `ObjectGrid`.

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid",
  new URL("file:etc/test/myGrid.xml"), true, false);
```

BackingMapLifecycleListener plug-in

A `BackingMapLifecycleListener` plug-in receives notification of WebSphere eXtreme Scale life cycle state change events for the backing map.

The `BackingMapLifecycleListener` plug-in receives an event containing a `BackingMapLifecycleListener.State` object for each state change of the backing map. Any `BackingMap` plug-in that also implements the `BackingMapLifecycleListener` interface will automatically be added as a listener for the `BackingMap` instance where the plug-in is registered.

Overview

A `BackingMapLifecycleListener` plug-in is useful when an existing `BackingMap` plug-in needs to perform activities relative to activities in a related plugin. As an example, a loader plug-in might need to retrieve configuration from a cooperating `MapIndexPlugin` or `DataSerializer` plug-in.

By implementing the `BackingMapLifecycleListener` interface, and detecting the `BackingMapLifecycleListener.State.INITIALIZED` event, the loader can know about the state of other plug-ins in the `BackingMap` instance. The loader can safely retrieve information from the cooperating `MapIndexPlugin` or `DataSerializer`

plug-in, since the BackingMap is in the INITIALIZED state, which means that the other plug-in has had its initialize() method called.

A BackingMapLifecycleListener can be added or removed at any time, either before or after the ObjectGrid and its BackingMaps are initialized.

Write a BackingMapLifecycleListener plug-in

A BackingMapLifecycleListener plug-in must implement the BackingMapLifecycleListener interface to receive notifications about significant eXtreme Scale events. Any BackingMap plug-in can implement the BackingMapLifecycleListener interface and be automatically added as a listener when it is also added to the backing map.

For more information about these interfaces, see the API documentation.

Life cycle event and plug-in relationships

The BackingMapLifecycleListener retrieves the life cycle state from the event in the backingMapStateChanged method; for example:

```
public void backingMapStateChanged(BackingMap map,
                                   LifecycleEvent event)
throws LifecycleFailedException {
    switch(event.getState()) {
        case INITIALIZED: // All other plug-ins are initialized.
            // Retrieve reference to plug-in X for use from map.
            break;
        case DESTROYING: // Destroy phase is starting
            // Eliminate reference to plug-in X it may be destroyed before this plug-in
            break;
    }
}
```

The following table describes the relationship between life cycle events sent to a BackingMapLifecycleListener plug-in and the states of the BackingMap and other plug-in objects.

BackingMapLifecycleListener.State value	Description
INITIALIZING	The BackingMap initialization phase is starting. The BackingMap and BackingMap plug-ins are about to be initialized.
INITIALIZED	The BackingMap initialization phase is complete. All BackingMap plug-ins are initialized. The INITIALIZED state might recur when shard placement activities (promotion or demotion) occur.
STARTING	The BackingMap instance is being activated for use as a local instance, client instance or as an instance in a primary or replica shard on the server. All ObjectGrid plug-ins in the ObjectGrid instance owning this BackingMap instance have been initialized. The STARTING state might recur when shard placement activities (promotion or demotion) occur.
PRELOAD	The BackingMap instance is set to the PRELOAD state by the StateManager API for preloading, or the configured loader is preloading data into the backing map.
ONLINE	The BackingMap instance is ready for work as a local instance, client instance, or as an instance in a primary or replica shard on the server. All ObjectGrid plug-ins in the ObjectGrid instance owning this BackingMap instance have been initialized. This steady state is typical of the BackingMap. The ONLINE state might recur when shard placement activities (promotion or demotion) occur.

BackingMapLifecycleListener.State value	Description
QUIESCE	Work is stopping on the BackingMap as a result of the StateManager API or other event. No new work is allowed. Your plug-in ends any existing work as soon as possible.
OFFLINE	All work is stopped on the BackingMap as a result of the StateManager API or another event. No new work is allowed.
DESTROYING	The BackingMap instance is starting the destroy phase. BackingMap plug-ins for the instance are about to be destroyed.
DESTROYED	The BackingMap instance and all BackingMap plug-ins have been destroyed.

Configure a BackingMapLifecycleListener plug-in with XML

Assume that the class name of the eXtreme Scale event listener is the `com.company.org.MyBackingMapLifecycleListener` class. This class implements the `BackingMapLifecycleListener` interface.

You can configure a `BackingMapLifecycleListener` plug-in using XML. The following text must be in the object grid XML file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<objectGridconfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config../objectGrid.xsd"
xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="myGrid">
      <backingMap name="myMap" pluginCollectionRef="myPlugins" />
    </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
    <backingMapPluginCollection id="myPlugins">
      <bean id="BackingMapLifecycleListener"
        className="com.company.org.MyBackingMapLifecycleListener" />
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

Provide this file to the `ObjectGridManager` plug-in to facilitate the creation of this configuration. The `BackingMap` instance that is created has a `BackingMapLifecycleListener` listener set on the `myGrid` `ObjectGrid`.

Like the `BackingMapLifecycleListener`, other `BackingMap` plug-ins, such as `Loader` or `MapIndexPlugin`, that you specify using XML that also implement the `BackingMapLifecycleListener` interface, will automatically be added as life cycle listeners.

ObjectGridLifecycleListener plug-in

An `ObjectGridLifecycleListener` plug-in receives notification of WebSphere eXtreme Scale life cycle, state change events for the data grid.

The `ObjectGridLifecycleListener` plug-in receives an event containing an `ObjectGridLifecycleListener.State` object for each state change of the `ObjectGrid`. Any `ObjectGrid` plug-in that also implements the `ObjectGridLifecycleListener` interface will automatically be added as a listener for the `ObjectGrid` instance where the plug-in is registered.

Overview

An `ObjectGridLifecycleListener` plug-in is useful when an existing `ObjectGrid` plug-in needs to perform activities relative to activities in a related plug-in. As an example, a `TransactionCallback` plug-in might need to retrieve the configuration from a cooperating `ObjectGridEventListener` or `ShardListener` plug-in.

By implementing the `ObjectGridLifecycleListener` interface, and detecting the `ObjectGridLifecycleListener.State.INITIALIZED` event, the `TransactionCallback` plug-in can detect the state of other plug-ins in the `ObjectGrid` instance. The `TransactionCallback` plug-in can safely retrieve information from the cooperating `ObjectGridEventListener` plug-in or `ShardListener` plug-in, since the `ObjectGrid` is in the `INITIALIZED` state, which means that the other plug-in has had its `initialize()` method called.

You can add an `ObjectGridLifecycleListener` plug-in at any time, either before or after the `ObjectGrid` is initialized.

Write an `ObjectGridLifecycleListener` plug-in

An `ObjectGridLifecycleListener` plug-in must implement the `ObjectGridLifecycleListener` interface to receive notifications about significant eXtreme Scale events. Any `ObjectGrid` plug-in can implement the `ObjectGridLifecycleListener` interface and be automatically added as a listener when it is also added to the `ObjectGrid`.

For more information about these interfaces, see the API documentation.

Life cycle event and plug-in relationships

The `ObjectGridLifecycleListener` retrieves the life cycle state from the event in the `objectGridStateChanged` method; for example:

```
public void objectGridStateChanged(ObjectGrid grid,
                                  LifecycleEvent event)
    throws LifecycleFailedException {
    switch(event.getState()) {
        case INITIALIZED: // All other plug-ins are initialized.
                          // Retrieve reference to plug-in X for use from grid.
                          break;
        case DESTROYING: // Destroy phase is starting
                          // Eliminate reference to plug-in X it may be destroyed before this plug-in
                          break;
    }
}
```

The following table describes the relationship between life cycle events sent to a `ObjectGridLifecycleListener` and the states of the `ObjectGrid` and other plug-in objects.

<code>ObjectGridLifecycleListener.State</code> value	Description
<code>INITIALIZING</code>	The <code>ObjectGrid</code> initialization phase is starting. The <code>ObjectGrid</code> and <code>ObjectGrid</code> plug-ins are about to be initialized.
<code>INITIALIZED</code>	The <code>ObjectGrid</code> initialization phase is complete. All <code>ObjectGrid</code> plug-ins are initialized. The <code>INITIALIZED</code> state might recur when shard placement activities (promotion or demotion) occur. All <code>BackingMap</code> plug-ins in the <code>BackingMap</code> instances owned by this <code>ObjectGrid</code> instance have been initialized.

ObjectGridLifecycleListener.State value	Description
STARTING	The ObjectGrid instance is being activated for use as a local instance, client instance, or as an instance in a primary or replica shard on the server. The STARTING state might recur when shard placement activities (promotion or demotion) occur.
PRELOAD	The ObjectGrid instance is set to the PRELOAD state by the StateManager API or other configuration.
ONLINE	The ObjectGrid instance is ready for work as a local instance, client instance, or as an instance in a primary or replica shard on the server. This steady state is typical of the ObjectGrid. The ONLINE state might recur when shard placement activities (promotion or demotion) occur.
QUIESCE	Work is stopping on the ObjectGrid as a result of the StateManager API or other event. No new work is allowed. End any existing work as soon as possible.
OFFLINE	All work is stopped on the ObjectGrid as a result of the StateManager API or other event. No new work is allowed.
DESTROYING	The ObjectGrid instance is starting the destroy phase. ObjectGrid plug-ins for the instance are about to be destroyed. During the destroy phase, all BackingMap instances owned by this ObjectGrid instance are also destroyed.
DESTROYED	The ObjectGrid instance, its BackingMap instances, and all ObjectGrid plug-ins have been destroyed.

Configure an ObjectGridLifecycleListener plug-in with XML

Assume that the class name of the eXtreme Scale event listener is the `com.company.org.MyObjectGridLifecycleListener` class. This class implements the `ObjectGridLifecycleListener` interface.

You can configure an `ObjectGridLifecycleListener` plug-in using XML. The following XML creates a configuration using the `ObjectGridLifecycleListener`. The following text must be in the object grid xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="myGrid">
      <bean id="ObjectGridLifecycleListener"
        className="com.company.org.MyObjectGridLifecycleListener" />
      <backingMap name="Book"/>
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

Notice the bean declarations come before the backingMap declarations. Provide this file to the `ObjectGridManager` plug-in to facilitate the creation of this configuration.

Like the registered `ObjectGridLifecycleListener` in the previous example, other `ObjectGrid` plug-ins, `CollisionArbiter` or `TransactionCallback` for example, that you specify using XML that also implement the `ObjectGridLifecycleListener` interface, will automatically be added as life cycle listeners.

Plug-ins for indexing data

The built-in `HashIndex`, the `com.ibm.websphere.objectgrid.plugins.index.HashIndex` class, is a `MapIndexPlugin`

plug-in that you can add into BackingMap to build static or dynamic indexes. This class supports both the MapIndex and MapRangeIndex interfaces. Defining and implementing indexes can significantly improve query performance.

Configuring the HashIndex plug-in

You can configure the built-in HashIndex, the `com.ibm.websphere.objectgrid.plugins.index.HashIndex` class, with an XML file, programmatically, or with an entity annotation on an entity map.

About this task

Configuring a composite index is the same as configuring a regular index with XML, except for the **attributeName** property value. In a composite index, the value of **attributeName** property is a comma-delimited list of attributes. For example, the value class Address has three attributes: city, state, and zipcode. A composite index can be defined with the **attributeName** property value as "city,state,zipcode" indicating that city, state, and zipcode are included in the composite index.

Also, note that the composite HashIndexes do not support range lookups and therefore cannot have the RangeIndex property set to true.

Procedure

- Configure a composite index in the ObjectGrid descriptor XML file.

Use the backingMapPluginCollections element to define the plug-in:

```
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Address.CityStateZip"/>
  <property name="AttributeName" type="java.lang.String" value="city,state,zipcode"/>
</bean>
```

- Configure a composite index programmatically.

The following example code creates the same composite index:

```
HashIndex mapIndex = new HashIndex();
mapIndex.setName("Address.CityStateZip");
mapIndex.setAttributeName("city,state,zipcode");
mapIndex.setRangeIndex(true);

BackingMap bm = objectGrid.defineMap("mymap");
bm.addMapIndexPlugin(mapIndex);
```

- Configure a composite index with entity notations.

If you are using entity maps, you can use an annotation approach to define a composite index. You can define a list of CompositeIndex within the CompositeIndexes annotation on the entity class level. The CompositeIndex has a name and **attributeNames** property. Each CompositeIndex is associated with a HashIndex instance applied to the backing map that is associated with the entity. The HashIndex is configured as a non-range index.

```
@Entity
@CompositeIndexes({
    @CompositeIndex(name="CityStateZip", attributeNames="city,state,zipcode"),
    @CompositeIndex(name="lastnameBirthday", attributeNames="lastname,birthday")
})
public class Address {
    @Id int id;
    String street;
    String city;
    String state;
    String zipcode;
    String lastname;
    Date birthday;
}
```

The name property for each composite index must be unique within the entity and backing map. If the name is not specified, a generated name is used. The **attributeName** property is used to populate the HashIndex attributeName with the comma-delimited list of attributes. The attribute names coincide with the persistent field names when the entities are configured to use field-access, or the

property name as defined for the JavaBeans naming conventions for property-access entities. For example: If the attribute name is street, the property getter method is named getStreet.

Example: Adding HashIndex into BackingMap

In the following example, you configure the HashIndex plug-in by adding static index plug-ins to the XML file:

```
<backingMapPluginCollection id="person">
  <bean id="MapIndexPlugin"
    className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
    <property name="Name" type="java.lang.String" value="CODE"
      description="index name" />
    <property name="RangeIndex" type="boolean" value="true"
      description="true for MapRangeIndex" />
    <property name="AttributeName" type="java.lang.String" value="employeeCode"
      description="attribute name" />
  </bean>
</backingMapPluginCollection>
```

In this XML configuration example, the built-in HashIndex class is used as the index plug-in. The HashIndex supports properties that users can configure, such as Name, RangeIndex, and AttributeName.

- The **Name** property is configured as CODE, a string identifying this index plug-in. The **Name** property value must be unique within the scope of the backing map. The name can be used to retrieve the index object by name from the ObjectMap instance for the BackingMap.
- The **RangeIndex** property is configured as true, which means the application can cast the retrieved index object to the MapRangeIndex interface. If the RangeIndex property is configured as false, the application can only cast the retrieved index object to the MapIndex interface. A MapRangeIndex supports functions to find data using range functions such as greater than, less than, or both, while a MapIndex only supports equals functions. If the index is by query, the **RangeIndex** property must be configured to true on single-attribute indexes. For a relationship index and composite index, the **RangeIndex** property must be configured to false.
- The **AttributeName** property is configured as employeeCode, which means the employeeCode attribute of the cached object is used to build a single-attribute index. If an application must search for cached objects with multiple attributes, the **AttributeName** property can be set to a comma-delimited list of attributes, yielding a composite index.

In summary, the previous example defines a single-attribute range HashIndex. It is a single-attribute HashIndex because the **AttributeName** property value is employeeCode that includes only one attribute name. It also is a range HashIndex.

HashIndex plug-in attributes:

You can use the following attributes to configure the HashIndex plug-in. These attributes define properties such as if you are using an attribute or composite HashIndex, or if range indexing is enabled.

Attributes

Name Specifies the name of the index. The name must be unique for each map. The name is used to retrieve the index object from the object map instance for the backing map.

AttributeName

Specifies the comma-delimited names of the attributes to index. For field-access indexes, the attribute names are equivalent to the field names. For property-access indexes, the attribute names are the JavaBean compatible property names. If only one attribute name exists, the HashIndex is a single attribute index. If this attribute is a relationship, it is also a relationship index. If multiple attribute names are included in the attribute names, the HashIndex is a composite index.

FieldAccessAttribute

Used for non-entity maps. If true, the object is accessed using the fields directly. If not specified or false, the getter method for the attribute is used to access the data.

POJOKeyIndex

Used for non-entity maps. If true, the index introspects the object in the key part of the map. This setting is useful when the key is a composite key and the value does not have the key embedded within it. If not specified or false, then the index introspects the object in the value part of the map.

RangeIndex

If true, range indexing is enabled and the application can cast the retrieved index object to the MapRangeIndex interface. If the **RangeIndex** property is configured as false, the application can cast the retrieved index object to the MapIndex interface only.

Single-attribute HashIndex versus composite HashIndex

When the **AttributeName** property of HashIndex includes multiple attribute names, the HashIndex is a composite index. Otherwise, if it includes only one attribute name, it is a single-attribute index. For example, the AttributeName property value of a composite HashIndex might be city,state,zipcode. It includes three attributes delimited by commas. If the **AttributeName** property value is only zipcode that only has one attribute, it is a single-attribute HashIndex.

Composite HashIndex provides an efficient way to look up cached objects when search criteria involve many attributes. However, it does not support range index and its RangeIndex property must set to false.

See the topic on a composite HashIndex in the *Administration Guide*.

Relationship HashIndex

If the indexed attribute of single-attribute HashIndex is a relationship, either single- or multi-valued, the HashIndex is a relationship HashIndex. For relationship HashIndex, the RangeIndex property of HashIndex must set to "false".

Relationship HashIndex can speed up queries that use cyclical references or use the IS NULL, IS EMPTY, SIZE, and MEMBER OF query filters. For more information, see "Query optimization using indexes" on page 413 the information about query optimization with indexes in the *Programming Guide*.

Key HashIndex

For non-entity maps, when the **POJOKeyIndex** property of HashIndex is set to true, the HashIndex is a key HashIndex and the key part of entry are used for indexing.

When the `AttributeName` property of `HashIndex` is not specified, the whole key is indexed; otherwise, the key `HashIndex` can only be a single-attribute `HashIndex`.

For example, adding the following property into the preceding sample causes the `HashIndex` to become key `HashIndex` because the `POJOKeyIndex` property value is true.

```
<property name="POJOKeyIndex" type="boolean" value="true"
description="indicates if POJO key HashIndex" />
```

In the preceding key index example, because the **AttributeName** property value is specified as `employeeCode`, the indexed attribute is the **employeeCode** field of the key part of map entry. If you want to build key index on the whole key part of map entry, remove the **AttributeName** property.

Range HashIndex

When the `RangeIndex` property of `HashIndex` is set to true, the `HashIndex` is a range index and can support the `MapRangeIndex` interface. A `MapRangeIndex` implementation supports functions to find data using range functions, such as greater than, less than, or both, while a `MapIndex` supports equals functions only. For a single-attribute index, the **RangeIndex** property can be set to true only if the indexed attribute is of type `Comparable`. If the single-attribute index will be used by query, the `RangeIndex` property must set to true and the indexed attribute must be of type `Comparable`. For relationship `HashIndex` and composite `HashIndex`, the `RangeIndex` property must set to false.

The preceding sample is a range `HashIndex` because the `RangeIndex` property value is true.

The following table provides a summary for using range index.

Table 6. Support for range index. States whether `HashIndex` types support range index.

HashIndex type	Supports range index
Single-attribute <code>HashIndex</code> : indexed key or attribute is of type <code>Comparable</code>	Yes
Single-attribute <code>HashIndex</code> : indexed key or attribute is not of type <code>Comparable</code>	No
Composite <code>HashIndex</code>	No
Relationship <code>HashIndex</code>	No

Query optimization with `HashIndex` plug-ins

Defining indexes can significantly improve query performance. `WebSphere eXtreme Scale` queries can use built-in `HashIndex` plug-ins to improve performance of queries. Although using indexes can significantly improve query performance, it might have a performance impact on transactional map operations.

Plug-ins for custom indexing of cache objects:

With a `MapIndexPlugin` plug-in, or index, you can write custom indexing strategies that are beyond the built-in indexes that `eXtreme Scale` provides.

`MapIndexPlugin` implementations must use the `MapIndexPlugin` interface and follow the common `eXtreme Scale` plug-in conventions.

The following sections include some of the important methods of the index interface.

setProperties method

Use the setProperties method to initialize the index plug-in programmatically. The Properties object parameter that is passed into the method should contain required configuration information to initialize the index plug-in properly. The setProperties method implementation, along with the getProperties method implementation, are required in a distributed environment because the index plug-in configuration moves between client and server processes. An implementation example of this method follows.

```
setProperties(Properties properties)

// setProperties method sample code
public void setProperties(Properties properties) {
    ivIndexProperties = properties;

    String ivRangeIndexString = properties.getProperty("rangeIndex");
    if (ivRangeIndexString != null && ivRangeIndexString.equals("true")) {
        setRangeIndex(true);
    }
    setName(properties.getProperty("indexName"));
    setAttributeName(properties.getProperty("attributeName"));

    String ivFieldAccessAttributeString = properties.getProperty("fieldAccessAttribute");
    if (ivFieldAccessAttributeString != null && ivFieldAccessAttributeString.equals("true")) {
        setFieldAccessAttribute(true);
    }

    String ivPOJOKeyIndexString = properties.getProperty("POJOKeyIndex");
    if (ivPOJOKeyIndexString != null && ivPOJOKeyIndexString.equals("true")) {
        setPOJOKeyIndex(true);
    }
}
```

getProperties method

The getProperties method extracts the index plug-in configuration from a MapIndexPlugin instance. You can use the extracted properties to initialize another MapIndexPlugin instance to have the same internal states. The getProperties method and setProperties method implementations are required in a distributed environment. An implementation example of the getProperties method follows.

```
getProperties()

// getProperties method sample code
public Properties getProperties() {
    Properties p = new Properties();
    p.put("indexName", indexName);
    p.put("attributeName", attributeName);
    p.put("rangeIndex", ivRangeIndex ? "true" : "false");
    p.put("fieldAccessAttribute", ivFieldAccessAttribute ? "true" : "false");
    p.put("POJOKeyIndex", ivPOJOKeyIndex ? "true" : "false");
    return p;
}
```

setEntityMetadata method

The setEntityMetadata method is called by the WebSphere eXtreme Scale run time during initialization to set the EntityMetadata of the associated BackingMap on the MapIndexPlugin instance. The EntityMetadata is required for supporting indexing of tuple objects. A tuple is a data set that represents an entity object or its key. If the BackingMap is for an entity, then you must implement this method.

The following code sample implements the setEntityMetadata method.

```

setEntityMetadata(EntityMetadata entityMetadata)

// setEntityMetadata method sample code
public void setEntityMetadata(EntityMetadata entityMetadata) {
    ivEntityMetadata = entityMetadata;
    if (ivEntityMetadata != null) {
        // this is a tuple map
        TupleMetadata valueMetadata = ivEntityMetadata.getValueMetadata();
        int numAttributes = valueMetadata.getNumAttributes();
        for (int i = 0; i < numAttributes; i++) {
            String tupleAttributeName = valueMetadata.getAttribute(i).getName();
            if (attributeName.equals(tupleAttributeName)) {
                ivTupleValueIndex = i;
                break;
            }
        }

        if (ivTupleValueIndex == -1) {
            // did not find the attribute in value tuple, try to find it on key tuple.
            // if found on key tuple, implies key indexing on one of tuple key attributes.
            TupleMetadata keyMetadata = ivEntityMetadata.getKeyMetadata();
            numAttributes = keyMetadata.getNumAttributes();
            for (int i = 0; i < numAttributes; i++) {
                String tupleAttributeName = keyMetadata.getAttribute(i).getName();
                if (attributeName.equals(tupleAttributeName)) {
                    ivTupleValueIndex = i;
                    ivKeyTupleAttributeIndex = true;
                    break;
                }
            }
        }

        if (ivTupleValueIndex == -1) {
            // if entityMetadata is not null and we could not find the
            // attributeName in entityMetadata, this is an
            // error
            throw new ObjectGridRuntimeException("Invalid attributeName. Entity: " +
                ivEntityMetadata.getName());
        }
    }
}
}

```

Attribute name methods

The `setAttributeName` method sets the name of the attribute to be indexed. The cached object class must provide the get method for the indexed attribute. For example, if the object has an `employeeName` or `EmployeeName` attribute, the index calls the `getEmployeeName` method on the object to extract the attribute value. The attribute name must be the same as the name in the get method, and the attribute must implement the `Comparable` interface. If the attribute is boolean type, you can also use the `isAttributeName` method pattern.

The `getAttributeName` method returns the name of the indexed attribute.

getAttribute method

The `getAttribute` method returns the indexed attribute value from the specified object. For example, if an `Employee` object has an attribute called `employeeName` that is indexed, you can use the `getAttribute` method to extract the `employeeName` attribute value from a specified `Employee` object. This method is required in a distributed WebSphere eXtreme Scale environment.

```

getAttribute(Object value)

// getAttribute method sample code
public Object getAttribute(Object value) throws ObjectGridRuntimeException {
    if (ivPOJOKeyIndex) {
        // In the POJO key indexing case, no need to get attribute from value object.
        // The key itself is the attribute value used to build the index.
        return null;
    }

    try {
        Object attribute = null;
        if (value != null) {
            // handle Tuple value if ivTupleValueIndex != -1
            if (ivTupleValueIndex == -1) {
                // regular value
                if (ivFieldAccessAttribute) {

```



```

        attribute = this.getAttributeField(value).get(value);
    } else {
        attribute = getAttributeMethod(value).invoke(value, emptyArray);
    }
} else {
    // Tuple value
    attribute = extractValueFromTuple(value);
}
}
return attribute;
} catch (InvocationTargetException e) {
    throw new ObjectGridRuntimeException(
        "Caught unexpected Throwable during index update processing,
        index name = " + indexName + ": " + t,
        t);
} catch (Throwable t) {
    throw new ObjectGridRuntimeException(
        "Caught unexpected Throwable during index update processing,
        index name = " + indexName + ": " + t,
        t);
}
}
}

```

Using a composite index:

The composite HashIndex improves query performance and avoids expensive map scanning. The feature also provides a convenient way for the HashIndex API to find cached objects when search criteria involve many attributes.

Improved performance

A composite HashIndex provides a fast and convenient way to search for cached objects with multiple attributes in match-searching criteria. The composite index supports full attribute-match searches, but does not support range searches.

Note: Composite indexes do not support the BETWEEN operator in the ObjectGrid query language because BETWEEN would require range support. The greater than (>) and less than (<) conditionals also do not work because they require range indexes.

A composite index can improve performance of queries if the appropriate composite index is available for the WHERE condition. This means that the composite index has exactly the same attributes as involved in the WHERE condition with full attributes matched.

A query might have many attributes involved in a condition as in the following example.

```
SELECT a FROM Address a WHERE a.city='Rochester' AND a.state='MN' AND
a.zipcode='55901'
```

Composite index can improve query performance by avoiding scanning map or joining multiple single-attribute index results. In the example, if a composite index is defined with attributes (city,state,zipcode), the query engine can use the composite index to find the entry with city='Rochester', state='MN', and zipcode='55901'. Without composite index and attribute index on city, state, and zipcode attributes, the query engine will have to scan the map or join multiple single-attribute searches, which usually have expensive overhead. Also, querying for the composite index only supports a full-matched pattern.

Configuring a composite index

You can configure composite indexing in three ways: using XML, programmatically, and with entity annotations only for entity maps.

Programmatic configuration

The programmatic example code below will create the same composite index as the preceding XML.

```
HashIndex mapIndex = new HashIndex();
mapIndex.setName("Address.CityStateZip");
mapIndex.setAttributeName(("city,state,zipcode"));
mapIndex.setRangeIndex(true);

BackingMap bm = objectGrid.defineMap("mymap");
bm.addMapIndexPlugin(mapIndex);
```

Note that configuring a composite index is the same as configuring a regular index with XML except for the `attributeName` property value. In a composite index case, the value of `attributeName` is a comma-delimited list of attributes. For example, the value class `Address` has 3 attributes: `city`, `state`, and `zipcode`. A composite index can be defined with the `attributeName` property value as `"city,state,zipcode"` indicating that `city`, `state`, and `zipcode` are included in the composite index.

Also, note that the composite `HashIndexes` do not support range lookups and therefore cannot have the `RangeIndex` property set to `true`.

Using XML

In order to configure a composite index with XML, include code such as below in the configuration file's `backingMapPluginCollections` element.

```
Composite index - XML configuration approach
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
  <property name="Name" type="java.lang.String" value="Address.CityStateZip"/>
  <property name="AttributeName" type="java.lang.String" value="city,state,zipcode"/>
</bean>
```

With entity annotations

In the entity map case, annotation approach can be used to define a composite index. You can define a list of `CompositeIndex` within `CompositeIndexes` annotation on the entity class level. The `CompositeIndex` has a `name` and `attributeNames` property. Each `CompositeIndex` is associated with a `HashIndex` instance applied to the entity's associated `BackingMap`. The `HashIndex` is configured as a non-range index.

```
@Entity
@CompositeIndexes({
    @CompositeIndex(name="CityStateZip", attributeNames="city,state,zipcode"),
    @CompositeIndex(name="lastnameBirthday", attributeNames="lastname,birthday")
})
public class Address {
    @Id int id;
    String street;
    String city;
    String state;
    String zipcode;
    String lastname;
    Date birthday;
}
```

The `name` property for each composite index must be unique within the entity and `BackingMap`. If the `name` is not specified, a generated name will be used. The `attributeNames` property is used to populate the `HashIndex` `attributeName` with the comma-delimited list of attributes. The attribute names coincide with the persistent field names when the entities are configured to use field-access, or the property name as defined for the JavaBeans naming conventions for property-access entities. For example: If the attribute name is `"street"`, the property getter method is named `getStreet`.

Performing composite index lookups

After a composite index is configured, an application can use the `findAll(Object)` method of the `MapIndex` interface to perform lookups, as below.

```
Session sess = objectgrid.getSession();
ObjectMap map = sess.getMap("MAP_NAME");
MapIndex codeIndex = (MapIndex) map.getIndex("INDEX_NAME");
Object[] compositeValue = new Object[]{ MapIndex.EMPTY_VALUE,
    "MN", "55901"};
Iterator iter = mapIndex.findAll(compositeValue);
// Close the session (optional in Version 7.1.1 and later) for improved performance
sess.close();
```

The `MapIndex.EMPTY_VALUE` is assigned to the `compositeValue[0]` which indicates that the city attribute is excluded from evaluation. Only objects with state attribute equal to "MN" and zipcode attribute equal to "55901" will be included in the result.

The following queries benefit from the previous composite index configuration:

```
SELECT a FROM Address a WHERE a.city='Rochester' AND a.state='MN' AND
a.zipcode='55901'
```

```
SELECT a FROM Address a WHERE a.state='MN' AND a.zipcode='55901'
```

The query engine will find the appropriate composite index and use it to improve query performance in full attribute-match cases.

In some scenarios, the application might need to define multiple composite indexes with overlapped attributes in order to satisfy all queries with full attributes matched. A disadvantage of increasing the number of indexes is the possible performance overhead on map operations.

Migration and interoperability

The only constraint for the use of a composite index is that an application cannot configure it in a distributed environment with heterogeneous containers. Old and new containers cannot be mixed, since older containers will not recognize a composite index configuration. The composite index is just like the existing regular attribute index, except that the former allows indexing over multiple attributes. When using only the regular attribute index, a mixed-container environment is still viable.

Plug-ins for communicating with databases

With a Loader plug-in, an `ObjectGrid` map can behave as a memory cache for data that is typically kept in a persistent store on either the same system or some other system. Typically, a database or file system is used as the persistent store. A remote Java virtual machine (JVM) can also be used as the source of data, allowing hub-based caches to be built using `ObjectGrid`. A loader has the logic for reading and writing data to and from a persistent store.

Loaders are backing map plug-ins that are invoked when changes are made to the backing map or when the backing map is unable to satisfy a data request (a cache miss).

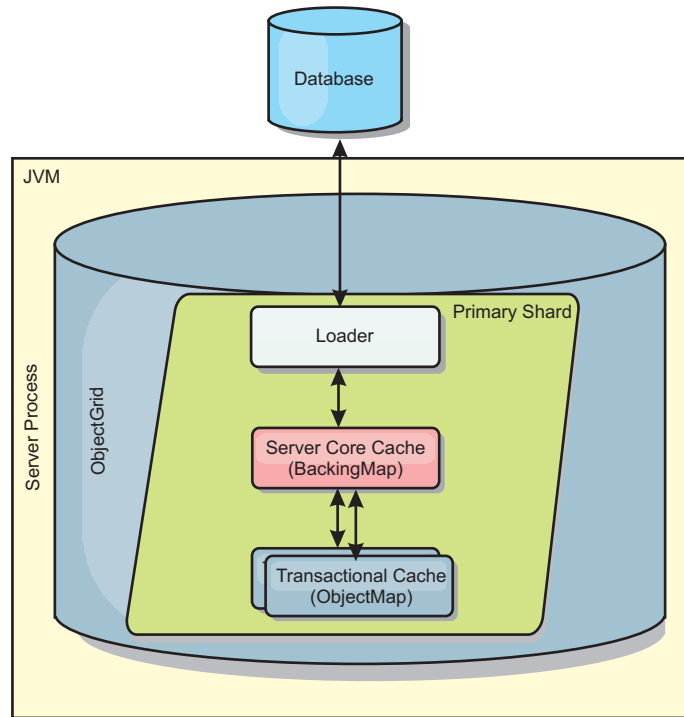


Figure 27. Loader

WebSphere eXtreme Scale includes two built-in loaders to integrate with relational database back ends. The Java Persistence API (JPA) loaders use the Object-Relational Mapping (ORM) capabilities of both the OpenJPA and Hibernate implementations of the JPA specification.

Using a loader

To add a loader into the BackingMap configuration, you can use programmatic configuration or XML configuration. A loader has the following relationship with a backing map:

- A backing map can have only one loader.
- A client backing map (near cache) cannot have a loader.
- A loader definition can be applied to multiple backing maps, but each backing map has its own loader instance.

Loaders in multi-master configurations

For considerations about using loaders in multi-master configurations, see “Loader considerations in a multi-master topology” on page 109.

Programmatically plug in a Loader

The following snippet of code demonstrates how to plug an application-provided Loader into the backing map for map1 using the ObjectGrid API:

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid" );
BackingMap bm = og.defineMap( "map1" );
```

```

MyLoader loader = new MyLoader();
loader.setDataBaseName("testdb");
loader.setIsolationLevel("read committed");
bm.setLoader( loader );

```

This snippet assumes that the MyLoader class is the application-provided class that implements the com.ibm.websphere.objectgrid.plugins.Loader interface. Because the association of a Loader with a backing map cannot be changed after ObjectGrid is initialized, the code must be run before invoking the initialize method of the ObjectGrid interface that is being called. An IllegalStateException exception occurs on a setLoader method call if it is called after initialization has occurred.

The application-provided Loader can have set properties. In the example, the MyLoader loader is used to read and write data from a table in a relational database. The loader must specify the name of the database and the SQL isolation level. The MyLoader loader has the setDataBaseName and setIsolationLevel methods that allow the application to set these two Loader properties.

XML configuration approach to plug in a Loader

An application-provided Loader can also be plugged in by using an XML file. The following example demonstrates how to plug the MyLoader loader into the map1 backing map with the same database name and isolation level Loader properties. You must specify the className for your loader, the database name and connection details, and the isolation level properties. You can use the same XML structure if you are only using a preloader by specifying the preloader classname instead of a complete loader classname.:

```

<?xml version="1.0" encoding="UTF-8" ?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
  <objectGrid name="grid">
    <backingMap name="map1" pluginCollectionRef="map1" lockStrategy="OPTIMISTIC" />
  </objectGrid>
</objectGrids>
<backingMapPluginCollections>
  <backingMapPluginCollection id="map1">
    <bean id="Loader" className="com.myapplication.MyLoader">
      <property name="dataBaseName"
        type="java.lang.String"
        value="testdb"
        description="database name" />
      <property name="isolationLevel"
        type="java.lang.String"
        value="read committed"
        description="iso level" />
    </bean>
  </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>

```

Configuring database loaders

Loaders are backing map plug-ins that are invoked when changes are made to the backing map or when the backing map is unable to satisfy a data request (a cache miss).

Preload considerations

Loaders are backing map plug-ins that are invoked when changes are made to the backing map or when the backing map is unable to satisfy a data request (a cache miss). For an overview of how eXtreme Scale interacts with a loader, see “In-line cache” on page 93.

Each backing map has a boolean `preloadMode` attribute that is set to indicate if preload of a map runs asynchronously. By default, the `preloadMode` attribute is set to `false`, which indicates that the backing map initialization does not complete until the preload of the map is complete. For example, backing map initialization is not complete until the `preloadMap` method returns. If the `preloadMap` method reads a large amount of data from its back end and loads it into the map, it might take a relatively long time to complete. In this case, you can configure a backing map to use asynchronous preload of the map by setting the `preloadMode` attribute to `true`. This setting causes the backing map initialization code to start a thread that invokes the `preloadMap` method, allowing initialization of a backing map to complete while the preload of the map is still in progress.

In a distributed eXtreme Scale scenario, one of the preload patterns is client preload. In the client preload pattern, an eXtreme Scale client is responsible for retrieving data from the backend and then inserting the data into the distributed container server using DataGrid agents. Furthermore, client preload could be executed in the `Loader.preloadMap` method in one and only one specific partition. In this case, asynchronously loading the data to the grid becomes very important. If the client preload were executed in the same thread, the backing map would never be initialized, so the partition it resides in would never become ONLINE. Therefore, the eXtreme Scale client could not send the request to the partition, and eventually it would cause an exception.

If an eXtreme Scale client is used in the `preloadMap` method, you should set the **`preloadMode`** attribute to `true`. The alternative is to start a thread in the client preload code.

The following snippet of code illustrates how the `preloadMode` attribute is set to enable asynchronous preload:

```
BackingMap bm = og.defineMap( "map1" );  
bm.setPreloadMode( true );
```

The `preloadMode` attribute can also be set by using a XML file as illustrated in the following example:

```
<backingMap name="map1" preloadMode="true" pluginCollectionRef="map1"  
  lockStrategy="OPTIMISTIC" />
```

TxID and use of the TransactionCallback interface

Both the `get` method and `batchUpdate` methods on the `Loader` interface are passed a `TxID` object that represents the Session transaction that requires the `get` or `batchUpdate` operation to be performed. The `get` and `batchUpdate` methods can be called more than once per transaction. Therefore, transaction-scoped objects that are needed by the `Loader` are typically kept in a slot of the `TxID` object. A Java database connectivity (JDBC) `Loader` is used to illustrate how a `Loader` uses the `TxID` and `TransactionCallback` interfaces.

Several `ObjectGrid` maps can be stored in the same database. Each map has its own loader, and each loader might need to connect to the same database. When the loaders connect to the database, they should use the same JDBC connection. Using the same connection commits the changes to each table as part of the same database transaction. Typically, the same person who writes the `Loader` implementation also writes the `TransactionCallback` implementation. The best method is when the `TransactionCallback` interface is extended to add methods that the `Loader` needs for getting a database connection and for caching prepared

statements. The reason for this methodology becomes apparent as you see how the TransactionCallback and TxID interfaces are used by the loader.

As an example, the loader might need the TransactionCallback interface to be extended as follows:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.TxID;
public interface MyTransactionCallback extends TransactionCallback
{
    Connection getAutoCommitConnection(TxID tx, String databaseName) throws SQLException;
    Connection getConnection(TxID tx, String databaseName, int isolationLevel ) throws SQLException;
    PreparedStatement getPreparedStatement(TxID tx, Connection conn, String tableName, String sql)
    throws SQLException;
    Collection getPreparedStatementCollection( TxID tx, Connection conn, String tableName );
}
```

Using these new methods, the Loader get and batchUpdate methods can get a connection as follows:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.TxID;
private Connection getConnection(TxID tx, int isolationLevel)
{
    Connection conn = ivTcb.getConnection(tx, databaseName, isolationLevel );
    return conn;
}
```

In the previous example and in the examples that follow, ivTcb and ivOcb are Loader instance variables that were initialized as described in the Preload considerations section. The ivTcb variable is a reference to the MyTransactionCallback instance and the ivOcb is a reference to the MyOptimisticCallback instance. The databaseName variable is an instance variable of the Loader that was set as a Loader property during the initialization of the backing map. The isolationLevel argument is one of the JDBC Connection constants that are defined for the various isolation levels that JDBC supports. If the Loader is using an optimistic implementation, the get method typically uses a JDBC auto-commit connection to fetch the data from the database. In that case, the Loader might have a getAutoCommitConnection method that is implemented as follows:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.TxID;
private Connection getAutoCommitConnection(TxID tx)
{
    Connection conn = ivTcb.getAutoCommitConnection(tx, databaseName);
    return conn;
}
```

Recall that the batchUpdate method has the following switch statement:

```
switch ( logElement.getType().getCode() )
{
    case LogElement.CODE_INSERT:
        buildBatchSQLInsert( tx, key, value, conn );
        break;
    case LogElement.CODE_UPDATE:
        buildBatchSQLUpdate( tx, key, value, conn );
        break;
}
```

```

        case LogElement.CODE_DELETE:
            buildBatchSQLDelete( tx, key, conn );
            break;
    }

```

Each of the buildBatchSQL methods uses the MyTransactionCallback interface to get a prepared statement. Following is a snippet of code that shows the buildBatchSQLUpdate method building an SQL update statement for updating an EmployeeRecord entry and adding it for the batch update:

```

private void buildBatchSQLUpdate( TxID tx, Object key, Object value,
    Connection conn )
    throws SQLException, LoaderException
{
    String sql = "update EMPLOYEE set LASTNAME = ?, FIRSTNAME = ?, DEPTNO = ?,
        SEQNO = ?, MGRNO = ? where EMPNO = ?";
    PreparedStatement sqlUpdate = ivTcb.getPreparedStatement( tx, conn,
        "employee", sql );
    EmployeeRecord emp = (EmployeeRecord) value;
    sqlUpdate.setString(1, emp.getLastName());
    sqlUpdate.setString(2, emp.getFirstName());
    sqlUpdate.setString(3, emp.getDepartmentName());
    sqlUpdate.setLong(4, emp.getSequenceNumber());
    sqlUpdate.setInt(5, emp.getManagerNumber());
    sqlUpdate.setInt(6, key);
    sqlUpdate.addBatch();
}

```

After the batchUpdate loop has built all of the prepared statements, it calls the getPreparedStatementCollection method. This method is implemented as follows:

```

private Collection getPreparedStatementCollection( TxID tx, Connection conn )
{
    return ( ivTcb.getPreparedStatementCollection( tx, conn, "employee" ) );
}

```

When the application invokes the commit method on the Session, the Session code calls the commit method on the TransactionCallback method after it has pushed all the changes made by the transaction out to the Loader for each map that was changed by the transaction. Because all of the Loaders used the MyTransactionCallback method to get any connection and prepared statements they needed, the TransactionCallback method knows which connection to use to request that the back end commits the changes. So, extending the TransactionCallback interface with methods that are needed by each of the Loaders has the following advantages:

- The TransactionCallback object encapsulates the use of TxID slots for transaction-scoped data, and the Loader does not require information about the TxID slots. The Loader only needs to know about the methods that are added to TransactionCallback using the MyTransactionCallback interface for the supporting functions needed by the Loader.
- The TransactionCallback object can ensure that connection sharing occurs between each Loader that connects to the same backend so that a two phase commit protocol can be avoided.
- The TransactionCallback object can ensure that connecting to the backend is driven to completion through a commit or rollback invoked on the connection when appropriate.
- TransactionCallback ensures that the cleanup of database resources occurs when a transaction completes.
- TransactionCallback hides if it is obtaining a managed connection from a managed environment such as WebSphere Application Server or some other Java

2 Platform, Enterprise Edition (J2EE) compliant application server. This advantage allows the same Loader code to be used in both a managed and unmanaged environments. Only the TransactionCallback plug-in must be changed.

- For detailed information about how the TransactionCallback implementation uses the TxID slots for transaction-scoped data, see TransactionCallback plug-in.

OptimisticCallback

As mentioned earlier, the Loader might use an optimistic approach for concurrency control. In this case, the buildBatchSQLUpdate method example must be modified slightly for implementing an optimistic approach. Several possible ways exist for using an optimistic approach. A typical way is to have either a timestamp column or sequence number counter column for versioning each update of the row. Assume that the employee table has a sequence number column that increments each time the row is updated. You then modify the signature of the buildBatchSQLUpdate method so that it is passed the LogElement object instead of the key and value pair. It also needs to use the OptimisticCallback object that is plugged into the backing map for getting both the initial version object and for updating the version object. The following is an example of a modified buildBatchSQLUpdate method that uses the ivOcb instance variable that was initialized as described in the preloadMap section:

modified batch-update method code example

```
private void buildBatchSQLUpdate( TxID tx, LogElement le, Connection conn )
    throws SQLException, LoaderException
{
    // Get the initial version object when this map entry was last read
    // or updated in the database.
    Employee emp = (Employee) le.getCurrentValue();
    long initialVersion = ((Long) le.getVersionedValue()).longValue();
    // Get the version object from the updated Employee for the SQL update
    //operation.
    Long currentVersion = (Long)ivOcb.getVersionedObjectForValue( emp );
    long nextVersion = currentVersion.longValue();
    // Now build SQL update that includes the version object in where clause
    // for optimistic checking.
    String sql = "update EMPLOYEE set LASTNAME = ?, FIRSTNAME = ?,
    DEPTNO = ?,SEQNO = ?, MGRNO = ? where EMPNO = ? and SEQNO = ?";
    PreparedStatement sqlUpdate = ivTcb.getPreparedStatement( tx, conn,
    "employee", sql );
    sqlUpdate.setString(1, emp.getLastName());
    sqlUpdate.setString(2, emp.getFirstName());
    sqlUpdate.setString(3, emp.getDepartmentName());
    sqlUpdate.setLong(4, nextVersion );
    sqlUpdate.setInt(5, emp.getManagerNumber());
    sqlUpdate.setInt(6, key);
    sqlUpdate.setLong(7, initialVersion);
    sqlUpdate.addBatch();
}
```

The example shows that the LogElement is used to obtain the initial version value. When the transaction first accesses the map entry, a LogElement is created with the initial Employee object that is obtained from the map. The initial Employee object is also passed to the getVersionedObjectForValue method on the OptimisticCallback interface and the result is saved in the LogElement. This processing occurs before an application is given a reference to the initial Employee object and has a chance to call some method that changes the state of the initial Employee object.

The example shows that the Loader uses the `getVersionObjectForValue` method to obtain the version object for the current updated Employee object. Before calling the `batchUpdate` method on the Loader interface, eXtreme Scale calls the `updateVersionedObjectForValue` method on the `OptimisticCallback` interface to cause a new version object to be generated for the updated Employee object. After the `batchUpdate` method returns to the ObjectGrid, the `LogElement` is updated with the current version object and becomes the new initial version object. This step is necessary because the application might have called the `flush` method on the map instead of the `commit` method on the Session. It is possible for the Loader to be called multiple times by a single transaction for the same key. For that reason, eXtreme Scale ensures that the `LogElement` is updated with the new version object each time the row is updated in the employee table.

Now that the Loader has both the initial version object and the next version object, it can run an SQL update statement that sets the `SEQNO` column to the next version object value and uses the initial version object value in the where clause. This approach is sometimes referred to as an overqualified update statement. The use of the overqualified update statement allows the relational database to verify that the row was not changed by some other transaction between the time that this transaction read the data from the database and the time that this transaction updates the database. If another transaction modified the row, then the count array that is returned by the batch update indicates that zero rows were updated for this key. The Loader is responsible for verifying that the SQL update operation did update the row. If it does not, the Loader displays a `com.ibm.websphere.objectgrid.plugins.OptimisticCollisionException` exception to inform the Session that the `batchUpdate` method failed due to more than one concurrent transaction trying to update the same row in the database table. This exception causes the Session to roll back and the application must retry the entire transaction. The rationale is that the retry will be successful, which is why this approach is called optimistic. The optimistic approach performs better if data is infrequently changed or concurrent transactions rarely try to update the same row.

It is important for the Loader to use the key parameter of the `OptimisticCollisionException` constructor to identify which key or set of keys caused the optimistic `batchUpdate` method to fail. The key parameter can either be the key object itself or an array of key objects if more than one key resulted in optimistic update failure. And eXtreme Scale uses the `getKey` method of the `OptimisticCollisionException` constructor to determine which map entries contain stale data and caused the exception to result. Part of the rollback processing is to evict each stale map entry from the map. Evicting stale entries is necessary so that any subsequent transaction that accesses the same key or keys results in the `get` method of the Loader interface being called to refresh the map entries with the current data from the database.

Other ways for a Loader to implement an optimistic approach include:

- No timestamp or sequence number column exists. In this case, the `getVersionObjectForValue` method on the `OptimisticCallback` interface simply returns the value object itself as the version. With this approach, the Loader needs to build a where clause that includes each of the fields of the initial version object. This approach is not efficient, and not all column types are eligible to be used in the where clause of an overqualified SQL update statement. This approach is typically not used.
- No timestamp or sequence number column exists. However, unlike the prior approach, the where clause only contains the value fields that were modified by the transaction. One method to detect which fields are modified is to set the

copy mode on the backing map to be CopyMode.COPY_ON_WRITE mode. This copy mode requires that a value interface be passed to the setCopyMode method on the BackingMap interface. The BackingMap creates dynamic proxy objects that implement the provided value interface. With this copy mode, the Loader can cast each value to a com.ibm.websphere.objectgrid.plugins.ValueProxyInfo object. The ValueProxyInfo interface has a method that allows the Loader to obtain the List of attribute names that were changed by the transaction. This method enables the Loader to call the get methods on the value interface for the attribute names to obtain the changed data and to build an SQL update statement that only sets the changed attributes. The where clause can now be built to have the primary key column plus each of the changed attribute columns. This approach is more efficient than the prior approach, but it requires more code to be written in the Loader and leads to the possibility that the prepared statement cache needs to be larger to handle the different permutations. However, if transactions typically only modify a few of the attributes, this limitation might not be a problem.

- Some relational databases might have an API to assist in automatically maintaining column data that is useful for optimistic versioning. Consult your database documentation to determine if this possibility exists.

Writing a loader

You can write your own loader plug-in implementation in your applications, which must follow the common WebSphere eXtreme Scale plug-in conventions.

Including a loader plug-in

The Loader interface has the following definition:

```
public interface Loader
{
    static final SpecialValue KEY_NOT_FOUND;
    List get(TxID txid, List keyList, boolean forUpdate) throws LoaderException;
    void batchUpdate(TxID txid, LogSequence sequence) throws
        LoaderException, OptimisticCollisionException;
    void preloadMap(Session session, BackingMap backingMap) throws LoaderException;
}
```

See “Loaders” on page 97 for more information.

get method

The backing map calls the Loader get method to get the values that are associated with a key list that is passed as the keyList argument. The get method is required to return a java.lang.util.List list of values, one value for each key that is in the key list. The first value that is returned in the value list corresponds to the first key in the key list, the second value returned in the value list corresponds to the second key in the key list, and so on. If the loader does not find the value for a key in the key list, the Loader is required to return the special KEY_NOT_FOUND value object that is defined in the Loader interface. Because a backing map can be configured to allow null as a valid value, it is very important for the Loader to return the special KEY_NOT_FOUND object when the Loader cannot find the key. This special value allows the backing map to distinguish between a null value and a value that does not exist because the key was not found. If a backing map does not support null values, a Loader that returns a null value instead of the KEY_NOT_FOUND object for a key that does not exist results in an exception.

The forUpdate argument tells the Loader if the application called a get method on the map or a getForUpdate method on the map. See the ObjectMap interface in the API documentation for more information. The Loader is responsible for implementing a concurrency control policy that controls concurrent access to the

persistent store. For example, many relational database management systems support the for update syntax on the SQL select statement that is used to read data from a relational table. The Loader can choose to use the for update syntax on the SQL select statement based on whether boolean true is passed as the argument value for the forUpdate parameter of this method. Typically, the Loader uses the for update syntax only when the pessimistic concurrency control policy is used. For an optimistic concurrency control, the Loader never uses for update syntax on the SQL select statement. The Loader is responsible to decide to use the forUpdate argument based on the concurrency control policy that is being used by the Loader.

For an explanation of the txid parameter, see “Plug-ins for managing transaction life cycle events” on page 351.

batchUpdate method

The batchUpdate method is important on the Loader interface. This method is called whenever the eXtreme Scale needs to apply all the current changes to the Loader. The Loader is given a list of changes for the selected Map. The changes are iterated and applied to the backend. The method receives the current TxID value and the changes to apply. The following sample iterates over the set of changes and batches three Java database connectivity (JDBC) statements, one with insert, another with update, and one with delete.

```
import java.util.Collection;
import java.util.Map;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.TxID;
import com.ibm.websphere.objectgrid.plugins.Loader;
import com.ibm.websphere.objectgrid.plugins.LoaderException;
import com.ibm.websphere.objectgrid.plugins.LogElement;
import com.ibm.websphere.objectgrid.plugins.LogSequence;

public void batchUpdate(TxID tx, LogSequence sequence) throws LoaderException {
    // Get a SQL connection to use.
    Connection conn = getConnection(tx);
    try {
        // Process the list of changes and build a set of prepared
        // statements for executing a batch update, insert, or delete
        // SQL operation.
        Iterator iter = sequence.getPendingChanges();
        while (iter.hasNext()) {
            LogElement logElement = (LogElement) iter.next();
            Object key = logElement.getKey();
            Object value = logElement.getCurrentValue();
            switch (logElement.getType().getCode()) {
                case LogElement.CODE_INSERT:
                    buildBatchSQLInsert(tx, key, value, conn);
                    break;
                case LogElement.CODE_UPDATE:
                    buildBatchSQLUpdate(tx, key, value, conn);
                    break;
                case LogElement.CODE_DELETE:
                    buildBatchSQLDelete(tx, key, conn);
                    break;
            }
        }
        // Execute the batch statements that were built by above loop.
        Collection statements = getPreparedStatementCollection(tx, conn);
        iter = statements.iterator();
        while (iter.hasNext()) {
            PreparedStatement pstmt = (PreparedStatement) iter.next();
            pstmt.executeBatch();
        }
    } catch (SQLException e) {
        LoaderException ex = new LoaderException(e);
        throw ex;
    }
}
```

The preceding sample illustrates the high level logic of processing the LogSequence argument, but the details of how a SQL insert, update, or delete statement is built are not illustrated. Some of the key points that are illustrated include:

- The `getPendingChanges` method is called on the `LogSequence` argument to obtain an iterator over the list of `LogElements` that the `Loader` needs to process.
- The `LogElement.getType().getCode()` method is used to determine if the `LogElement` is for a SQL insert, update, or delete operation.
- An `SQLException` exception is caught and is chained to a `LoaderException` exception that prints to report that an exception occurred during the batch update.
- JDBC batch update support is used to minimize the number of queries to the backend that must be made.

preloadMap method

During the eXtreme Scale initialization, each backing map that is defined is initialized. If a `Loader` is plugged into a backing map, the backing map invokes the `preloadMap` method on the `Loader` interface to allow the loader to prefetch data from its back end and load the data into the map. The following sample assumes the first 100 rows of an `Employee` table is read from the database and is loaded into the map. The `EmployeeRecord` class is an application-provided class that holds the employee data that is read from the employee table.

Note: This sample fetches all the data from database and then insert them into the base map of one partition. In a real-world distributed eXtreme Scale deployment scenario, data should be distributed into all the partitions. Refer to “Developing client-based JPA loaders” on page 367 for more information.

```
import java.sql.PreparedStatement;
import java.sql.SQLException;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.TxID;
import com.ibm.websphere.objectgrid.plugins.Loader;
import com.ibm.websphere.objectgrid.plugins.LoaderException

public void preloadMap(Session session, BackingMap backingMap) throws LoaderException {
    boolean tranActive = false;
    ResultSet results = null;
    Statement stmt = null;
    Connection conn = null;
    try {
        session.beginNoWriteThrough();
        tranActive = true;
        ObjectMap map = session.getMap(backingMap.getName());
        TxID tx = session.getTxID();
        // Get a auto-commit connection to use that is set to
        // a read committed isolation level.
        conn = getAutoCommitConnection(tx);
        // Preload the Employee Map with EmployeeRecord
        // objects. Read all Employees from table, but
        // limit preload to first 100 rows.
        stmt = conn.createStatement();
        results = stmt.executeQuery(SELECT_ALL);
        int rows = 0;
        while (results.next() && rows < 100) {
            int key = results.getInt(EMPNO_INDEX);
            EmployeeRecord emp = new EmployeeRecord(key);
            emp.setLastName(results.getString(LASTNAME_INDEX));
            emp.setFirstName(results.getString(FIRSTNAME_INDEX));
            emp.setDepartmentName(results.getString(DEPTNAME_INDEX));
            emp.updateSequenceNumber(results.getLong(SEQNO_INDEX));
            emp.setManagerNumber(results.getInt(MGRNO_INDEX));
            map.put(new Integer(key), emp);
            ++rows;
        }
        // Commit the transaction.
        session.commit();
        tranActive = false;
    } catch (Throwable t) {
        throw new LoaderException("preload failure: " + t, t);
    } finally {
        if (tranActive) {
            try {
                session.rollback();
            } catch (Throwable t2) {
                // Tolerate any rollback failures and
                // allow original Throwable to be thrown.
            }
        }
    }
}
```

```

    }
    // Be sure to clean up other databases resources here
    // as well such a closing statements, result sets, etc.
}
}
}

```

This sample illustrates the following key points:

- The `preloadMap` backing map uses the `Session` object that is passed to it as the session argument.
- The `Session.beginNoWriteThrough` method is used to begin the transaction instead of the `begin` method.
- The `Loader` cannot be called for each put operation that occurs in this method for loading the map.
- The `Loader` can map columns of the employee table to a field in the `EmployeeRecord` Java object. The `Loader` catches all throwable exceptions that occur and throws a `LoaderException` exception with the caught throwable exception chained to it.
- The `finally` block ensures that any throwable exception that occurs between the time the `beginNoWriteThrough` method is called and the `commit` method is called cause the `finally` block to roll back the active transaction. This action is critical to ensure that any transaction that has been started by the `preloadMap` method is completed before returning to the caller. The `finally` block is a good place to perform other cleanup actions that might be needed, like closing the Java Database Connectivity (JDBC) connection and other JDBC objects.

The `preloadMap` sample is using a SQL select statement that selects all rows of the table. In your application-provided `Loader`, you might need to set one or more `Loader` properties to control how much of the table needs to be preloaded into the map.

Because the `preloadMap` method is only called one time during the `BackingMap` initialization, it is also a good place to run the one time `Loader` initialization code. Even if a `Loader` chooses not to prefetch data from the backend and load the data into the map, it probably needs to perform some other one time initialization to make other methods of the `Loader` more efficient. The following example illustrates caching the `TransactionCallback` object and `OptimisticCallback` object as instance variables of the `Loader` so that the other methods of the `Loader` do not have to make method calls to get access to these objects. This caching of the `ObjectGrid` plug-in values can be performed because after the `BackingMap` is initialized, the `TransactionCallback` and the `OptimisticCallback` objects cannot be changed or replaced. It is acceptable to cache these object references as instance variables of the `Loader`.

```

import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.plugins.OptimisticCallback;
import com.ibm.websphere.objectgrid.plugins.TransactionCallback;

// Loader instance variables.
MyTransactionCallback ivTcb; // MyTransactionCallback

// extends TransactionCallback
MyOptimisticCallback ivOcb; // MyOptimisticCallback

// implements OptimisticCallback
// ...
public void preloadMap(Session session, BackingMap backingMap) throws LoaderException
[Replication programming]
    // Cache TransactionCallback and OptimisticCallback objects
    // in instance variables of this Loader.
    ivTcb = (MyTransactionCallback) session.getObjectGrid().getTransactionCallback();
    ivOcb = (MyOptimisticCallback) backingMap.getOptimisticCallback();
    // The remainder of preloadMap code (such as shown in prior example).
}

```

For information about preloading and recoverable preloading as it pertains to replication failover, see Replication for availabilitythe information about replication in the *Product Overview*.

Loaders with entity maps

If the loader is plugged into an entity map, the loader must handle tuple objects. Tuple objects are a special entity data format. The loader must conversion data between tuple and other data formats. For example, the get method returns a list of values that correspond to the set of keys that are passed in to the method. The passed-in keys are in the type of Tuple, says key tuples. Assuming that the loader persists the map with a database using JDBC, the get method must convert each key tuple into a list of attribute values that correspond to the primary key columns of the table that is mapped to the entity map, run the SELECT statement with the WHERE clause that uses converted attribute values as criteria to fetch data from database, and then convert the returned data into value tuples. The get method gets data from the database and converts the data into value tuples for passed-in key tuples, and then returns a list of value tuples correspond to the set of tuple keys that are passed in to the caller. The get method can perform one SELECT statement to fetch all data at one time, or run a SELECT statement for each key tuple. For programming details that show how to use the Loader when the data is store using an entity manager, see “Using a loader with entity maps and tuples” on page 342.

Map pre-loading

Maps can be associated with Loaders. A loader is used to fetch objects when they cannot be found in the map (a cache miss) and also to write changes to a back-end when a transaction commits. Loaders can also be used for pre-loading data into a map. The preloadMap method of the Loader interface is called on each map when its corresponding partition in the map set becomes a primary. The preloadMap method is not called on replicas. It attempts to load all the intended referenced data from the back-end into the map using the provided session. The relevant map is identified by the BackingMap argument that is passed to the preloadMap method.

```
void preloadMap(Session session, BackingMap backingMap) throws LoaderException;
```

Pre-loading in partitioned map set

Maps can be partitioned into N partitions. Maps can therefore be striped across multiple servers, with each entry identified by a key that is stored only on one of those servers. Very large maps can be held in a data grid because the application is no longer limited by the heap size of a single Java virtual machine (JVM) to hold all the entries of a Map. Applications that want to preload with the preloadMap method of the Loader interface must identify the subset of the data that it preloads. A fixed number of partitions always exists. You can determine this number by using the following code example:

```
int numPartitions = backingMap.getPartitionManager().getNumOfPartitions();  
int myPartition = backingMap.getPartitionId();
```

This code example shows how an application can identify the subset of the data to preload from the database. Applications must always use these methods even when the map is not initially partitioned. These methods allow flexibility: If the map is later partitioned by the administrators, then the loader continues to work correctly.

The application must issue queries to retrieve the *myPartition* subset from the backend. If a database is used, then it might be easier to have a column with the partition identifier for a given record unless there is some natural query that allows the data in the table to partition easily.

See “Writing a loader with a replica preload controller” on page 346 for an example on how to implement a Loader for a replicated data grid.

Performance

The preload implementation copies data from the back-end into the map by storing multiple objects in the map in a single transaction. The optimal number of records to store per transaction depends on several factors, including complexity and size. For example, after the transaction includes blocks of more than 100 entries, the performance benefit decreases as you increase the number of entries. To determine the optimal number, begin with 100 entries and then increase the number until the performance benefit decreases to none. Larger transactions result in better replication performance. Remember, only the primary runs the preload code. The preloaded data is replicated from the primary to any replicas that are online.

Pre-loading map set

If the application uses a map set with multiple maps then each map has its own loader. Each loader has a preload method. Each map is loaded serially by the data grid. It might be more efficient to preload all the maps by designating a single map as the pre-loading map. This process is an application convention. For example, two maps, department and employee, might use the department Loader to preload both the department and the employee maps. This procedure ensures that, transactionally, if an application wants a department then the employees for that department are in the cache. When the department Loader preloads a department from the back-end, it also fetches the employees for that department. The department object and its associated employee objects are then added to the map using a single transaction.

Recoverable pre-loading

Some customers have very large data sets that need caching. Pre-loading this data can be very time consuming. Sometimes, the pre-loading must complete before the application can go online. You can benefit from making pre-loading recoverable. Suppose there are a million records to preload. The primary is pre-loading them and fails at the 800,000th record. Normally, the replica chosen to be the new primary clears any replicated state and starts from the beginning. eXtreme Scale can use a `ReplicaPreloadController` interface. The loader for the application would also need to implement the `ReplicaPreloadController` interface. This example adds a single method to the Loader: `Status checkPreloadStatus(Session session, BackingMap bmap);`. This method is called by the eXtreme Scale run time before the preload method of the Loader interface is normally called. The eXtreme Scale tests the result of this method (Status) to determine its behavior whenever a replica is promoted to a primary.

Table 7. Status value and response

Returned status value	eXtreme Scale response
Status.PRELOADED_ALREADY	eXtreme Scale does not call the preload method at all because this status value indicates that the map is fully preloaded.
Status.FULL_PRELOAD_NEEDED	eXtreme Scale clears the map and calls the preload method normally.

Table 7. Status value and response (continued)

Returned status value	eXtreme Scale response
Status.PARTIAL_PRELOAD_NEEDED	eXtreme Scale leaves the map as-is and calls preload. This strategy allows the application loader to continue pre-loading from that point onwards.

Clearly, while a primary is pre-loading the map, it must leave some state in a map in the MapSet that is being replicated so that the replica determines what status to return. You can use an extra map named, for example, RecoveryMap map. This RecoveryMap map must be part of the same MapSet map set that is being preloaded to ensure that the map is replicated consistently with the data being preloaded. A suggested implementation follows.

As the preload commits each block of records, the process also updates a counter or value in the RecoveryMap map as part of that transaction. The preloaded data and the RecoveryMap map data are replicated atomically to the replicas. When the replica is promoted to primary, it can now check the RecoveryMap map to see what has happened.

The RecoveryMap map can hold a single entry with the state key. If no object exists for this key then you need a full preload (checkPreloadStatus returns FULL_PRELOAD_NEEDED). If an object exists for this state key and the value is COMPLETE, the preload completes, and the checkPreloadStatus method returns PRELOADED_ALREADY. Otherwise, the value object indicates where the preload restarts and the checkPreloadStatus method returns PARTIAL_PRELOAD_NEEDED. The loader can store the recovery point in an instance variable for the loader so that when preload is called, the loader knows the starting point. The RecoveryMap map can also hold an entry per map if each map is preloaded independently.

Handling recovery in synchronous replication mode with a Loader

The eXtreme Scale run time is designed not to lose committed data when the primary fails. The following section shows the algorithms used. These algorithms apply only when a replication group uses synchronous replication. A loader is optional.

The eXtreme Scale run time can be configured to replicate all changes from a primary to the replicas synchronously. When a synchronous replica is placed, it receives a copy of the existing data on the primary shard. During this time, the primary continues to receive transactions and copies them to the replica asynchronously. The replica is not considered to be online at this time.

After the replica catches up the primary, the replica enters peer mode and synchronous replication begins. Every transaction committed on the primary is sent to the synchronous replicas and the primary waits for a response from each replica. A synchronous commit sequence with a Loader on the primary looks like the following set of steps:

Table 8. Commit sequence on the primary

Step with loader	Step without loader
Get locks for entries	same
Flush changes to the loader	no-op
Save changes to the cache	same

Table 8. Commit sequence on the primary (continued)

Step with loader	Step without loader
Send changes to replicas and wait for acknowledgement	same
Commit to the loader through the TransactionCallback plug-in	plug-in commit called, but does nothing
Release locks for entries	same

Notice that the changes are sent to the replica before they are committed to the loader. To determine when the changes are committed on the replica, revise this sequence: At initialize time, initialize the tx lists on the primary as below.

CommittedTx = {}, RolledBackTx = {}

During synchronous commit processing, use the following sequence:

Table 9. Synchronous commit processing

Step with loader	Step without loader
Get locks for entries	same
Flush changes to the loader	no-op
Save changes to the cache	same
Send changes with a committed transaction, roll back transaction to replica, and wait for acknowledgement	same
Clear list of committed transactions and rolled back transactions	same
Commit the loader through the TransactionCallBack plug-in	TransactionCallBack plug-in commit is still called, but typically does not do anything
If commit succeeds, add the transaction to the committed transactions, otherwise add to the rolled back transactions	no-op
Release locks for entries	same

For replica processing, use the following sequence:

1. Receive changes
2. Commit all received transactions in the committed transaction list
3. Roll back all received transactions in the rolled back transaction list
4. Start a transaction or session
5. Apply changes to the transaction or session
6. Save the transaction or session to the pending list
7. Send back reply

Notice that on the replica, no loader interactions occur while the replica is in replica mode. The primary must push all changes through the Loader. The replica does not change data. A side effect of this algorithm is that the replica always has the transactions, but they are not committed until the next primary transaction sends the commit status of those transactions. The transactions are then committed or rolled back on the replica. Until then, the transactions are not committed. You can add a timer on the primary that sends the transaction outcome after a small time period (a few seconds). This timer limits, but does not eliminate, any staleness

to that time window. This staleness is only a problem when using replica read mode. Otherwise, the staleness does not have an impact on the application.

When the primary fails, it is likely that a few transactions were committed or rolled back on the primary, but the message never made it to the replica with these outcomes. When a replica is promoted to the new primary, one of the first actions is to handle this condition. Each pending transaction is reprocessed against the new primary's set of maps. If there is a Loader, then each transaction is given to the Loader. These transactions are applied in strict first in first out (FIFO) order. If a transaction fails, it is ignored. If three transactions are pending, A, B, and C, then A might commit, B might rollback, and C might also commit. No one transaction has any impact on the others. Assume that they are independent.

A loader might want to use slightly different logic when it is in failover recovery mode versus normal mode. The loader can easily know when it is in failover recovery mode by implementing the `ReplicaPreloadController` interface. The `checkPreloadStatus` method is only called when failover recovery completes. Therefore, if the `apply` method of the Loader interface is called before the `checkPreloadStatus` method, then it is a recovery transaction. After the `checkPreloadStatus` method is called, the failover recovery is complete.

Configuring write-behind loader support

You can enable write-behind support either using the ObjectGrid descriptor XML file or programmatically using the `BackingMap` interface.

Use either the ObjectGrid descriptor XML file to enable write-behind support, or programmatically by using the `BackingMap` interface.

ObjectGrid descriptor XML file

When configuring an ObjectGrid using an ObjectGrid descriptor XML file, the write-behind loader is enabled by setting the `writeBehind` attribute on the `backingMap` tag. An example follows:

```
<objectGrid name="library" >
  <backingMap name="book" writeBehind="T300;C900" pluginCollectionRef="bookPlugins"/>
</objectGrid>
```

In the previous example, write-behind support of the book backing map is enabled with parameter `T300;C900`. The write-behind attribute specifies the maximum update time and/or a maximum key update count. The format of the write-behind parameter is:

```
write-behind attribute ::= <defaults> | <update time> | <update key count> | <update time> ";" <update key count>
update time ::= "T" <positive integer>
update key count ::= "C" <positive integer>
defaults ::= "" {table}
```

Updates to the loader occur when one of the following events occurs:

1. The maximum update time in seconds has elapsed since the last update.
2. The number of updated keys in the queue map has reached the update key count.

These parameters are only hints. The real update count and update time will be within close range of the parameters. However, we do not guarantee that the actual update count or update time are the same as defined in the parameters. Also, the first behind update could happen after up to twice as long as the update time. This is because ObjectGrid randomizes the update starting time so all partitions will not hit the database simultaneously.

In the previous example T300;C900, the loader writes the data to the back-end when 300 seconds have passed since the last update or when 900 keys are pending to be updated. The default update time is 300 seconds and the default update key count is 1000.

Table 10. Some write-behind options

Attribute value	Time
T100	The update time is 100 seconds, and the update key count is 1000 (the default value)
C2000	The update time is 300 seconds (the default value), and the update key count is 2000.
T300;C900	The update time is 300 seconds and the update key count is 900.
""	The update time is 300 second (the default value), and the update key count is 1000 (the default value). Note: If you configure the write-behind loader as an empty string: writeBehind="", the write-behind loader is enabled using the default values. Therefore, do not specify the writeBehind attribute if you do not want write-behind support enabled.

Programmatically enabling write-behind support

When you are creating a backing map programmatically for a local, in-memory eXtreme Scale, you can use the following method on the BackingMap interface to enable and disable write-behind support.

```
public void setWriteBehind(String writeBehindParam);
```

For more details about how to use the setWriteBehind method, see the information about the BackingMap interface in the *Programming Guide*.

Write-behind caching:

You can use write-behind caching to reduce the overhead that occurs when updating a database you are using as a back end.

Write-behind caching overview

Write-behind caching asynchronously queues updates to the Loader plug-in. You can improve performance by disconnecting updates, inserts, and removes for a map, the overhead of updating the back-end database. The asynchronous update is performed after a time-based delay (for example, five minutes) or an entry-based delay (1000 entries).

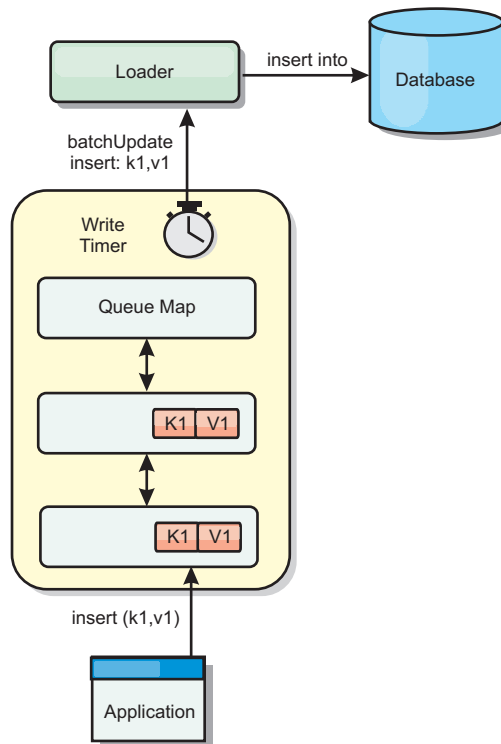


Figure 28. Write-behind caching

The write-behind configuration on a `BackingMap` creates a thread between the loader and the map. The loader then delegates data requests through the thread according to the configuration settings in the `BackingMap.setWriteBehind` method. When an eXtreme Scale transaction inserts, updates, or removes an entry from a map, a `LogElement` object is created for each of these records. These elements are sent to the write-behind loader and queued in a special `ObjectMap` called a queue map. Each backing map with the write-behind setting enabled has its own queue maps. A write-behind thread periodically removes the queued data from the queue maps and pushes them to the real back-end loader.

The write-behind loader only sends insert, update, and delete types of `LogElement` objects to the real loader. All other types of `LogElement` objects, for example, `EVICT` type, are ignored.

Write-behind support is an extension of the Loader plug-in, which you use to integrate eXtreme Scale with the database. For example, consult the [Configuring JPA loaders](#) information about configuring a JPA loader.

Benefits

Enabling write-behind support has the following benefits:

- **Back end failure isolation:** Write-behind caching provides an isolation layer from back end failures. When the back-end database fails, updates are queued in the queue map. The applications can continue driving transactions to eXtreme Scale. When the back end recovers, the data in the queue map is pushed to the back-end.
- **Reduced back end load:** The write-behind loader merges the updates on a key basis so only one merged update per key exists in the queue map. This merge decreases the number of updates to the back-end database.

- **Improved transaction performance:** Individual eXtreme Scale transaction times are reduced because the transaction does not need to wait for the data to be synchronized with the back-end.

Write-behind loader application design considerations:

When you implement a write-behind loaders, you must consider several issues such as integrity constraints, locking behavior, and performance.

Application design considerations

Enabling write-behind support is simple, but designing an application to work with write-behind support needs careful consideration. Without write-behind support, the ObjectGrid transaction encloses the back-end transaction. The ObjectGrid transaction starts before the back-end transaction starts, and it ends after the back-end transaction ends.

With write-behind support enabled, the ObjectGrid transaction finishes before the back-end transaction starts. The ObjectGrid transaction and back-end transaction are de-coupled.

Referential integrity constraints

Each backing map that is configured with write-behind support has its own write-behind thread to push the data to the back-end. Therefore, the data that updated to different maps in one ObjectGrid transaction are updated to the back-end in different back-end transactions. For example, transaction T1 updates key key1 in map Map1 and key key2 in map Map2. The key1 update to map Map1 is updated to the back-end in one back-end transaction, and the key2 updated to map Map2 is updated to the back-end in another back-end transaction by different write-behind threads. If data stored in Map1 and Map2 have relations, such as foreign key constraints in the back-end, the updates might fail.

When designing the referential integrity constraints in your back-end database, ensure that out-of-order updates are allowed.

Queue map locking behavior

Another major transaction behavior difference is the locking behavior. ObjectGrid supports three different locking strategies: PESSIMISTIC, OPTIMISITIC, and NONE. The write-behind queue maps uses pessimistic locking strategy no matter which lock strategy is configured for its backing map. Two different types of operations exist that acquire a lock on the queue map:

- When an ObjectGrid transaction commits, or a flush (map flush or session flush) happens, the transaction reads the key in the queue map and places an S lock on the key.
- When an ObjectGrid transaction commits, the transaction tries to upgrade the S lock to X lock on the key.

Because of this extra queue map behavior, you can see some locking behavior differences.

- If the user map is configured as PESSIMISTIC locking strategy, there isn't much locking behavior difference. Every time a flush or commit is called, an S lock is

placed on the same key in the queue map. During the commit time, not only is an X lock acquired for key in the user map, it is also acquired for the key in the queue map.

- If the user map is configured as OPTIMISTIC or NONE locking strategy, the user transaction will follow the PESSIMISTIC locking strategy pattern. Every time a flush or commit is called, an S lock is acquired for the same key in the queue map. During the commit time, an X lock is acquired for the key in the queue map using the same transaction.

Loader transaction retries

ObjectGrid does not support 2-phase or XA transactions. The write-behind thread removes records from the queue map and updates the records to the back-end. If the server fails in the middle of the transaction, some back-end updates can be lost.

The write-behind loader will automatically retry to write failed transactions and will send an in-doubt LogSequence to the back-end to avoid data loss. This action requires the loader to be idempotent, which means when the `Loader.batchUpdate(TxId, LogSequence)` is called twice with the same value, it gives the same result as if it were applied one time. Loader implementations must implement the `RetryableLoader` interface to enable this feature. See the API documentation for more details.

Write-behind caching performance considerations

Write-behind caching support improves response time by removing the loader update from the transaction. It also increases database throughput because database updates are combined. It is important to understand the overhead introduced by write-behind thread, which pulls the data out of the queue map and pushed to the loader.

The maximum update count or the maximum update time need to be adjusted based on the expected usage patterns and environment. If the value of the maximum update count or the maximum update time is too small, the overhead of the write-behind thread may exceed the benefits. Setting a large value for these two parameters could also increase the memory usage for queuing the data and increase the stale time of the database records.

For best performance, tune the write-behind parameters based on the following factors:

- Ratio of read and write transactions
- Same record update frequency
- Database update latency.

Handling failed write-behind updates:

Because the WebSphere eXtreme Scale transaction finishes before the back-end transaction starts, the transaction could have false success.

If you try to insert an entry in an eXtreme Scale transaction that does not exist in the backing map but exists in the backend database, causing a duplicate key, the eXtreme Scale transaction succeeds. However, the transaction in which the write-behind thread inserts the object into the backend database fails with a duplicate key exception.

Handling failed write-behind updates: client side

Such an update, or any other failed back-end update, is a failed write-behind update. Failed write-behind updates are stored in a failed write-behind update map. This map serves as an event queue for failed updates. The key of the update is a unique Integer object, and the value is an instance of FailedUpdateElement. The failed write-behind update map is configured with an evictor, which evicts the records one hour after it has been inserted. So the failed-update records are lost if they are not retrieved within one hour.

The ObjectMap API can be used to retrieve the failed write-behind update map entries. The failed write-behind update map name is: IBM_WB_FAILED_UPDATES_<map name>. See the WriteBehindLoaderConstants API documentation for the prefix names of each of the write-behind system maps. The following is an example.

process failed - example code

```
ObjectMap failedMap = session.getMap(
    WriteBehindLoaderConstants.WRITE_BEHIND_FAILED_UPDATES_MAP_PREFIX + "Employee");
Object key = null;

session.begin();
while(key = failedMap.getNextKey(ObjectMap.QUEUE_TIMEOUT_NONE)) {
    FailedUpdateElement element = (FailedUpdateElement) failedMap.get(key);
    Throwable throwable = element.getThrowable();
    Object failedKey = element.getKey();
    Object failedValue = element.getAfterImage();
    failedMap.remove(key);
    // Do something interesting with the key, value, or exception.
}
session.commit();
```

A getNextKey method call works with a specific partition for each eXtreme Scale transaction. In a distributed environment, to get keys from all partitions, you must start multiple transactions, as shown in the following example:

getting keys from all partitions - example code

```
ObjectMap failedMap = session.getMap(
    WriteBehindLoaderConstants.WRITE_BEHIND_FAILED_UPDATES_MAP_PREFIX + "Employee");
while (true) {
    session.begin();
    Object key = null;
    while(( key = failedMap.getNextKey(5000) )!= null ) {
        FailedUpdateElement element = (FailedUpdateElement) failedMap.get(key);
        Throwable throwable = element.getThrowable();
        Object failedKey = element.getKey();
        Object failedValue = element.getAfterImage();
        failedMap.remove(key);
        // Do something interesting with the key, value, or exception.
    }
    Session.commit();
}
```

Note: The failed update map provides a way to monitor the application health. If a system produces many records in the failed update map, it is a sign that you need to revise the application or architecture to use the write-behind support. You can use the **xscmd -showMapSizes** command to see the failed update map entry size.

Handling failed write-behind updates: shard listener

It is important to detect and log when a write-behind transaction fails. Every application using write-behind needs to implement a watcher to handle failed write-behind updates. This avoids potentially running out of memory as records in the bad update Map are not evicted because the application is expected to handle them.

The following code shows how to plug in such a watcher, or "dumper," which must be added to the ObjectGrid descriptor XML as in the snippet.

```
<objectGrid name="Grid">
  <bean id="ObjectGridEventListener" className="utils.WriteBehindDumper"/>
```

You can see the ObjectGridEventListener bean that has been added, which is the write-behind watcher referred to above. The watcher interacts over the Maps for all primary shards in a JVM looking for ones with write-behind enabled. If it finds one then it tries to log up to 100 bad updates. It keeps watching a primary shard until the shard is moved to a different JVM. All applications using write-behind must use a watcher similar to this one. Otherwise, the Java virtual machines run out of memory because this error map is never evicted

See "Example: Writing a write-behind dumper class" for more information.

Example: Writing a write-behind dumper class:

This sample source code shows how to write a watcher (dumper) to handle failed write-behind updates.

```
//
//This sample program is provided AS IS and may be used, executed, copied and
//modified without royalty payment by customer (a) for its own instruction and
//study, (b) in order to develop applications designed to run with an IBM
//WebSphere product, either for customer's own internal use or for redistribution
//by customer, as part of such an application, in customer's own products. "
//
//5724-J34 (C) COPYRIGHT International Business Machines Corp. 2009
//All Rights Reserved * Licensed Materials - Property of IBM
//
package utils;

import java.util.Collection;
import java.util.Iterator;
import java.util.concurrent.Callable;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.ScheduledFuture;
import java.util.concurrent.ScheduledThreadPoolExecutor;
import java.util.concurrent.TimeUnit;
import java.util.logging.Logger;

import com.ibm.websphere.objectgrid.BackingMap;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridRuntimeException;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.UndefinedMapException;
import com.ibm.websphere.objectgrid.plugins.ObjectGridEventGroup;
import com.ibm.websphere.objectgrid.plugins.ObjectGridEventListener;
import com.ibm.websphere.objectgrid.writebehind.FailedUpdateElement;
import com.ibm.websphere.objectgrid.writebehind.WriteBehindLoaderConstants;

/**
 * Write behind expects transactions to the Loader to succeed. If a transaction for a key fails then
 * it inserts an entry in a Map called PREFIX + mapName. The application should be checking this
 * map for entries to dump out write behind transaction failures. The application is responsible for
 * analyzing and then removing these entries. These entries can be large as they include the key, before
 * and after images of the value and the exception itself. Exceptions can easily be 20k on their own.
 *
 * The class is registered with the grid and an instance is created per primary shard in a JVM. It creates
 * a single thread
 * and that thread then checks each write behind error map for the shard, prints out the problem and
 * then removes the entry.
 */
```

```

* This means there will be one thread per shard. If the shard is moved to another JVM then the deactivate
* method stops the thread.
* @author bnewport
*
*/
public class WriteBehindDumper implements ObjectGridEventListener, ObjectGridEventGroup.ShardEvents,
Callable<Boolean>
{
    static Logger logger = Logger.getLogger(WriteBehindDumper.class.getName());

    ObjectGrid grid;

    /**
     * Thread pool to handle table checkers. If the application has it's own pool
     * then change this to reuse the existing pool
     */
    static ScheduledExecutorService pool = new ScheduledThreadPoolExecutor(2); // two threads to dump records

    // the future for this shard
    ScheduledFuture<Boolean> future;

    // true if this shard is active
    volatile boolean isShardActive;

    /**
     * Normal time between checking Maps for write behind errors
     */
    final long BLOCKTIME_SECS = 20L;

    /**
     * An allocated session for this shard. No point in allocating them again and again
     */
    Session session;
    /**
     * When a primary shard is activated then schedule the checks to periodically check
     * the write behind error maps and print out any problems
     */
    public void shardActivated(ObjectGrid grid)
    {
        try
        {
            this.grid = grid;
            session = grid.getSession();

            isShardActive = true;
            future = pool.schedule(this, BLOCKTIME_SECS, TimeUnit.SECONDS); // check every BLOCKTIME_SECS seconds initially
        }
        catch(ObjectGridException e)
        {
            throw new ObjectGridRuntimeException("Exception activating write dumper", e);
        }
    }

    /**
     * Mark shard as inactive and then cancel the checker
     */
    public void shardDeactivate(ObjectGrid arg0)
    {
        isShardActive = false;
        // if it's cancelled then cancel returns true
        if(future.cancel(false) == false)
        {
            // otherwise just block until the checker completes
            while(future.isDone() == false) // wait for the task to finish one way or the other
            {
                try
                {
                    Thread.sleep(1000L); // check every second
                }
                catch(InterruptedException e)
                {
                }
            }
        }
    }

    /**
     * Simple test to see if the map has write behind enabled and if so then return
     * the name of the error map for it.
     * @param mapName The map to test
     * @return The name of the write behind error map if it exists otherwise null
     */
    static public String getWriteBehindNameIfPossible(ObjectGrid grid, String mapName)
    {
        BackingMap map = grid.getMap(mapName);
        if(map != null && map.getWriteBehind() != null)
        {
            return WriteBehindLoaderConstants.WRITE_BEHIND_FAILED_UPDATES_MAP_PREFIX + mapName;
        }
        else
            return null;
    }
}

```

```

}

/**
 * This runs for each shard. It checks if each map has write behind enabled and if it does
 * then it prints out any write behind
 * transaction errors and then removes the record.
 */
public Boolean call()
{
    logger.fine("Called for " + grid.toString());
    try
    {
        // while the primary shard is present in this JVM
        // only user defined maps are returned here, no system maps like write behind maps are in
        // this list.
        Iterator<String> iter = grid.getListOfMapNames().iterator();
        boolean foundErrors = false;
        // iterate over all the current Maps
        while(iter.hasNext() && isShardActive)
        {
            String origName = iter.next();

            // if it's a write behind error map
            String name = getWriteBehindNameIfPossible(grid, origName);
            if(name != null)
            {
                // try to remove blocks of N errors at a time
                ObjectMap errorMap = null;
                try
                {
                    errorMap = session.getMap(name);
                }
                catch(UndefinedMapException e)
                {
                    // at startup, the error maps may not exist yet, patience...
                    continue;
                }
                // try to dump out up to N records at once
                session.begin();
                for(int counter = 0; counter < 100; ++counter)
                {
                    Integer seqKey = (Integer)errorMap.getNextKey(1L);
                    if(seqKey != null)
                    {
                        foundErrors = true;
                        FailedUpdateElement elem = (FailedUpdateElement)errorMap.get(seqKey);
                        //
                        // Your application should log the problem here
                        logger.info("WriteBehindDumper ( " + origName + ") for key ( " + elem.getKey() + ") Exception: " +
                            elem.getThrowable().toString());
                        //
                        //
                        errorMap.remove(seqKey);
                    }
                    else
                        break;
                }
                session.commit();
            }
            // do next map
            // loop faster if there are errors
            if(isShardActive)
            {
                // reschedule after one second if there were bad records
                // otherwise, wait 20 seconds.
                if(foundErrors)
                    future = pool.schedule(this, 1L, TimeUnit.SECONDS);
                else
                    future = pool.schedule(this, BLOCKTIME_SECS, TimeUnit.SECONDS);
            }
        }
        catch(ObjectGridException e)
        {
            logger.fine("Exception in WriteBehindDumper" + e.toString());
            e.printStackTrace();

            //don't leave a transaction on the session.
            if(session.isTransactionActive())
            {
                try { session.rollback(); } catch(Exception e2) {}
            }
        }
        return true;
    }
}

public void destroy() {
    // TODO Auto-generated method stub
}

public void initialize(Session arg0) {

```

```

// TODO Auto-generated method stub
}

public void transactionBegin(String arg0, boolean arg1) {
// TODO Auto-generated method stub
}

public void transactionEnd(String arg0, boolean arg1, boolean arg2,
Collection arg3) {
// TODO Auto-generated method stub
}
}
}

```

JPA loader programming considerations

A Java Persistence API (JPA) Loader is a loader plug-in implementation that uses JPA to interact with the database. Use the following considerations when you develop an application that uses a JPA loader.

eXtreme Scale entity and JPA entity

You can designate any POJO class as an eXtreme Scale entity using eXtreme Scale entity annotations, XML configuration, or both. You can also designate the same POJO class as a JPA entity using JPA entity annotations, XML configuration, or both.

eXtreme Scale entity: An eXtreme Scale entity represents persistent data that is stored in ObjectGrid maps. An entity object is transformed into a key tuple and a value tuple, which are then stored as key-value pairs in the maps. A tuple is an array of primitive attributes.

JPA entity: A JPA entity represents persistent data that is stored in a relational database automatically using container-managed persistence. The data is persisted in some form of a data storage system in the appropriate form, such as database tuples in a database.

When an eXtreme Scale entity is persisted, its relations are stored in other entity maps. For example, when you are persisting a Consumer entity with a one-to-many relation to a ShippingAddress entity, if cascade-persist is enabled, the ShippingAddress entity is stored in the shippingAddress map in tuple format. If you are persisting a JPA entity, the related JPA entities are also persisted to database tables if cascade-persist is enabled. When a POJO class is designated as both an eXtreme Scale entity and a JPA entity, the data can be persisted to both ObjectGrid entity maps and databases. Common uses follow:

- **Preload scenario:** An entity is loaded from a database using a JPA provider and persists it into ObjectGrid entity maps.
- **Loader scenario:** A Loader implementation is plugged in for the ObjectGrid entity maps so an entity stored in ObjectGrid entity maps can be persisted into or loaded from a database using JPA providers.

It is also common that a POJO class is designated as a JPA entity only. In that case, what is stored in the ObjectGrid maps are the POJO instances, versus the entity tuples in the ObjectGrid entity case.

Application design considerations for entity maps

When you are plugging in a JPALoader interface, the object instances are directly stored in the ObjectGrid maps.

However, you are when plugging in a `JPAEntityLoader`, the entity class is designated as both an eXtreme Scale entity and a JPA entity. In that case, treat this entity as if it has two persistent stores: the ObjectGrid entity maps and the JPA persistence store. The architecture becomes more complicated than the `JPALoader` case.

For more information about the `JPAEntityLoader` plug-in and application design considerations, see the information about the `JPAEntityLoader` plug-in in the *Administration Guide*. This information can also help if you plan to implement your own loader for the entity maps.

Performance considerations

Ensure that you set the proper eager or lazy fetch type for relationships. For example, a bidirectional one-to-many relationship `Consumer` to `ShippingAddress`, with OpenJPA to help explain the performance differences. In this example, a JPA query attempts `select o from Consumer o where . . .` to do a bulk load, and also load all related `ShippingAddress` objects. A one-to-many relationship defined in the `Consumer` class follows:

```
@Entity
public class Consumer implements Serializable {

    @OneToMany(mappedBy="consumer", cascade=CascadeType.ALL, fetch =FetchType.EAGER)
    ArrayList <ShippingAddress> addresses;
```

The many-to-one relation `consumer` defined in the `ShippingAddress` class follows:

```
@Entity
public class ShippingAddress implements Serializable{

    @ManyToOne(fetch=FetchType.EAGER)
    Consumer consumer;
}
```

If the fetch types of both relationships are configured as eager, OpenJPA uses $N+1+1$ queries to get all the `Consumer` objects and `ShippingAddress` objects, where N is the number of `ShippingAddress` objects. However if the `ShippingAddress` is changed to use lazy fetch type as follows, it only takes two queries to get all the data.

```
@Entity
public class ShippingAddress implements Serializable{

    @ManyToOne(fetch=FetchType.LAZY)
    Consumer consumer;
}
```

Although the query returns the same results, having a lower number of queries significantly decreases interaction with the database, which can increase application performance.

JPAEntityLoader plug-in:

The `JPAEntityLoader` plug-in is a built-in Loader implementation that uses Java Persistence API (JPA) to communicate with the database when you are using the `EntityManager` API. When you are using the `ObjectMap` API, use the `JPALoader` loader.

Loader details

Use the JPALoader plug-in when you are storing data using the ObjectMap API. Use the JPAEntityLoader plug-in when you are storing data using the EntityManager API.

Loaders provide two main functions:

1. **get**: In the get method, the JPAEntityLoader plug-in first calls the `javax.persistence.EntityManager.find(Class entityClass, Object key)` method to find the JPA entity. Then the plug-in projects this JPA entity into entity tuples. During the projection, both the tuple attributes and the association keys are stored in the value tuple. After processing each key, the get method returns a list of entity value tuples.
2. **batchUpdate**: The batchUpdate method takes a LogSequence object that contains a list of LogElement objects. Each LogElement object contains a key tuple and a value tuple. To interact with the JPA provider, you must first find the eXtreme Scale entity based on the key tuple. Based on the LogElement type, you run the following JPA calls:
 - **insert**: `javax.persistence.EntityManager.persist(Object o)`
 - **update**: `javax.persistence.EntityManager.merge(Object o)`
 - **remove**: `javax.persistence.EntityManager.remove(Object o)`

A LogElement with type **update** makes the JPAEntityLoader call the `javax.persistence.EntityManager.merge(Object o)` method to merge the entity. However, an **update** type LogElement might be the result of either a `com.ibm.websphere.objectgrid.em.EntityManager.merge(object o)` call or an attribute change of the eXtreme Scale EntityManager managed-instance. See the following example:

```
com.ibm.websphere.objectgrid.em.EntityManager em = og.getSession().getEntityManager();
em.getTransaction().begin();
Consumer c1 = (Consumer) em.find(Consumer.class, c.getConsumerId());
c1.setName("New Name");
em.getTransaction().commit();
```

In this example, an update type LogElement is sent to the JPAEntityLoader of the map consumer. The `javax.persistence.EntityManager.merge(Object o)` method is called to the JPA entity manager instead of an attribute update to the JPA-managed entity. Because of this changed behavior, some limitations exist with using this programming model.

Application design rules

Entities have relationships with other entities. Designing an application with relationships involved and with JPAEntityLoader plugged in requires additional considerations. The application should follow the following four rules, described in the following sections.

Limited relationship depth support

The JPAEntityLoader is only supported when using entities without any relationships or entities with single-level relationships. Relationships with more than one level, such as Company > Department > Employee are not supported.

One loader per map

Using the Consumer-ShippingAddress entity relationships as an example, when you load a consumer with eager fetch enabled, you could load all the related ShippingAddress objects. When you persist or merge a Consumer object, you could persist or merge related ShippingAddress objects if cascade-persist or cascade-merge is enabled.

You cannot plug in a loader for the root entity map which stores the Consumer entity tuples. You must configure a loader for each entity map.

Same cascade type for JPA and eXtreme Scale

Reconsider the scenario where the entity Consumer has a one-to-many relationship with ShippingAddress. You can look at the scenario where cascade-persist is enabled for this relationship. When a Consumer object is persisted into eXtreme Scale, the associated N number of ShippingAddress objects are also persisted into eXtreme Scale.

A persist call of the Consumer object with a cascade-persist relationship to ShippingAddress translates to one `javax.persistence.EntityManager.persist(consumer)` method call and *N* `javax.persistence.EntityManager.persist(shippingAddress)` method calls by the JPAEntityLoader layer. However, these *N* extra persist calls to ShippingAddress objects are unnecessary because of the cascade-persist setting from the JPA provider point of view. To solve this problem, eXtreme Scale provides a new method `isCascaded` on the `LogElement` interface. The `isCascaded` method indicates whether the `LogElement` is a result of an eXtreme Scale `EntityManager` cascade operation. In this example, the JPAEntityLoader of the ShippingAddress map receives *N* `LogElement` objects because of the cascade persist calls. The JPAEntityLoader finds out that the `isCascaded` method returns true and then ignores them without making any JPA calls. Therefore, from a JPA point of view, only one `javax.persistence.EntityManager.persist(consumer)` method call is received.

The same behavior is exhibited if you merge an entity or remove an entity with cascade enabled. The cascaded operations are ignored by the JPAEntityLoader plug-in.

The design of the cascade support is to replay the eXtreme Scale `EntityManager` operations to the JPA providers. These operations include `persist`, `merge`, and `remove` operations. To enable cascade support, verify that the cascade setting for the JPA and the eXtreme Scale `EntityManager` are the same.

Use entity update with caution

As previously described, the design of the cascade support is to replay eXtreme Scale `EntityManager` operations to the JPA providers. If your application calls the `ogEM.persist(consumer)` method to the eXtreme Scale `EntityManager`, even the associated ShippingAddress objects are persisted because of the cascade-persist setting, and the JPAEntityLoader only calls the `jpAEM.persist(consumer)` method to the JPA providers.

However, if your application updates a managed entity, this update translates to a JPA `merge` call by the JPAEntityLoader plug-in. In this scenario, support for multiple levels of relationships and key associations is not guaranteed. In this case,

the best practice is to use the `javax.persistence.EntityManager.merge(o)` method instead of updating a managed entity.

Using a loader with entity maps and tuples

The entity manager converts all entity objects into tuple objects before they are stored in an WebSphere eXtreme Scale map. Every entity has a key tuple and a value tuple. This key-value pair is stored in the associated eXtreme Scale map for the entity. When you are using an eXtreme Scale map with a loader, the loader must interact with the tuple objects.

eXtreme Scale includes loader plug-ins that simplify integration with relational databases. The Java Persistence API (JPA) Loaders use a Java Persistence API to interact with the database and create the entity objects. The JPA loaders are compatible with eXtreme Scale entities.

Tuples

A tuple contains information about the attributes and associations of an entity. Primitive values are stored using their primitive wrappers. Other supported object types are stored in their native format. Associations to other entities are stored as a collection of key tuple objects that represent the keys of the target entities.

Each attribute or association is stored using a zero-based index. You can retrieve the index of each attribute using the `getAttributePosition` or `getAssociationPosition` methods. After the position is retrieved, it remains unchanged for the duration of the eXtreme Scale life cycle. The position can change when the eXtreme Scale is restarted. The `setAttribute`, `setAssociation` and `setAssociations` methods are used to update the elements in the tuple.

Attention: When you are creating or updating tuple objects, update every primitive field with a non-null value. Primitive values such as `int` should not be null. If you do not change the value to a default, poor performance issues can result, also affecting fields marked with the `@Version` annotation or `version` attribute in the entity descriptor XML file.

The following example further explains how to process tuples. For more information about defining entities for this example, see the information about the order entity schema, which is in the entity manager tutorial in the *Product Overview*. WebSphere eXtreme Scale is configured for using loaders with each of the entities. Additionally, only the Order entity is taken, and this specific entity has a many-to-one relationship with the Customer entity. The attribute name is `customer`, and it has a one-to-many relationship with the OrderLine entity.

Use the Projector to build Tuple objects automatically from entities. Using the Projector can simplify loaders when you are using an object-relational mapping utility such as Hibernate or JPA.

order.java

```
@Entity
public class Order
{
    @Id String orderNumber;
    java.util.Date date;
    @OneToOne(cascade=CascadeType.PERSIST) Customer customer;
    @OneToMany(cascade=CascadeType.ALL, mappedBy="order") @OrderBy("lineNumber") List<OrderLine> lines;
}
```


customer.java

```
@Entity
public class Customer {
    @Id String id;
    String firstName;
    String surname;
    String address;
    String phoneNumber;
}
```

orderLine.java

```
@Entity
public class OrderLine
{
    @Id @ManyToOne(cascade=CascadeType.PERSIST) Order order;
    @Id int lineNumber;
    @OneToOne(cascade=CascadeType.PERSIST) Item item;
    int quantity;
    double price;
}
```

A OrderLoader class that implements the Loader interface is shown in the following code. The following example assumes that an associated TransactionCallback plug-in is defined.

orderLoader.java

```
public class OrderLoader implements com.ibm.websphere.objectgrid.plugins.Loader {

    private EntityMetadata entityMetadata;
    public void batchUpdate(TxID txid, LogSequence sequence)
        throws LoaderException, OptimisticCollisionException {
        ...
    }
    public List get(TxID txid, List keyList, boolean forUpdate)
        throws LoaderException {
        ...
    }
    public void preloadMap(Session session, BackingMap backingMap)
        throws LoaderException {
        this.entityMetadata=backingMap.getEntityMetadata();
    }
}
```

The instance variable `entityMetadata` is initialized during the `preloadMap` method call from the eXtreme Scale. The `entityMetadata` variable is not null if the Map is configured to use entities. Otherwise, the value is null.

batchUpdate method

The `batchUpdate` method provides the ability to know what action the application intended to perform. Based on an insert, update or a delete operation, a connection can be opened to the database and the work performed. Because the key and values are of type `Tuple`, they must be transformed so the values make sense on the SQL statement.

The ORDER table was created with the following Data Definition Language (DDL) definition, as shown in the following code:

```
CREATE TABLE ORDER (ORDERNUMBER VARCHAR(250) NOT NULL, DATE TIMESTAMP, CUSTOMER_ID VARCHAR(250))
ALTER TABLE ORDER ADD CONSTRAINT PK_ORDER PRIMARY KEY (ORDERNUMBER)
```

The following code demonstrates how to convert a `Tuple` to an Object:

```
public void batchUpdate(TxID txid, LogSequence sequence)
    throws LoaderException, OptimisticCollisionException {
    Iterator iter = sequence.getPendingChanges();
    while (iter.hasNext()) {
```

```

        LogElement logElement = (LogElement) iter.next();
        Object key = logElement.getKey();
        Object value = logElement.getCurrentValue();

        switch (logElement.getType().getCode()) {
        case LogElement.CODE_INSERT:

1)             if (entityMetaData!=null) {
2)                 // The order has just one key orderNumber
                    String ORDERNUMBER=(String) getKeyAttribute("orderNumber", (Tuple) key);
3)                 // Get the value of date
                    java.util.Date unFormattedDate = (java.util.Date) getValueAttribute("date", (Tuple) value);
4)                 // The values are 2 associations. Lets process customer because
                    // the our table contains customer.id as primary key
                    Object[] keys= getForeignKeyForValueAssociation("customer","id", (Tuple) value);
                    //Order to Customer is M to 1. There can only be 1 key
5)                 String CUSTOMER_ID=(String)keys[0];
6)                 // parse variable unFormattedDate and format it for the database as formattedDate
                    String formattedDate = "2007-05-08-14.01.59.780272"; // formatted for DB2
7)                 // Finally, the following SQL statement to insert the record
                    //INSERT INTO ORDER (ORDERNUMBER, DATE, CUSTOMER_ID) VALUES(ORDERNUMBER,formattedDate, CUSTOMER_ID)
                    }
                    break;
        case LogElement.CODE_UPDATE:
                    break;
        case LogElement.CODE_DELETE:
                    break;
        }
    }
}

// returns the value to attribute as stored in the key Tuple
private Object getKeyAttribute(String attr, Tuple key) {
    //get key metadata
    TupleMetadata keyMD = entityMetaData.getKeyMetadata();
    //get position of the attribute
    int keyAt = keyMD.getAttributePosition(attr);
    if (keyAt > -1) {
        return key.getAttribute(keyAt);
    } else { // attribute undefined
        throw new IllegalArgumentException("Invalid position index for "+attr);
    }
}

// returns the value to attribute as stored in the value Tuple
private Object getValueAttribute(String attr, Tuple value) {
    //similar to above, except we work with value metadata instead
    TupleMetadata valueMD = entityMetaData.getValueMetadata();

    int keyAt = valueMD.getAttributePosition(attr);
    if (keyAt > -1) {
        return value.getAttribute(keyAt);
    } else {
        throw new IllegalArgumentException("Invalid position index for "+attr);
    }
}

// returns an array of keys that refer to association.
private Object[] getForeignKeyForValueAssociation(String attr, String fk_attr, Tuple value) {
    TupleMetadata valueMD = entityMetaData.getValueMetadata();
    Object[] ro;

    int customerAssociation = valueMD.getAssociationPosition(attr);
    TupleAssociation tupleAssociation = valueMD.getAssociation(customerAssociation);

    EntityMetadata targetEntityMetadata = tupleAssociation.getTargetEntityMetadata();

    Tuple[] customerKeyTuple = ((Tuple) value).getAssociations(customerAssociation);

    int numberOfKeys = customerKeyTuple.length;
    ro = new Object[numberOfKeys];

    TupleMetadata keyMD = targetEntityMetadata.getKeyMetadata();
    int keyAt = keyMD.getAttributePosition(fk_attr);
    if (keyAt < 0) {
        throw new IllegalArgumentException("Invalid position index for " + attr);
    }
    for (int i = 0; i < numberOfKeys; ++i) {
        ro[i] = customerKeyTuple[i].getAttribute(keyAt);
    }

    return ro;
}
}

```

1. Ensure that entityMetaData is not null, which implies that the key and value cache entries are of type Tuple. From the entityMetaData, Key TupleMetadata is retrieved, which really reflects only the key part of Order metadata.
2. Process the KeyTuple and get the value of Key Attribute orderNumber

3. Process the ValueTuple and get the value of attribute date
4. Process the ValueTuple and get the value of Keys from association customer
5. Extract CUSTOMER_ID. Based on relationship, an Order can only have one customer, we will only have one key. Hence the size of keys is 1. For simplicity, we skipped parsing of date to correct format.
6. Because this is an insert operation, the SQL statement is passed onto the data source connection to complete the insert operation.

Transaction demarcation and access to database is covered in “Writing a loader” on page 321.

get method

If the key is not found in the cache, call the get method in the Loader plug-in to find the key.

The key is a Tuple. The first step is to convert the Tuple to primitive values that can be passed onto the SELECT SQL statement. After all the attributes are retrieved from the database, you must convert into Tuples. The following code demonstrates the Order class.

```
public List get(TxID txid, List keyList, boolean forUpdate) throws LoaderException {
    System.out.println("OrderLoader: Get called");
    ArrayList returnList = new ArrayList();

1)  if (entityMetaData != null) {
        int index=0;
        for (Iterator iter = keyList.iterator(); iter.hasNext();) {
2)      Tuple orderKeyTuple=(Tuple) iter.next();

            // The order has just one key orderNumber
3)      String ORDERNUMBERKEY = (String) getKeyAttribute("orderNumber",orderKeyTuple);
            //We need to run a query to get values of
4)      // SELECT CUSTOMER_ID, date FROM ORDER WHERE ORDERNUMBER='ORDERNUMBERKEY'

5)      //1) Foreign key: CUSTOMER_ID
6)      //2) date
            // Assuming those two are returned as
7)      String CUSTOMER_ID = "C001"; // Assuming Retrieved and initialized
8)      java.util.Date retrievedDate = new java.util.Date();
            // Assuming this date reflects the one in database

            // We now need to convert this data into a tuple before returning

            //create a value tuple
9)      TupleMetadata valueMD = entityMetaData.getValueMetadata();
            Tuple valueTuple=valueMD.createTuple();

            //add retrievedDate object to Tuple
            int datePosition = valueMD.getAttributePosition("date");
10)     valueTuple.setAttribute(datePosition, retrievedDate);

            //Next need to add the Association
11)     int customerPosition=valueMD.getAssociationPosition("customer");
            TupleAssociation customerTupleAssociation =
                valueMD.getAssociation(customerPosition);
            EntityMetadata customerEMD = customerTupleAssociation.getTargetEntityMetadata();
            TupleMetadata customerTupleMDForKEY=customerEMD.getKeyMetadata();
12)     int customerKeyAt=customerTupleMDForKEY.getAttributePosition("id");

            Tuple customerKeyTuple=customerTupleMDForKEY.createTuple();
            customerKeyTuple.setAttribute(customerKeyAt, CUSTOMER_ID);
13)     valueTuple.addAssociationKeys(customerPosition, new Tuple[] {customerKeyTuple});

14)     int linesPosition = valueMD.getAssociationPosition("lines");
            TupleAssociation linesTupleAssociation = valueMD.getAssociation(linesPosition);
            EntityMetadata orderLineEMD = linesTupleAssociation.getTargetEntityMetadata();
            TupleMetadata orderLineTupleMDForKEY = orderLineEMD.getKeyMetadata();
            int lineNumberAt = orderLineTupleMDForKEY.getAttributePosition("lineNumber");
            int orderAt = orderLineTupleMDForKEY.getAssociationPosition("order");

            if (lineNumberAt < 0 || orderAt < 0) {
                throw new IllegalArgumentException(
                    "Invalid position index for lineNumber or order "+
                    lineNumberAt + " " + orderAt);
            }
        }
    }
}
```

```

    }
15) // SELECT LINENUMBER FROM ORDERLINE WHERE ORDERNUMBER='ORDERNUMBERKEY'
    // Assuming two rows of line number are returned with values 1 and 2

    Tuple orderLineKeyTuple1 = orderLineTupleMDForKEY.createTuple();
    orderLineKeyTuple1.setAttribute(lineNumberAt, new Integer(1)); // set Key
    orderLineKeyTuple1.addAssociationKey(orderAt, orderKeyTuple);

    Tuple orderLineKeyTuple2 = orderLineTupleMDForKEY.createTuple();
    orderLineKeyTuple2.setAttribute(lineNumberAt, new Integer(2)); // Init Key
    orderLineKeyTuple2.addAssociationKey(orderAt, orderKeyTuple);

16) valueTuple.addAssociationKeys(linesPosition, new Tuple[]
    {orderLineKeyTuple1, orderLineKeyTuple2 });

    returnList.add(index, valueTuple);

    index++;

}
} else {
// does not support tuples
}
return returnList;
}

```

1. The get method is called when the ObjectGrid cache could not find the key and requests the loader to fetch. Test for entityMetaData value and proceed if not null.
2. The keyList contains Tuples.
3. Retrieve value of attribute orderNumber.
4. Run query to retrieve date (value) and customer ID (foreign key).
5. CUSTOMER_ID is a foreign key that must be set in the association tuple.
6. The date is a value and should already be set.
7. Since this example does not perform JDBC calls, CUSTOMER_ID is assumed.
8. Since this example does not perform JDBC calls, date is assumed.
9. Create value Tuple.
10. Set the value of date into the Tuple, based on its position.
11. Order has two associations. Start with the attribute customer which refers to the customer entity. You must have the value of ID to set in the Tuple.
12. Find the position of ID on the customer entity.
13. Set the values of the association keys only.
14. Also, lines is an association that must be set up as a group of association keys, in the same way as you did for customer association.
15. Since you must set up keys for the lineNumber associated with this order, run the SQL to retrieve lineNumber values.
16. Set up the association keys in the valueTuple. This completes the creation of the Tuple that is returned to the BackingMap.

This topic offers the steps create tuples, and a description of the Order entity only. Complete similar steps for the other entities, and the entire process that is tied together with the TransactionCallback plug-in. See “Plug-ins for managing transaction life cycle events” on page 351 for details.

Writing a loader with a replica preload controller

A Loader with a replica preload controller is a Loader that implements the ReplicaPreloadController interface in addition to the Loader interface.

The ReplicaPreloadController interface is designed to provide a way for a replica that becomes the primary shard to know whether the previous primary shard has completed the preload process. If the preload is partially completed, the information to pick up where the previous primary left is provided. With the

ReplicaPreloadController interface implementation, a replica that becomes the primary continues the preload process where the previous primary left and continues to finish the overall preload.

In a distributed WebSphere eXtreme Scale environment, a map can have replicas and might preload large volume of data during initialization. The preload is a Loader activity and only occurs in the primary map during initialization. The preload might take a long time to complete if a large volume of data is preloaded. If the primary map has completed large portion of preload data, but is stopped for unknown reason during initialization, a replica becomes the primary. In this situation, the preloaded data that was completed by the previous primary is lost because the new primary normally performs an unconditional preload. With an unconditional preload, the new primary starts the preload process from the beginning and the previous preloaded data is ignored. If you want the new primary to pick up where the previous primary left during preload process, provide a Loader that implements the ReplicaPreloadController interface. For more information see the API documentation.

For information about Loaders, see “Loaders” on page 97 the information about loaders in the *Product Overview*. If you are interested in writing a regular Loader plug-in, see “Writing a loader” on page 321.

The ReplicaPreloadController interface has the following definition:

```
public interface ReplicaPreloadController
{
    public static final class Status
    {
        static public final Status PRELOADED_ALREADY =
            new Status(K_PRELOADED_ALREADY);
        static public final Status FULL_PRELOAD_NEEDED =
            new Status(K_FULL_PRELOAD_NEEDED);
        static public final Status PARTIAL_PRELOAD_NEEDED =
            new Status(K_PARTIAL_PRELOAD_NEEDED);
    }

    Status checkPreloadStatus(Session session,
        BackingMap bmap);
}
```

The following sections discuss some of the methods of the Loader and ReplicaPreloadController interface.

checkPreloadStatus method

When a Loader implements ReplicaPreloadController interface, the checkPreloadStatus method is called before the preloadMap method during map initialization. The return status of this method determines if the preloadMap method is called. If this method returns Status#PRELOADED_ALREADY, the preload method is not called. Otherwise, the preload method runs. Because of this behavior, this method should serve as the Loader initialization method. You must initialize the Loader properties in this method. This method must return the correct status, or the preload might not work as expected.

```
public Status checkPreloadStatus(Session session,
    BackingMap backingMap) {
    // When a loader implements ReplicaPreloadController interface,
    // this method will be called before preloadMap method during
    // map initialization. Whether the preloadMap method will be
    // called depends on the returned status of this method. So, this
    // method also serve as Loader's initialization method. This method
    // has to return the right status, otherwise the preload may not
    // work as expected.

    // Note: must initialize this loader instance here.
```

```

ivOptimisticCallback = backingMap.getOptimisticCallback();
ivBackingMapName = backingMap.getName();
ivPartitionId = backingMap.getPartitionId();
ivPartitionManager = backingMap.getPartitionManager();
ivTransformer = backingMap.getObjectTransformer();
preloadStatusKey = ivBackingMapName + "_" + ivPartitionId;

try {
    // get the preloadStatusMap to retrieve preload status that
    // could be set by other JVMs.
    ObjectMap preloadStatusMap = session.getMap(ivPreloadStatusMapName);

    // retrieve last recorded preload data chunk index.
    Integer lastPreloadedDataChunk = (Integer) preloadStatusMap
        .get(preloadStatusKey);

    if (lastPreloadedDataChunk == null) {
        preloadStatus = Status.FULL_PRELOAD_NEEDED;
    } else {
        preloadedLastDataChunkIndex = lastPreloadedDataChunk.intValue();
        if (preloadedLastDataChunkIndex == preloadCompleteMark) {
            preloadStatus = Status.PRELOADED_ALREADY;
        } else {
            preloadStatus = Status.PARTIAL_PRELOAD_NEEDED;
        }
    }

    System.out.println("TupleHeapCacheWithReplicaPreloadControllerLoader.
checkPreloadStatus()
-> map = " + ivBackingMapName + ", preloadStatusKey = " + preloadStatusKey
    + ", retrieved lastPreloadedDataChunk = " + lastPreloadedDataChunk + ",
    determined preloadStatus = "
    + getStatusString(preloadStatus));

} catch (Throwable t) {
    t.printStackTrace();
}

return preloadStatus;
}

```

preloadMap method

Running the preloadMap method depends on the returned result from checkPreloadStatus method. If the preloadMap method is called, it typically must retrieve preload status information from designated preload status map and determine how to proceed. Ideally, the preloadMap method should know if the preload is partially complete and exactly where to start. During the data preload, the preloadMap method should update the preload status on the designated preload status map. The preload status that is stored in the preload status map is retrieved by the checkPreloadStatus method when it needs to check the preload status.

```

public void preloadMap(Session session, BackingMap backingMap)
throws LoaderException {
    EntityMetadata emd = backingMap.getEntityMetadata();
    if (emd != null && tupleHeapPreloadData != null) {
        // The getPreLoadData method is similar to fetching data
        // from database. These data will be push into cache as
        // preload process.
        ivPreloadData = tupleHeapPreloadData.getPreLoadData(emd);

        ivOptimisticCallback = backingMap.getOptimisticCallback();
        ivBackingMapName = backingMap.getName();
        ivPartitionId = backingMap.getPartitionId();
        ivPartitionManager = backingMap.getPartitionManager();
        ivTransformer = backingMap.getObjectTransformer();
        Map preloadMap;

        if (ivPreloadData != null) {
            try {
                ObjectMap map = session.getMap(ivBackingMapName);

                // get the preloadStatusMap to record preload status.
                ObjectMap preloadStatusMap = session.

```

```

getMap(ivPreloadStatusMapName);

    // Note: when this preloadMap method is invoked, the
// checkPreloadStatus has been called, Both preloadStatus
// and preloadedLastDataChunkIndex have been set. And the
// preloadStatus must be either PARTIAL_PRELOAD_NEEDED
// or FULL_PRELOAD_NEEDED that will require a preload again.

    // If large amount of data will be preloaded, the data usually
// is divided into few chunks and the preload process will
// process each chunk within its own tran. This sample only
// preload few entries and assuming each entry represent a chunk.
    // so that the preload process an entry in a tran to simulate
// chunk preloading.

    Set entrySet = ivPreloadData.entrySet();
    preloadMap = new HashMap();
    ivMap = preloadMap;

    // The dataChunkIndex represent the data chunk that is in
// processing
    int dataChunkIndex = -1;
    boolean shouldRecordPreloadStatus = false;
    int numberOfDataChunk = entrySet.size();
    System.out.println("    numberOfDataChunk to be preloaded = "
+ numberOfDataChunk);

    Iterator it = entrySet.iterator();
    int whileCounter = 0;
    while (it.hasNext()) {
        whileCounter++;
        System.out.println("preloadStatusKey = " + preloadStatusKey
+ " ,
whileCounter = " + whileCounter);

        dataChunkIndex++;

        // if the current dataChunkIndex <= preloadedLastDataChunkIndex
// no need to process, because it has been preloaded by
// other JVM before. only need to process dataChunkIndex
// > preloadedLastDataChunkIndex
        if (dataChunkIndex <= preloadedLastDataChunkIndex) {
            System.out.println("ignore current dataChunkIndex = "
+ dataChunkIndex + " that has been previously
preloaded.");
            continue;
        }

        // Note: This sample simulate data chunk as an entry.
        // each key represent a data chunk for simplicity.
        // If the primary server or shard stopped for unknown
// reason, the preload status that indicates the progress
// of preload should be available in preloadStatusMap. A
// replica that become a primary can get the preload status
// and determine how to preload again.
        // Note: recording preload status should be in the same
// tran as putting data into cache; so that if tran
// rollback or error, the recorded preload status is the
// actual status.

        Map.Entry entry = (Entry) it.next();
        Object key = entry.getKey();
        Object value = entry.getValue();
        boolean tranActive = false;

        System.out.println("processing data chunk. map = " +
this.ivBackingMapName + " , current dataChunkIndex = " +
dataChunkIndex + " , key = " + key);

        try {
            shouldRecordPreloadStatus = false; // re-set to false
            session.beginNoWriteThrough();
            tranActive = true;

            if (ivPartitionManager.getNumOfPartitions() == 1) {
                // if just only 1 partition, no need to deal with
// partition.
                // just push data into cache
                map.put(key, value);
                preloadMap.put(key, value);
            }
        }
    }
}

```

```

        shouldRecordPreloadStatus = true;
    } else if (ivPartitionManager.getPartition(key) ==
ivPartitionId) {
        // if map is partitioned, need to consider the
// partition key only preload data that belongs
// to this partition.
        map.put(key, value);
        preloadMap.put(key, value);
        shouldRecordPreloadStatus = true;
    } else {
        // ignore this entry, because it does not belong to
// this partition.
    }

    if (shouldRecordPreloadStatus) {
        System.out.println("record preload status. map = " +
this.ivBackingMapName + ", preloadStatusKey = " +
preloadStatusKey + ", current dataChunkIndex = "
+ dataChunkIndex);
        if (dataChunkIndex == numberOfDataChunk) {
            System.out.println("record preload status. map = " +
this.ivBackingMapName + ", preloadStatusKey = " +
preloadStatusKey + ", mark complete = " +
preloadCompleteMark);
            // means we are at the lastest data chunk, if commit
// successfully, record preload complete.
// at this point, the preload is considered to be done
            // use -99 as special mark for preload complete status.

            preloadStatusMap.get(preloadStatusKey);

            // a put follow a get will become update if the get
// return an object, otherwise, it will be insert.
            preloadStatusMap.put(preloadStatusKey, new
Integer(preloadCompleteMark));

        } else {
            // record preloaded current dataChunkIndex into
// preloadStatusMap a put follow a get will become
// update if teh get return an object, otherwise, it
// will be insert.
            preloadStatusMap.get(preloadStatusKey);
            preloadStatusMap.put(preloadStatusKey, new
Integer(dataChunkIndex));
        }
    }

    session.commit();
    tranActive = false;

    // to simulate preloading large amount of data
    // put this thread into sleep for 30 secs.
    // The real app should NOT put this thread to sleep
    Thread.sleep(10000);

} catch (Throwable e) {
    e.printStackTrace();
    throw new LoaderException("preload failed with
exception: " + e, e);
} finally {
    if (tranActive && session != null) {
        try {
            session.rollback();
        } catch (Throwable e1) {
            // preload ignoring exception from rollback
        }
    }
}

// at this point, the preload is considered to be done for sure
// use -99 as special mark for preload complete status.
// this is a insurance to make sure the complete mark is set.
// besides, when partitioning, each partition does not know when
// is its last data chunk. so the following block serves as the
// overall preload status complete reporting.
System.out.println("Overall preload status complete -> record
preload status. map = " + this.ivBackingMapName + ",
preloadStatusKey = " + preloadStatusKey + ", mark complete = " +
preloadCompleteMark);

```


- The loader knows how to perform the optimistic comparison without assistance from an OptimisticCallback plug-in.

How can the loader know how to deal with optimistic versioning without assistance from an OptimisticCallback object? The loader has knowledge of the value class object and knows which field of the value object is used as an optimistic versioning value. For example, suppose the following interface is used for the value object for the employees map:

```
public interface Employee
{
    // Sequential sequence number used for optimistic versioning.
    public long getSequenceNumber();
    public void setSequenceNumber(long newSequenceNumber);
    // Other get/set methods for other fields of Employee object.
}
```

In this case, the loader knows that it can use the `getSequenceNumber` method to get the current version information for an `Employee` value object. The loader increments the returned value to generate a new version number before updating the persistent storage with the new `Employee` value. For a Java database connectivity (JDBC) loader, the current sequence number in the `where` clause of an overqualified SQL update statement is used, and it uses the new generated sequence number to set the sequence number column to the new sequence number value.

Another possibility is that the loader makes use of some backend-provided function that automatically updates a hidden column that can be used for optimistic versioning. In some cases, a stored procedure or trigger can possibly be used to help maintain a column that holds versioning information. If the loader is using one of these techniques for maintaining optimistic versioning information, then the application does not need to provide an `OptimisticCallback` implementation. You can use the default `OptimisticCallback` implementation because the loader is able to handle optimistic versioning without any assistance from an `OptimisticCallback` object.

Default implementation for entities

Entities are stored in the `ObjectGrid` using tuple objects. The default `OptimisticCallback` implementation behaves similarly to the behavior for non-entity maps. However, the version field in the entity is identified using the `@Version` annotation or the version attribute in the entity descriptor XML file.

The version attribute can be of the following types: `int`, `Integer`, `short`, `Short`, `long`, `Long` or `java.sql.Timestamp`. An entity should have only one version attribute defined. The version attribute should be set only during construction. After the entity is persisted, the value of the version attribute should not be modified.

If a version attribute is not configured and the optimistic locking strategy is used, then the entire tuple is implicitly versioned using the entire state of the tuple.

In the following example, the `Employee` entity has a long version attribute named `SequenceNumber`:

```
@Entity
public class Employee
{
    private long sequence;
    // Sequential sequence number used for optimistic versioning.
    @Version
```

```

    public long getSequenceNumber() {
        return sequence;
    }
    public void setSequenceNumber(long newSequenceNumber) {
        this.sequence = newSequenceNumber;
    }
    // Other get/set methods for other fields of Employee object.
}

```

Writing an OptimisticCallback implementation

An OptimisticCallback implementation must implement the OptimisticCallback interface and follow the common ObjectGrid plug-in conventions

The following list provides a description or consideration for each of the methods in the OptimisticCallback interface:

NULL_OPTIMISTIC_VERSION

This special value is returned by getVersionedObjectForValue method if the default OptimisticCallback implementation is used instead of an application-provided OptimisticCallback implementation.

getVersionedObjectForValue method

The getVersionedObjectForValue method might return a copy of the value or it might return an attribute of the value that can be used for versioning purposes. This method is called whenever an object is associated with a transaction. When no Loader is set into a backing map, the backing map uses this value at commit time to perform an optimistic version comparison. The optimistic version comparison is used by the backing map to ensure that the version has not changed since this transaction first accessed the map entry that was modified by this transaction. If another transaction had already modified the version for this map entry, the version comparison fails and the backing map displays an OptimisticCollisionException exception to force rollback of the transaction. If a Loader is plugged in, the backing map does not use the optimistic versioning information. Instead, the Loader is responsible for performing the optimistic versioning comparison and updating the versioning information when necessary. The Loader typically gets the initial versioning object from the LogElement passed to the batchUpdate method on the Loader, which is called when a flush operation occurs or a transaction is committed.

The following code shows the implementation used by the EmployeeOptimisticCallbackImpl object:

```

public Object getVersionedObjectForValue(Object value)
{
    if (value == null)
    {
        return null;
    }
    else
    {
        Employee emp = (Employee) value;
        return new Long( emp.getSequenceNumber() );
    }
}

```

As demonstrated in the previous example, the sequenceNumber attribute is returned in a java.lang.Long object as expected by the Loader, which implies that

the same person that wrote the Loader either wrote the `EmployeeOptimisticCallbackImpl` implementation or worked closely with the person that implemented the `EmployeeOptimisticCallbackImpl` implementation. For example, these people agreed on the value that is returned by the `getVersionedObjectForValue` method. As previously described, the default `OptimisticCallback` implementation returns the special value `NULL_OPTIMISTIC_VERSION` as the version object.

updateVersionedObjectForValue method

The `updateVersionedObjectForValue` method is called when a transaction has updated a value and a new versioned object is needed. If the `getVersionedObjectForValue` method returns an attribute of the value, this method typically updates the attribute value with a new version object. If the `getVersionedObjectForValue` method returns a copy of the value, this method typically would not update. The default `OptimisticCallback` does not update because the default implementation of the `getVersionedObjectForValue` method always returns the special value `NULL_OPTIMISTIC_VERSION` as the version object. The following example shows the implementation used by the `EmployeeOptimisticCallbackImpl` object that is used in the `OptimisticCallback` section:

```
public void updateVersionedObjectForValue(Object value)
{
    if ( value != null )
    {
        Employee emp = (Employee) value;
        long next = emp.getSequenceNumber() + 1;
        emp.updateSequenceNumber( next );
    }
}
```

As demonstrated in the previous example, the `sequenceNumber` attribute is incremented by one so that the next time the `getVersionedObjectForValue` method is called, the `java.lang.Long` value that is returned has a long value that is the original sequence number value. Plus one, for example, is the next version value for this employee instance. Again, this example implies that the same person that wrote the Loader either wrote `EmployeeOptimisticCallbackImpl` implementation or worked closely with the person that implemented the `EmployeeOptimisticCallbackImpl` implementation.

serializeVersionedValue method

This method writes the versioned value to the specified stream. Depending on the implementation, the versioned value can be used to identify optimistic update collisions. In some implementations, the versioned value is a copy of the original value. Other implementations might have a sequence number or some other object to indicate the version of the value. Because the actual implementation is unknown, this method is provided to perform the proper serialization. The default implementation calls the `writeObject` method.

inflateVersionedValue method

This method takes the serialized version of the versioned value and returns the actual versioned value object. Depending on the implementation, the versioned value can be used to identify optimistic update collisions. In some implementations, the versioned value is a copy of the original value. Other implementations might have a sequence number or some other object to indicate

the version of the value. Because the actual implementation is unknown, this method is provided to perform the proper deserialization. The default implementation calls the readObject method.

Using an application-provided OptimisticCallback implementation

You have two approaches to add an application-provided OptimisticCallback into the BackingMap configuration: programmatic configuration and XML configuration.

Programmatically plug in an OptimisticCallback implementation

The following example demonstrates how an application can programmatically plug in an OptimisticCallback object for the employee backing map in the grid1 ObjectGrid instance:

```
import com.ibm.websphere.objectgrid.ObjectGridManagerFactory;
import com.ibm.websphere.objectgrid.ObjectGridManager;
import com.ibm.websphere.objectgrid.ObjectGrid;
import com.ibm.websphere.objectgrid.BackingMap;
ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid og = ogManager.createObjectGrid( "grid1" );
BackingMap bm = dg.defineMap("employees");
EmployeeOptimisticCallbackImpl cb = new EmployeeOptimisticCallbackImpl();
bm.setOptimisticCallback( cb );
```

XML configuration approach to plug in an OptimisticCallback implementation

The EmployeeOptimisticCallbackImpl object in the preceding example must implement the OptimisticCallback interface. The application can also use an XML file to plug in its OptimisticCallback object as shown in the following example:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
  <objectGrid name="grid1">
    <backingMap name="employees" pluginCollectionRef="employees" lockStrategy="OPTIMISTIC" />
  </objectGrid>
</objectGrids>

<backingMapPluginCollections>
  <backingMapPluginCollection id="employees">
    <bean id="OptimisticCallback" className="com.xyz.EmployeeOptimisticCallbackImpl" />
  </backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>
```

Transaction processing overview

WebSphere eXtreme Scale uses transactions as its mechanism for interaction with data.

To interact with data, the thread in your application needs its own session. When the application wants to use the ObjectGrid on a thread, call one of the ObjectGrid.getSession methods to obtain a session. With the session, the application can work with data that is stored in the ObjectGrid maps.

When an application uses a Session object, the session must be in the context of a transaction. A transaction begins and commits or begins and rolls back using the begin, commit, and rollback methods on the Session object. Applications can also work in auto-commit mode, in which the Session automatically begins and commits a transaction whenever an operation is performed on the map. Auto-commit mode cannot group multiple operations into a single transaction, so

it is the slower option if you are creating a batch of multiple operations into a single transaction. However, for transactions that contain only one operation, auto-commit is the faster option.

When your application is finished with the Session, use the optional `Session.close()` method to close the session. Closing the Session releases it from the heap and allows subsequent calls to the `getSession()` method to be reused, improving performance.

Introduction to plug-in slots

A plug-in slot is a transactional storage space that is reserved for plug-ins that share transactional context. These slots provide a way for eXtreme Scale plug-ins to communicate with each other, share transactional context, and ensure that transactional resources are used correctly and consistently within a transaction.

A plug-in can store transactional context, such as database connection, Java Message Service (JMS) connection, and so on, in a plug-in slot. The stored transactional context can be retrieved by any plug-in that knows the plug-in slot number, which serves as the key to retrieve transactional context.

Using plug-in slots

Plug-in slots are part of the TxID Interface. See the API documentation for more information about the interface. The slots are entries on an `ArrayList` array. Plug-ins can reserve an entry in the `ArrayList` array by calling the `ObjectGrid.reserveSlot` method and indicating that it wants a slot on all TxID objects. After the slots are reserved, plug-ins can put transactional context into slots of each TxID object and retrieve the context later. The put and get operations are coordinated by slot numbers that are returned by the `ObjectGrid.reserveSlot` method.

A plug-in typically has a life cycle. Using plug-in slots has to fit into the life cycle of plug-in. Typically, the plug-in must reserve plug-in slots during the initialization stage and obtain slot numbers for each slot. During normal run time, the plug-in puts transactional context into the reserved slot in the TxID object at the appropriate point. This appropriate point is typically at the beginning of the transaction. The plug-in or other plug-ins can then get the stored transactional context by the slot number from the TxID within the transaction.

The plug-in typically performs a clean up by removing transactional context and the slots. The following snippet of code illustrates how to use plug-in slots in a `TransactionCallback` plug-in:

```
public class DatabaseTransactionCallback implements TransactionCallback {
    int connectionSlot;
    int autoCommitConnectionSlot;
    int psCacheSlot;
    Properties ivProperties = new Properties();

    public void initialize(ObjectGrid objectGrid) throws TransactionCallbackException {
        // In initialization stage, reserve desired plug-in slots by calling the
        // reserveSlot method of ObjectGrid and
        // passing in the designated slot name, TxID.SLOT_NAME.
        // Note: you have to pass in this TxID.SLOT_NAME that is designated
        // for application.
        try {
            // cache the returned slot numbers
            connectionSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
            psCacheSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
            autoCommitConnectionSlot = objectGrid.reserveSlot(TxID.SLOT_NAME);
        } catch (Exception e) {
        }
    }

    public void begin(TxID tx) throws TransactionCallbackException {
        // put transactional contexts into the reserved slots at the
        // beginning of the transaction.
    }
}
```

```

    try {
        Connection conn = null;
        conn = DriverManager.getConnection(ivDriverUrl, ivProperties);
        tx.putSlot(connectionSlot, conn);
        conn = DriverManager.getConnection(ivDriverUrl, ivProperties);
        conn.setAutoCommit(true);
        tx.putSlot(autoCommitConnectionSlot, conn);
        tx.putSlot(psCacheSlot, new HashMap());
    } catch (SQLException e) {
        SQLException ex = getLastSQLException(e);
        throw new TransactionCallbackException("unable to get connection", ex);
    }
}

public void commit(TxID id) throws TransactionCallbackException {
    // get the stored transactional contexts and use them
    // then, clean up all transactional resources.
    try {
        Connection conn = (Connection) id.getSlot(connectionSlot);
        conn.commit();
        cleanUpSlots(id);
    } catch (SQLException e) {
        SQLException ex = getLastSQLException(e);
        throw new TransactionCallbackException("commit failure", ex);
    }
}

void cleanUpSlots(TxID tx) throws TransactionCallbackException {
    closePreparedStatements((Map) tx.getSlot(psCacheSlot));
    closeConnection((Connection) tx.getSlot(connectionSlot));
    closeConnection((Connection) tx.getSlot(autoCommitConnectionSlot));
}

/**
 * @param map
 */
private void closePreparedStatements(Map psCache) {
    try {
        Collection statements = psCache.values();
        Iterator iter = statements.iterator();
        while (iter.hasNext()) {
            PreparedStatement stmt = (PreparedStatement) iter.next();
            stmt.close();
        }
    } catch (Throwable e) {
    }
}

/**
 * Close connection and swallow any Throwable that occurs.
 */
private void closeConnection(Connection connection) {
    try {
        connection.close();
    } catch (Throwable e1) {
    }
}

public void rollback(TxID id) throws TransactionCallbackException
    // get the stored transactional contexts and use them
    // then, clean up all transactional resources.
    try {
        Connection conn = (Connection) id.getSlot(connectionSlot);
        conn.rollback();
        cleanUpSlots(id);
    } catch (SQLException e) {
    }
}

public boolean isExternalTransactionActive(Session session) {
    return false;
}

// Getter methods for the slot numbers, other plug-in can obtain the slot numbers
// from these getter methods.

public int getConnectionSlot() {
    return connectionSlot;
}

public int getAutoCommitConnectionSlot() {
    return autoCommitConnectionSlot;
}

public int getPreparedStatementSlot() {
    return psCacheSlot;
}
}

```

The following snippet of code illustrates how a Loader can get the stored transactional context that is put by previous TransactionCallback plug-in example:

```
public class DatabaseLoader implements Loader
{
    DatabaseTransactionCallback tcb;
    public void preloadMap(Session session, BackingMap backingMap) throws LoaderException
    {
        // The preload method is the initialization method of the Loader.
        // Obtain interested plug-in from Session or ObjectGrid instance.
        tcb =
        (DatabaseTransactionCallback)session.getObjectGrid().getTransactionCallback();
    }
    public List get(TxID txid, List keyList, boolean forUpdate) throws LoaderException
    {
        // get the stored transactional contexts that is put by tcb's begin method.
        Connection conn = (Connection)txid.getSlot(tcb.getConnectionSlot());
        // implement get here
        return null;
    }
    public void batchUpdate(TxID txid, LogSequence sequence) throws LoaderException,
    OptimisticCollisionException
    {
        // get the stored transactional contexts that is put by tcb's begin method.
        Connection conn = (Connection)txid.getSlot(tcb.getConnectionSlot());
        // implement batch update here ...
    }
}
```

External transaction managers

Typically, eXtreme Scale transactions begin with the Session.begin method and end with the Session.commit method. However, when an ObjectGrid is embedded, an external transaction coordinator can start and end transactions. In this case, you do not need to call the begin or commit methods.

External transaction coordination

The TransactionCallback plug-in is extended with the isExternalTransactionActive(Session session) method that associates the eXtreme Scale session with an external transaction. The method header follows:

```
public synchronized boolean isExternalTransactionActive(Session session)
```

For example, eXtreme Scale can be set up to integrate with WebSphere Application Server and WebSphere Extended Deployment.

Also, eXtreme Scale provides a built in plug-in called the WebSphere “Plug-ins for managing transaction life cycle events” on page 351, which describes how to build the plug-in for WebSphere Application Server environments, but you can adapt the plug-in for other frameworks.

The key to this seamless integration is the exploitation of the ExtendedJTATransaction API in WebSphere Application Server Version 5.x and Version 6.x. However, if you are using WebSphere Application Server Version 6.0.2, you must apply APAR PK07848 to support this method. Use the following sample code to associate an ObjectGrid session with a WebSphere Application Server transaction ID:

```
/**
 * This method is required to associate an objectGrid session with a WebSphere
 * Application Server transaction ID.
 */
Map/**/ localIdToSession;
public synchronized boolean isExternalTransactionActive(Session session)
{
    // remember that this localid means this session is saved for later.
    localIdToSession.put(new Integer(jta.getLocalId()), session);
    return true;
}
```


Retrieve an external transaction

Sometimes you might need to retrieve an external transaction service object for the TransactionCallback plug-in to use. In the WebSphere Application Server server, look up the ExtendedJTATransaction object from its namespace as shown in the following example:

```
public J2EETransactionCallback() {
    super();
    localIdToSession = new HashMap();
    String lookupName="java:comp/websphere/ExtendedJTATransaction";
    try
    {
        InitialContext ic = new InitialContext();
        jta = (ExtendedJTATransaction)ic.lookup(lookupName);
        jta.registerSynchronizationCallback(this);
    }
    catch(NotSupportedException e)
    {
        throw new RuntimeException("Cannot register jta callback", e);
    }
    catch(NamingException e){
        throw new RuntimeException("Cannot get transaction object");
    }
}
```

For other products, you can use a similar approach to retrieve the transaction service object.

Control commit by external callback

The TransactionCallback plug-in must receive an external signal to commit or roll back the eXtreme Scale session. To receive this external signal, use the callback from the external transaction service. Implement the external callback interface and register it with the external transaction service. For example, with WebSphere Application Server, implement the SynchronizationCallback interface, as shown in the following example:

```
public class J2EETransactionCallback implements
com.ibm.websphere.objectgrid.plugins.TransactionCallback, SynchronizationCallback {
    public J2EETransactionCallback() {
        super();
        String lookupName="java:comp/websphere/ExtendedJTATransaction";
        localIdToSession = new HashMap();
        try {
            InitialContext ic = new InitialContext();
            jta = (ExtendedJTATransaction)ic.lookup(lookupName);
            jta.registerSynchronizationCallback(this);
        } catch(NotSupportedException e) {
            throw new RuntimeException("Cannot register jta callback", e);
        }
        catch(NamingException e) {
            throw new RuntimeException("Cannot get transaction object");
        }
    }

    public synchronized void afterCompletion(int localId, byte[] arg1,boolean didCommit) {
        Integer lid = new Integer(localId);
        // find the Session for the localId
        Session session = (Session)localIdToSession.get(lid);
        if(session != null) {
            try {
                // if WebSphere Application Server is committed when
                // hardening the transaction to backingMap.
                // We already did a flush in beforeCompletion
                if(didCommit) {
                    session.commit();
                } else {
                    // otherwise rollback
                    session.rollback();
                }
            }
            catch(NoActiveTransactionException e) {
                // impossible in theory
            }
            catch(TransactionException e) {
            }
        }
    }
}
```

```

        // given that we already did a flush, this should not fail
    } finally {
        // always clear the session from the mapping map.
        localIdToSession.remove(lid);
    }
}

public synchronized void beforeCompletion(int localId, byte[] arg1) {
    Session session = (Session)localIdToSession.get(new Integer(localId));
    if(session != null) {
        try {
            session.flush();
        } catch(TransactionException e) {
            // WebSphere Application Server does not formally define
            // a way to signal the
            // transaction has failed so do this
            throw new RuntimeException("Cache flush failed", e);
        }
    }
}
}
}
}

```

Use eXtreme Scale APIs with the TransactionCallback plug-in

The TransactionCallback plug-in disables autocommit in eXtreme Scale. The normal usage pattern for an eXtreme Scale follows:

```

Session ogSession = ...;
ObjectMap myMap = ogSession.getMap("MyMap");
ogSession.begin();
MyObject v = myMap.get("key");
v.setAttribute("newValue");
myMap.update("key", v);
ogSession.commit();

```

When this TransactionCallback plug-in is in use, eXtreme Scale assumes that the application uses the eXtreme Scale when a container-managed transaction is present. The previous code snippet changes the following code in this environment:

```

public void myMethod() {
    UserTransaction tx = ...;
    tx.begin();
    Session ogSession = ...;
    ObjectMap myMap = ogSession.getMap("MyMap");
    MyObject v = myMap.get("key");
    v.setAttribute("newValue");
    myMap.update("key", v);
    tx.commit();
}

```

The myMethod method is similar to a Web application scenario. The application uses the normal UserTransaction interface to begin, commit, and roll back transactions. The eXtreme Scale automatically begins and commits around the container transaction. If the method is an Enterprise JavaBeans (EJB) method that uses the TX_REQUIRES attribute, then remove the UserTransaction reference and the calls to begin and commit transactions and the method works the same way. In this case, the container is responsible for starting and ending the transaction.

WebSphereTransactionCallback plug-in

When you use the WebSphereTransactionCallback plug-in, enterprise applications that are running in a WebSphere Application Server environment can manage ObjectGrid transactions. This plug-in is deprecated. Use the WebSphere eXtreme Scale resource adapter instead.



The WebSphereTransactionCallback interface has been replaced by the WebSphere eXtreme Scale resource adapter, which enables Java Transaction API

(JTA) transaction management. You can install this resource adapter on WebSphere Application Server or other Java Platform, Enterprise Edition (Java EE) application servers. The WebSphereTransactionCallback plug-in is not an enlisted JTA API, and therefore, is not designed to roll back the JTA transaction if the commit fails.

When you are using an ObjectGrid session within a method that is configured to use container-managed transactions, the enterprise container automatically begins, commits or rolls back the ObjectGrid transaction. When you are using Java Transaction API (JTA) UserTransaction objects, the ObjectGrid transaction is managed by the UserTransaction object automatically.

For a detailed discussion of the implementation of this plug-in, see “External transaction managers” on page 358.

Note: The ObjectGrid does not support 2-phase, XA transactions. This plug-in does not enlist the ObjectGrid transaction with the transaction manager. Therefore, if the ObjectGrid fails to commit, any other resources that are managed by the XA transaction do not roll back.

Programmatically plug in the WebSphereTransactionCallback object

You can enable the WebSphereTransactionCallback into the ObjectGrid configuration with programmatic configuration or XML configuration. The following code snippet uses the application to create the WebSphereTransactionCallback object and add it to an ObjectGrid:

```
ObjectGridManager objectGridManager = ObjectGridManagerFactory.getObjectGridManager();
ObjectGrid myGrid = objectGridManager.createObjectGrid("myGrid", false);
WebSphereTransactionCallback wsTxCallback= new WebSphereTransactionCallback ();
myGrid.setTransactionCallback(wsTxCallback);
```

XML configuration approach to plug in the WebSphereTransactionCallback object

The following XML configuration creates the WebSphereTransactionCallback object and adds it to an ObjectGrid. The following text must be in the myGrid.xml file:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="myGrid">
      <bean id="TransactionCallback" className=
        "com.ibm.websphere.objectgrid.plugins.builtins.WebSphereTransactionCallback" />
    </objectGrid>
  </objectGrids>
</objectGridConfig>
```

Programming to use the OSGi framework

You can start eXtreme Scale servers and clients in an OSGi container, which allows you to dynamically add and update eXtreme Scale plug-ins to the runtime environment.

Building eXtreme Scale dynamic plug-ins

WebSphere eXtreme Scale includes ObjectGrid and BackingMap plug-ins. These plug-ins are implemented in Java and are configured using the ObjectGrid descriptor XML file. To create a dynamic plug-in that can be dynamically upgraded, they need to be aware of ObjectGrid and BackingMap life cycle events because they might need to complete some actions during an update. Enhancing a

plug-in bundle with life cycle callback methods, event listeners, or both allows the plug-in to complete those actions at the appropriate times.

Before you begin

This topic assumes that you have built the appropriate plug-in. For more information about developing eXtreme Scale plug-ins, see the System APIs and plug-ins topic.

About this task

All eXtreme Scale plug-ins apply to either a BackingMap or ObjectGrid instance. Many plug-ins also interact with other plug-ins. For example, a Loader and TransactionCallback plug-in work together to properly interact with a database transaction and the various database JDBC calls. Some plug-ins might also need to cache configuration data from other plug-ins to improve performance.

The BackingMapLifecycleListener and ObjectGridLifecycleListener plug-ins provide life cycle operations for the respective BackingMap and ObjectGrid instances. This process allows plug-ins to be notified when the parent BackingMap or ObjectGrid and their respective plug-ins might be changed. BackingMap plug-ins implement the BackingMapLifecycleListener interface, and ObjectGrid plug-ins implement the ObjectGridLifecycleListener interface. These plug-ins are automatically invoked when the life cycle of the parent BackingMap or ObjectGrid changes. For more information about life cycle plug-ins, see the “Managing plug-in life cycles” on page 279 topic.

You can expect to enhance bundles using the life cycle methods or event listeners in the following common tasks:

- Starting and stopping resources, such as threads or messaging subscribers.
- Specifying that a notification occur when peer plug-ins have been updated, allowing direct access to the plug-in and detecting any changes.

Whenever you access another plug-in directly, access that plug-in through the OSGi container to ensure that all parts of the system reference the correct plug-in. If, for example, some component in the application directly references, or caches, an instance of a plug-in, it will maintain its reference to that version of the plug-in, even after that plug-in has been dynamically updated. This behavior can cause application-related problems as well as memory leaks. Therefore, write code that depends on dynamic plug-ins that obtain its reference using OSGi, `getService()` semantics. If the application must cache one or more plug-ins, it listens for life cycle events using `ObjectGridLifecycleListener` and `BackingMapLifecycleListener` interfaces. The application must also be able to refresh its cache when necessary, in a thread safe manner.

All eXtreme Scale plug-ins used with OSGi must also implement the respective `BackingMapPlugin` or `ObjectGridPlugin` interfaces. New plug-ins such as the `MapSerializerPlugin` interface enforce this practice. These interfaces provide the eXtreme Scale runtime environment and OSGi a consistent interface for injecting state into the plug-in and controlling its life cycle.

Use this task to specify that a notification occurs when peer plug-ins are updated, you might create a listener factory that produces a listener instance.

Procedure

- Update the ObjectGrid plug-in class to implement the ObjectGridPlugin interface. This interface includes methods that allow eXtreme Scale to initialize, set the ObjectGrid instance and destroy the plug-in. See the following code example:

```
package com.mycompany;
import com.ibm.websphere.objectgrid.plugins.ObjectGridPlugin;
...

public class MyTranCallback implements TransactionCallback, ObjectGridPlugin {

    private ObjectGrid og = null;

    private enum State {
        NEW, INITIALIZED, DESTROYED
    }

    private State state = State.NEW;

    public void setObjectGrid(ObjectGrid grid) {
        this.og = grid;
    }

    public ObjectGrid getObjectGrid() {
        return this.og;
    }
    void initialize() {
        // Handle any plug-in initialization here. This is called by
        // eXtreme Scale, and not the OSGi bean manager.
        state = State.INITIALIZED;
    }
    boolean isInitialized() {
        return state == State.INITIALIZED;
    }

    public void destroy() {
        // Destroy the plug-in and release any resources. This
        // can be called by the OSGi Bean Manager or by eXtreme Scale.
        state = State.DESTROYED;
    }

    public boolean isDestroyed() {
        return state == State.DESTROYED;
    }
}
```

- Update the ObjectGrid plug-in class to implement the ObjectGridLifecycleListener interface. See the following code example:

```
package com.mycompany;
import com.ibm.websphere.objectgrid.plugins.ObjectGridLifecycleListener;
import com.ibm.websphere.objectgrid.plugins.ObjectGridLifecycleListener.LifecycleEvent;
...

public class MyTranCallback implements TransactionCallback, ObjectGridPlugin, ObjectGridLifecycleListener{
    public void objectGridStateChanged(LifecycleEvent event) {
        switch(event.getState()) {
            case NEW:
            case DESTROYED:
            case DESTROYING:
            case INITIALIZING:
                break;
            case INITIALIZED:
                // Lookup a Loader or MapSerializerPlugin using
                // OSGi or directly from the ObjectGrid instance.
                lookupOtherPlugins()
                break;
            case STARTING:
            case PRELOAD:
                break;
            case ONLINE:
                if (event.isWritable()) {
                    startupProcessingForPrimary();
                } else {
                    startupProcessingForReplica();
                }
                break;
            case QUIESCE:
                if (event.isWritable()) {
                    quiesceProcessingForPrimary();
                } else {
                    quiesceProcessingForReplica();
                }
                break;
            case OFFLINE:
                shutdownShardComponents();
                break;
        }
    }
}
```

```

    }
    ...
}

```

- Update a BackingMap plug-in. Update the BackingMap plug-in class to implement the BackingMap plu-in interface. This interface includes methods that allow eXtreme Scale to initialize, set the BackingMap instance, and destroy the plug-in. See the following code example:

```

package com.mycompany;
import com.ibm.websphere.objectgrid.plugins.BackingMapPlugin;
...

public class MyLoader implements Loader, BackingMapPlugin {

    private BackingMap bmap = null;

    private enum State {
        NEW, INITIALIZED, DESTROYED
    }

    private State state = State.NEW;

    public void setBackingMap(BackingMap map) {
        this.bmap = map;
    }

    public BackingMap getBackingMap() {
        return this.bmap;
    }

    void initialize() {
        // Handle any plug-in initialization here. This is called by
        // eXtreme Scale, and not the OSGi bean manager.
        state = State.INITIALIZED;
    }

    boolean isInitialized() {
        return state == State.INITIALIZED;
    }

    public void destroy() {
        // Destroy the plug-in and release any resources. This
        // can be called by the OSGi Bean Manager or by eXtreme Scale.
        state = State.DESTROYED;
    }

    public boolean isDestroyed() {
        return state == State.DESTROYED;
    }
}

```

- Update the BackingMap plug-in class to implement the BackingMapLifecycleListener interface. See the following code example:

```

package com.mycompany;

import com.ibm.websphere.objectgrid.plugins.BackingMapLifecycleListener;
import com.ibm.websphere.objectgrid.plugins.BackingMapLifecycleListener.LifecycleEvent;
...

public class MyLoader implements Loader, ObjectGridPlugin, ObjectGridLifecycleListener{
    ...
    public void backingMapStateChanged(LifecycleEvent event) {
        switch(event.getState()) {
            case NEW:
            case DESTROYED:
            case DESTROYING:
            case INITIALIZING:
                break;
            case INITIALIZED:
                // Lookup a MapSerializerPlugin using
                // OSGi or directly from the ObjectGrid instance.
                lookupOtherPlugins()
                break;
            case STARTING:
            case PRELOAD:
                break;
            case ONLINE:
                if (event.isWritable()) {
                    startupProcessingForPrimary();
                } else {
                    startupProcessingForReplica();
                }
                break;
            case QUIESCE:
                if (event.isWritable()) {
                    quiesceProcessingForPrimary();
                } else {
                    quiesceProcessingForReplica();
                }
                break;
            case OFFLINE:

```

```

        shutdownShardComponents();
        break;
    }
    ...
}

```

Results

By implementing the `ObjectGridPlugin` or `BackingMapPlugin` interface, eXtreme Scale can control the life cycle of your plug-in at the right times.

By implementing the `ObjectGridLifecycleListener` or `BackingMapLifecycleListener` interface, the plug-in is automatically registered as a listener of the associated `ObjectGrid` or `BackingMap` life cycle events. The `INITIALIZING` event is used to signal that all of the `ObjectGrid` and `BackingMap` plug-ins have been initialized and are available for lookup and use. The `ONLINE` event is used to signal that the `ObjectGrid` is online and ready to start processing events.

Programming for JPA integration

The Java Persistence API (JPA) is a specification that allows mapping Java objects to relational databases. JPA contains a full object-relational mapping (ORM) specification using Java language metadata annotations, XML descriptors, or both to define the mapping between Java objects and a relational database. A number of open-source and commercial implementations are available.

To use JPA, you must have a supported JPA provider, such as OpenJPA or Hibernate, JAR files, and a `META-INF/persistence.xml` file in your class path.

JPA Loaders

The Java Persistence API (JPA) is a specification that allows mapping Java objects to relational databases. JPA contains a full object-relational mapping (ORM) specification using Java language metadata annotations, XML descriptors, or both to define the mapping between Java objects and a relational database. A number of open-source and commercial implementations are available.

You can use a Java Persistence API (JPA) loader plug-in implementation with eXtreme Scale to interact with any database supported by your chosen loader. To use JPA, you must have a supported JPA provider, such as OpenJPA or Hibernate, JAR files, and a `META-INF/persistence.xml` file in your class path.

The `JPALoader` `com.ibm.websphere.objectgrid.jpa.JPALoader` and the `JPAEntityLoader` `com.ibm.websphere.objectgrid.jpa.JPAEntityLoader` plug-ins are two built-in JPA loader plug-ins that are used to synchronize the `ObjectGrid` maps with a database. You must have a JPA implementation, such as Hibernate or OpenJPA, to use this feature. The database can be any back end that is supported by the chosen JPA provider.

You can use the `JPALoader` plug-in when you are storing data using the `ObjectMap` API. Use the `JPAEntityLoader` plug-in when you are storing data using the `EntityManager` API.

JPA loader architecture

The JPA Loader is used for eXtreme Scale maps that store plain old Java objects (POJO).

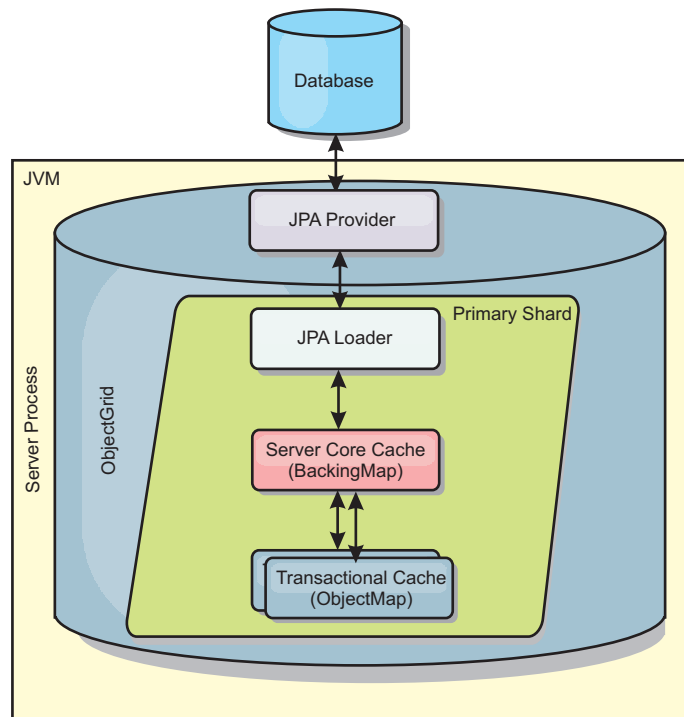


Figure 29. JPA Loader architecture

When an `ObjectMap.get(Object key)` method is called, the eXtreme Scale run time first checks whether the entry is contained in the `ObjectMap` layer. If not, the run time delegates the request to the JPA Loader. Upon request of loading the key, the `JPALoader` calls the `JPA EntityManager.find(Object key)` method to find the data from the JPA layer. If the data is contained in the JPA entity manager, it is returned; otherwise, the JPA provider interacts with the database to get the value.

When an update to `ObjectMap` occurs, for example, using the `ObjectMap.update(Object key, Object value)` method, the eXtreme Scale run time creates a `LogElement` for this update and sends it to the `JPALoader`. The `JPALoader` calls the `JPA EntityManager.merge(Object value)` method to update the value to the database.

For the `JPAEntityLoader`, the same four layers are involved. However, because the `JPAEntityLoader` plug-in is used for maps that store eXtreme Scale entities, relations among entities could complicate the usage scenario. An eXtreme Scale entity is distinguished from JPA entity. For more details, see “`JPAEntityLoader` plug-in” on page 339.

Methods

Loaders provide three main methods:

1. `get`: Returns a list of values that correspond to the list of keys that are passed in by retrieving the data using JPA. The method uses JPA to find the entities in the database. For the `JPALoader` plug-in, the returned list contains a list of JPA entities directly from the find operation. For the `JPAEntityLoader` plug-in, the returned list contains eXtreme Scale entity value tuples that are converted from the JPA entities.
2. `batchUpdate`: Writes the data from `ObjectGrid` maps to the database. Depending on different operation types (insert, update, or delete), the loader

uses the JPA persist, merge, and remove operations to update the data to the database. For the JPALoader, the objects in the map are directly used as JPA entities. For the JPAEntityLoader, the entity tuples in the map are converted into objects which are used as JPA entities.

3. preloadMap: Preloads the map using the ClientLoader.load client loader method. For partitioned maps, the preloadMap method is only called in one partition. The partition is specified the preloadPartition property of the JPALoader or JPAEntityLoader class. If the preloadPartition value is set to less than zero, or greater than (*total_number_of_partitions* - 1), preload is disabled.

Both JPALoader and JPAEntityLoader plug-ins work with the JPATxCallback class to coordinate the eXtreme Scale transactions and JPA transactions. JPATxCallback must be configured in the ObjectGrid instance to use these two loaders.

Configuration and programming

If you are using JPA loaders in a multi-master environment, see “Loader considerations in a multi-master topology” on page 109. For more information about configuring JPA loaders, see the information about JPA loaders in the *Administration Guide*. For more information about programming JPA loaders, see the *Programming Guide*.

Developing client-based JPA loaders

You can implement preloading and reloading of data in your application with a Java Persistence API (JPA) utility. This capability can simplify loading the maps when the queries to the database cannot be partitioned.

Before you begin

- You must be using a JPA provider with a supported database.
- Before you preload or reload maps, you must set the availability state of the ObjectGrid to PRELOAD. You can set the availability state with the setObjectGridState method of the StateManager interface. The StateManager interface prevents other clients from accessing the ObjectGrid when it is not yet online. After you preload or reload the map, you can set the state back to ONLINE.
- When you are preloading different maps in one ObjectGrid, set the ObjectGrid state to PRELOAD one time and set the value back to ONLINE after all maps finish data loading. This coordination can be done by the ClientLoadCallback interface. Set the ObjectGrid state to PRELOAD after the first preStart notification from the ObjectGrid instance, and set it back to ONLINE after the last postFinish notification.
- If you need to preload maps from different Java virtual machines, you have to coordinate between multiple Java virtual machines. Set the ObjectGrid state to PRELOAD one time before the first map is being preloaded in any of the Java virtual machines, and set the value back to ONLINE after all maps finish data loading in all the Java virtual machines. For more information, see Managing ObjectGrid availability.

About this task

When you run a preload or reload operation on your map, the following actions occur:

1. The initial action that is taken depends on if you are running a preload or reload operation.

- **Preload operation:** The map to be preloaded is cleared. For an entity map, if any relation is configured as cascade-remove, any related maps are cleared.
 - **Reload operation:** The provided query is run on the map and the results are invalidated. For an entity map, if any relation is configured with the **CascadeType.INVALIDATE** option, the related entities are also invalidated from their maps.
2. Run the query to JPA for the entities in a batch.
 3. For each batch, a key list and value list for each partition is built.
 4. For each partition, the data grid agent is called to insert or update the data on the server side directly if it is an eXtreme Scale client. If the data grid is a local instance, the data in the maps is directly inserted or updated.

Client-based JPA preload utility overview

The client-based Java Persistence API (JPA) preload utility loads data into eXtreme Scale backing maps using a client connection to the ObjectGrid.

This capability can simplify loading the maps when the queries to the database cannot be partitioned. A loader, such as a JPA Loader can also be used and is ideal when the data can be loaded in parallel.

The client-based JPA preload utility can use either the OpenJPA or Hibernate JPA implementations to load the ObjectGrid from a database. Because WebSphere eXtreme Scale does not directly interact with the database or Java Database Connectivity (JDBC), any database that OpenJPA or Hibernate supports can be used to load the ObjectGrid.

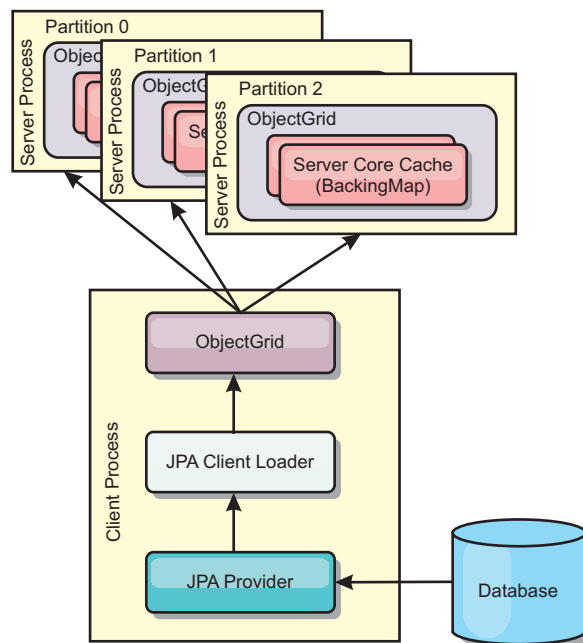


Figure 30. Client loader that uses JPA implementation to load the ObjectGrid

Typically, a user application provides a persistence unit name, an entity class name, and a JPA query to the client loader. The client loader retrieves the JPA entity manager based on the persistence unit name, uses the entity manager to query data from the database with the provided entity class and JPA query, and finally loads the data into the distributed ObjectGrid maps. When multi-level relations are

involved in the query, can use a custom query string to optimize the performance. Optionally, a persistence property map could be provided to override the configured persistence properties.

A client loader can load data in two different modes, as displayed in the following table:

Table 11. Client loader modes

Mode	Description
<i>Preload</i>	Clears and loads all entries into the backing map. If the map is an entity map, any related entity maps will also be cleared if the ObjectGrid CascadeType.REMOVE option is enabled.
<i>Reload</i>	The JPA query is executed against the ObjectGrid to invalidate all the entities in the map that match the query. If the map is an entity map, any related entity maps will also be cleared if the ObjectGrid CascadeType.INVALIDATE option is enabled.

In either case, a JPA query is used to select and load the desired entities from the database and to store them in the ObjectGrid maps. If the ObjectGrid map is a non-entity map, the JPA entities will be detached and stored directly. If the ObjectGrid map is an entity map, the JPA entities are stored as ObjectGrid entity tuples. You can provide a JPA query or use the default query `select o from EntityName o`.

For more information about configuring the client-based JPA preload utility, see “Developing client-based JPA loaders” on page 367 the information in the *Programming Guide*

Example: Preloading a map with the ClientLoader interface

You can preload a map to populate the map data before clients begin accessing the map.

Client-based preload example

The following sample code snippet shows a simple client loading. In this example, the CUSTOMER map is configured as an entity map. The Customer entity class, which is configured in the ObjectGrid entity metadata descriptor XML file, has a one-to-many relation with Order entities. The Customer entity has the CascadeType.ALL option enabled on the relation to the Order entity. Before the ClientLoader.load method is called, the ObjectGrid state is set to PRELOAD. The **isPreload** parameter on the load method is set to true.

```
// Get the StateManager
StateManager stateMgr = StateManagerFactory.getStateManager();

// Set ObjectGrid state to PRELOAD before calling ClientLoader.loader
stateMgr.setObjectGridState(AvailabilityState.PRELOAD, objectGrid);

ClientLoader c = ClientLoaderFactory.getClientLoader();

// Load the data
c.load(objectGrid, "CUSTOMER", "customerPU", null,
    null, null, null, true, null);
```

```
// Set ObjectGrid state back to ONLINE
stateMgr.setObjectGridState(AvailabilityState.ONLINE, objectGrid);
```

Example: Reloading a map with the ClientLoader interface

Reloading a map is the same as preloading a map, except that the `isPreload` argument is set to `false` in the `ClientLoader.load` method.

Client-based reload example

The following sample shows how to reload maps. Compared to the preload sample, the main difference is that a `loadSql` and parameters are provided. This sample only reloads the Customer data with an ID between 1000 and 2000. The `isPreload` parameter on the load method is set to `false`.

```
// Get the StateManager
StateManager stateMgr = StateManagerFactory.getStateManager();

// Set ObjectGrid state to PRELOAD before calling ClientLoader.loader
stateMgr.setObjectGridState(AvailabilityState.PRELOAD, objectGrid);

ClientLoader c = ClientLoaderFactory.getClientLoader();

// Load the data
String loadSql = "select c from CUSTOMER c
  where c.custId >= :startCustId and c.custId < :endCustId ";
Map<String, Long> params = new HashMap<String, Long>();
params.put("startCustId", 1000L);
params.put("endCustId", 2000L);

c.load(objectGrid, "CUSTOMER", "customerPU", null, null,
  loadSql, params, false, null);

// Set ObjectGrid state back to ONLINE
stateMgr.setObjectGridState(AvailabilityState.ONLINE, objectGrid);
```

Remember: This query string observes both JPA query syntax and eXtreme Scale entity query syntax. This query string is important because it runs twice: to invalidate the matched ObjectGrid entities and to load the matched JPA entities.

Example: Calling a client loader

You can use the `preload` method in the `Loader` interface to call a client loader.

Use the `preload` method in the `Loader` interface to call a client loader:

```
void preloadMap(Session session, BackingMap backingMap) throws LoaderException;
```

This method signals the loader to preload the data into the map. A loader implementation can use a client loader to preload the data to all its partitions. For example, the JPA loader uses the client loader to preload data into the map.

For more information, see the JPA loaders overview topic in the *Product Overview*.

Example: Calling a client loader with the preloadMap method

An example of how to preload the map using the client loader in the `preloadMap` method follows. The example first checks whether the current partition number is the same as the preload partition. If the partition number is not the same as the preload partition, no action occurs. If the partition numbers match, the client loader is called to load data into the maps. You must call the client loader in one and only one partition.

```

void preloadMap (Session session, BackingMap backingMap) throws LoaderException {
    ....
    ObjectGrid objectGrid = session.getObjectGrid();
    int partitionId = backingMap.getPartitionId();
    int numPartitions = backingMap.getPartitionManager().getNumOfPartitions();

    // Only call client loader data in one partition
    if (partitionId == preloadPartition) {
        ClientLoader c = ClientLoaderFactory.getClientLoader();
        // Call the client loader to load the data
        try {
            c.load(objectGrid, "CUSTOMER", "customerPU",
                null, entityClass, null, null, true, null);
        } catch (ObjectGridException e) {
            LoaderException le = new LoaderException("Exception caught in ObjectMap " +
                ogName + "." + mapName);
            le.initCause(e);
            throw le;
        }
    }
}

```

Remember: Configure the backingMap attribute "preloadMode" to true, so the preload method runs asynchronously. Otherwise, the preload method blocks the ObjectGrid instance from being activated.

Example: Creating a custom client-based JPA loader

The ClientLoader.load method in the Loader interface provides a client load function that satisfies most scenarios. However, if you want to load data without the ClientLoader.load method, you can implement your own preload utility.

Custom loader template

Use the following template to develop your loader:

```

// Get the StateManager
StateManager stateMgr = StateManagerFactory.getStateManager();

// Set ObjectGrid state to PRELOAD before calling ClientLoader.loader
stateMgr.setObjectGridState(AvailabilityState.PRELOAD, objectGrid);

// Load the data
...<your preload utility implementation>...

// Set ObjectGrid state back to ONLINE
stateMgr.setObjectGridState(AvailabilityState.ONLINE, objectGrid);

```

Developing a client-based JPA loader with a DataGrid agent

When loading from the client side, using a DataGrid agent could increase performance. By using a DataGrid agent, all the data reads and writes occur in the server process. You can also design your application to make sure that DataGrid agents on multiple partitions run in parallel to further boost performance.

About this task

For more information about the DataGrid agent, see "DataGrid APIs and partitioning" on page 248.

After you create the data preload implementation, you can create a generic Loader to complete the following tasks:

- Query the data from database in batches.
- Build a key list and value list for each partition.
- For each partition, call the `agentMgr.callReduceAgent(agent, aKey)` method to run the agent in the server in a thread. By running in a thread, you can run agents concurrently on multiple partitions.

Example

The following snippet of code is an example of how to load using a DataGrid agent:

```
import java.io.Externalizable;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectOutput;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;

import com.ibm.websphere.objectgrid.NoActiveTransactionException;
import com.ibm.websphere.objectgrid.ObjectGridException;
import com.ibm.websphere.objectgrid.ObjectGridRuntimeException;
import com.ibm.websphere.objectgrid.ObjectMap;
import com.ibm.websphere.objectgrid.Session;
import com.ibm.websphere.objectgrid.TransactionException;
import com.ibm.websphere.objectgrid.datagrid.ReduceGridAgent;
import com.ibm.websphere.objectgrid.em.EntityManager;

public class InsertAgent implements ReduceGridAgent, Externalizable {

    private static final long serialVersionUID = 6568906743945108310L;

    private List keys = null;

    private List vals = null;

    protected boolean isEntityMap;

    public InsertAgent() {
    }

    public InsertAgent(boolean entityMap) {
        isEntityMap = entityMap;
    }

    public Object reduce(Session sess, ObjectMap map) {
        throw new UnsupportedOperationException(
            "ReduceGridAgent.reduce(Session, ObjectMap)");
    }

    public Object reduce(Session sess, ObjectMap map, Collection arg2) {
        Session s = null;
        try {
            s = sess.getObjectGrid().getSession();
            ObjectMap m = s.getMap(map.getName());
            s.beginNoWriteThrough();
            Object ret = process(s, m);
            s.commit();
            return ret;
        } catch (ObjectGridRuntimeException e) {
            if (s.isTransactionActive()) {
                try {
                    s.rollback();
                } catch (TransactionException e1) {

```

```

        } catch (NoActiveTransactionException e1) {
        }
    }
    throw e;
} catch (Throwable t) {
    if (s.isTransactionActive()) {
        try {
            s.rollback();
        } catch (TransactionException e1) {
        } catch (NoActiveTransactionException e1) {
        }
    }
    throw new ObjectGridRuntimeException(t);
}
}

public Object process(Session s, ObjectMap m) {
    try {

        if (!isEntityMap) {
            // In the POJO case, it is very straightforward,
            // we can just put everything in the
            // map using insert
            insert(m);
        } else {
            // 2. Entity case.
            // In the Entity case, we can persist the entities
            EntityManager em = s.getEntityManager();
            persistEntities(em);

        }
        return Boolean.TRUE;
    } catch (ObjectGridRuntimeException e) {
        throw e;
    } catch (ObjectGridException e) {
        throw new ObjectGridRuntimeException(e);
    } catch (Throwable t) {
        throw new ObjectGridRuntimeException(t);
    }
}

/**
 * Basically this is fresh load.
 * @param s
 * @param m
 * @throws ObjectGridException
 */
protected void insert(ObjectMap m) throws ObjectGridException {

    int size = keys.size();

    for (int i = 0; i < size; i++) {
        m.insert(keys.get(i), vals.get(i));
    }
}

protected void persistEntities(EntityManager em) {
    Iterator<Object> iter = vals.iterator();

    while (iter.hasNext()) {
        Object value = iter.next();
        em.persist(value);
    }
}

```

```

    }

    public Object reduceResults(Collection arg0) {
        return arg0;
    }

    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        int v = in.readByte();
        isEntityMap = in.readBoolean();
        vals = readList(in);
        if (!isEntityMap) {
            keys = readList(in);
        }
    }

    public void writeExternal(ObjectOutput out) throws IOException {
        out.write(1);
        out.writeBoolean(isEntityMap);

        writeList(out, vals);
        if (!isEntityMap) {
            writeList(out, keys);
        }
    }

    public void setData(List ks, List vs) {
        vals = vs;
        if (!isEntityMap) {
            keys = ks;
        }
    }

    /**
     * @return Returns the isEntityMap.
     */
    public boolean isEntityMap() {
        return isEntityMap;
    }

    static public void writeList(ObjectOutput oo, Collection l)
        throws IOException {
        int size = l == null ? -1 : l.size();
        oo.writeInt(size);
        if (size > 0) {
            Iterator iter = l.iterator();
            while (iter.hasNext()) {
                Object o = iter.next();
                oo.writeObject(o);
            }
        }
    }

    public static List readList(ObjectInput oi)
        throws IOException, ClassNotFoundException {
        int size = oi.readInt();
        if (size == -1) {
            return null;
        }

        ArrayList list = new ArrayList(size);
        for (int i = 0; i < size; ++i) {
            Object o = oi.readObject();
            list.add(o);
        }
    }

```



```

        }
        return list;
    }
}

```

Example: Using the Hibernate plug-in to preload data into the ObjectGrid cache

You can use the preload method of the ObjectGridHibernateCacheProvider class to preload data into the ObjectGrid cache for an entity class.

Example: Using the EntityManagerFactory class

```

EntityManagerFactory emf = Persistence.createEntityManagerFactory("testPU");
ObjectGridHibernateCacheProvider.preload("objectGridName", emf, TargetEntity.class, 100, 100);

```

Important: By default, entities are not part of the second level cache. In the Entity classes that need caching, add the @cache annotation. An example follows:

```

import org.hibernate.annotations.Cache;
import org.hibernate.annotations.CacheConcurrencyStrategy;
@Entity
@Cache(usage=CacheConcurrencyStrategy.TRANSACTIONAL)
public class HibernateCacheTest { ... }

```

You can override this default by setting the shared-cache-mode element in your persistence.xml file or by using the javax.persistence.sharedCache.mode property.

Example: Using the SessionFactory class

```

org.hibernate.cfg.Configuration cfg = new Configuration();
// use addResource, addClass, and setProperty method of Configuration to prepare
// configuration required to create SessionFactory
SessionFactory sessionFactory= cfg.buildSessionFactory();
ObjectGridHibernateCacheProvider.preload("objectGridName", sessionFactory,
TargetEntity.class, 100, 100);

```

Note:

1. In a distributed system, this preload mechanism can only be invoked from one Java virtual machine. The preload mechanism cannot run simultaneously from multiple Java virtual machines.
2. Before running the preload, you must initialize the eXtreme Scale cache by creating EntityManager using EntityManagerFactory to have all corresponding BackingMaps created; otherwise, the preload forces the cache to be initialized with only one default BackingMap to support all entities. This means a single BackingMap is shared by all entities.

Starting the JPA time-based updater

When you start the Java Persistence API (JPA) time-based updater, the ObjectGrid maps are updated with the latest changes in the database.

Before you begin

Configure the time-based updater. See [Configuring a JPA time-based data updater](#) for the information about configuring a JPA time-based data updater in the *Administration Guide*.

About this task

For more information about how the Java Persistence API (JPA) time-based data updater works, see “JPA time-based data updater” on page 378.

Procedure

- Start a time-based database updater.
 - **Automatically for distributed eXtreme Scale:**

If you create the `timeBasedDBUpdate` configuration for the backing map, the time-based database updater is automatically started when a distributed ObjectGrid primary shard is activated. For an ObjectGrid that has multiple partitions, the time-based database updater only starts at partition 0.
 - **Automatically for local eXtreme Scale:**

If you create the `timeBasedDBUpdate` configuration for the backing map, the time-based database updater is automatically started when the local map is activated.
 - **Manually:**

You can also manually start or stop a time-based database updater using the `TimeBasedDBUpdater` API.

```
public synchronized void startDBUpdate(ObjectGrid objectGrid, String mapName,  
String punitName, Class entityClass, String timestampField, DBUpdateMode mode) {
```

1. **ObjectGrid:** the ObjectGrid instance (local or client).
2. **mapName:** the name of the backing map for which the time-based database updater is started.
3. **punitName:** the JPA persistence unit name for creating a JPA entity manager factory; the default value is the name of the first persistence unit defined in the `persistence.xml` file.
4. **entityClass:** The entity class name used to interact with the Java Persistence API (JPA) provider; the entity class name is used to get JPA entities using entity queries.
5. **timestampField:** A timestamp field of the entity class to identify the time or sequence when a database back end record was last updated or inserted.
6. **mode:** The time-based database update mode; an `INVALIDATE_ONLY` type causes it to invalidate the entries in the ObjectGrid map if the corresponding records in the database have changed; an `UPDATE_ONLY` type indicates to update the existing entries in the ObjectGrid map with the latest values from the database; however, all the newly inserted records to the database are ignored; an `INSERT_UPDATE` type indicates to update the existing entries in the ObjectGrid map with the latest values from the database; also, all the newly inserted records to the database are inserted into the ObjectGrid map.

If you want to stop the time-based database updater, you can call the following method to stop the updater:

```
public synchronized void stopDBUpdate(ObjectGrid objectGrid, String mapName)
```

The `ObjectGrid` and `mapName` parameters should be the same as those passed in the `startDBUpdate` method.

- Create the timestamp field in your database.
 - **DB2**

As a part of the optimistic locking feature, DB2 9.5 provides a row change timestamp feature. You can define a column `ROWCHGTS` using the `ROW CHANGE TIMESTAMP` format as follows:

```
ROWCHGTS TIMESTAMP NOT NULL  
GENERATED ALWAYS  
FOR EACH ROW ON UPDATE AS  
ROW CHANGE TIMESTAMP
```

Then you can indicate the entity field which corresponds to this column as the timestamp field by either annotation or configuration. An example follows:

```
@Entity(name = "USER_DB2")
@Table(name = "USER1")
public class User_DB2 implements Serializable {

    private static final long serialVersionUID = 1L;

    public User_DB2() {
    }

    public User_DB2(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @Column(name = "ID")
    public int id;

    @Column(name = "FIRSTNAME")
    public String firstName;

    @Column(name = "LASTNAME")
    public String lastName;

    @com.ibm.websphere.objectgrid.jpa.dbupdate.annotation.Timestamp
    @Column(name = "ROWCHGTS", updatable = false, insertable = false)
    public Timestamp rowChgTs;
}
```

– Oracle

In Oracle, there is a pseudo-column `ora_rowscn` for the system change number of the record. You can use this column for the same purpose. An example of the entity that uses the `ora_rowscn` field as the time-based database update timestamp field follows:

```
@Entity(name = "USER_ORA")
@Table(name = "USER1")
public class User_ORA implements Serializable {

    private static final long serialVersionUID = 1L;

    public User_ORA() {
    }

    public User_ORA(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @Column(name = "ID")
    public int id;

    @Column(name = "FIRSTNAME")
    public String firstName;

    @Column(name = "LASTNAME")
    public String lastName;
}
```

```

        @com.ibm.websphere.objectgrid.jpa.dbupdate.annotation.Timestamp
        @Column(name = "ora_rowscn", updatable = false, insertable = false)
        public long rowChgTs;
    }

```

– Other databases

For other types of databases, you can create a table column to track the changes. The column values have to be manually managed by the application that updates the table.

Take an Apache Derby database as an example: You can create a column "ROWCHGTS" to track the change numbers. Also, a latest change number is tracked for this table. Every time a record is inserted or updated, the latest change number for the table is incremented, and the ROWCHGTS column value for the record is updated with this incremented number.

```

@Entity(name = "USER_DER")
@Table(name = "USER1")
public class User_DER implements Serializable {

    private static final long serialVersionUID = 1L;

    public User_DER() {
    }

    public User_DER(int id, String firstName, String lastName) {
        this.id = id;
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Id
    @Column(name = "ID")
    public int id;

    @Column(name = "FIRSTNAME")
    public String firstName;

    @Column(name = "LASTNAME")
    public String lastName;

    @com.ibm.websphere.objectgrid.jpa.dbupdate.annotation.Timestamp
    @Column(name = "ROWCHGTS", updatable = true, insertable = true)
    public long rowChgTs;
}

```

JPA time-based data updater

A Java Persistence API (JPA) time-based database updater updates the ObjectGrid maps with the latest changes in the database.

When changes are made directly to a database that is being fronted by WebSphere eXtreme Scale, those changes are not concurrently reflected in the eXtreme Scale grid. To properly implement eXtreme Scale as an in-memory database processing space, take into consideration that your grid can get out of sync with the database. Time-based database updater uses the System Change Number (SCN) capability in Oracle 10g and row change timestamp in DB2 9.5 to monitor changes in the database for invalidation and update. The updater also allows applications to have a user-defined field for the same purpose.

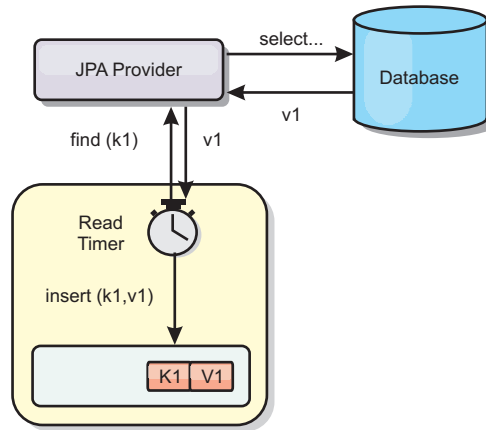


Figure 31. Periodic refresh

The time-based database updater periodically queries the database using JPA interfaces to get the JPA entities that represent the newly inserted and updated records in the database. To periodically update the records, every record in the database should have a timestamp to identify the time or sequence in which the record was last updated or inserted. It is not required that the timestamp be in a timestamp format. The timestamp value can be in an integer or long format, if it generates a unique, increasing value.

Several commercial databases have provided this capability.

For example, in DB2 9.5, you can define a column using the ROW CHANGE TIMESTAMP format as follows:

```

ROWCHGTS TIMESTAMP NOT NULL
GENERATED ALWAYS
FOR EACH ROW ON UPDATE AS
ROW CHANGE TIMESTAMP
  
```

In Oracle, you can use the pseudo-column `ora_rowscn`, which represents the system change number of the record.

The time-based database updater updates the ObjectGrid maps in three different ways:

1. `INVALIDATE_ONLY`. Invalidate the entries in the ObjectGrid map if the corresponding records in the database have changed.
2. `UPDATE_ONLY`. Update the entries in the ObjectGrid map if the corresponding records in the database have changed. However, all the newly inserted records to the database are ignored.
3. `INSERT_UPDATE`. Update the existing entries in the ObjectGrid map with the latest values from the database. Also, all the newly inserted records to the database are inserted into the ObjectGrid map.

For more information about configuring the JPA time-based data updater, see the information in the *Administration Guide*.

Developing applications with the Spring framework

Learn how to integrate your eXtreme Scale applications with the popular Spring framework.

Spring framework overview

Spring is a framework for developing Java applications. WebSphere eXtreme Scale provides support to allow Spring to manage transactions and configure the clients and servers comprising your deployed in-memory data grid.

8.5+ Spring cache provider

Spring Framework Version 3.1 introduced a new cache abstraction. With this new abstraction, you can transparently add caching to an existing Spring application. You can use WebSphere eXtreme Scale as the cache provider for the cache abstraction. For more information, see *Configuring a Spring cache provider*.

Spring managed native transactions

Spring provides container-managed transactions that are similar to a Java Platform, Enterprise Edition application server. However, the Spring mechanism can use different implementations. WebSphere eXtreme Scale provides transaction manager integration which allows Spring to manage the ObjectGrid transaction life cycles. For more information, see *"Managing transactions with Spring"* on page 381.

Spring managed extension beans and namespace support

Also, eXtreme Scale integrates with Spring to allow Spring-style beans defined for extension points or plug-ins. This feature provides more sophisticated configurations and more flexibility for configuring the extension points.

In addition to Spring managed extension beans, eXtreme Scale provides a Spring namespace called "objectgrid". Beans and built-in implementations are pre-defined in this namespace, which makes it easier for users to configure eXtreme Scale.

Shard scope support

With the traditional style Spring configuration, an ObjectGrid bean can either be a singleton type or prototype type. ObjectGrid also supports a new scope called the "shard" scope. If a bean is defined as shard scope, then only one bean is created per shard. All requests for beans with an ID or IDs matching that bean definition in the same shard results in that one specific bean instance being returned by the Spring container.

The following example shows that a `com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl` bean is defined with scope set to shard. Therefore, only one instance of the `JPAPropFactoryImpl` class is created per shard.

```
<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl" scope="shard" />
```

Spring Web Flow

Spring Web Flow stores its session state in an HTTP session by default. If a web application uses eXtreme Scale for session management, then Spring automatically stores state with eXtreme Scale. Also, fault tolerance is enabled in the same manner as the session.

See the HTTP session management information in the *Product Overview* for further details.

Packaging

The eXtreme Scale Spring extensions are in the `ogspring.jar` file. This Java archive (JAR) file must be on the class path for Spring support to work. If a Java EE application that is running in a WebSphere Extended Deployment augmented WebSphere Application Server Network Deployment, put the `spring.jar` file and its associated files in the enterprise archive (EAR) modules. You must also place the `ogspring.jar` file in the same location.

Managing transactions with Spring

Spring is a popular framework for developing Java applications. WebSphere eXtreme Scale provides support to allow Spring to manage eXtreme Scale transactions and configure eXtreme Scale clients and servers.

About this task

The Spring Framework is highly integrable with eXtreme Scale, as discussed in the following sections.

Procedure

- **Native transactions:** Spring provides container-managed transactions along the style of a Java Platform, Enterprise Edition application server but has the advantage that Springs mechanism can have different implementations plugged in. This topic describes an eXtreme Scale Platform Transaction manager that can be used with Spring. This allows programmers to annotate their POJOs (plain old Java objects) and then have Spring automatically acquire Sessions from eXtreme Scale and begin, commit, rollback, suspend, and resume eXtreme Scale transactions. Spring transactions are described more fully in Chapter 10 of the official Spring reference documentation. The following explains how to create an eXtreme Scale transaction manager and use it with annotated POJOs. It also explains how to use this approach with client or local eXtreme Scale as well as a collocated Data Grid style application.
- **Transaction manager:** To work with Spring,, eXtreme Scale provides an implementation of a Spring PlatformTransactionManager. This manager can provide managed eXtreme Scale sessions to POJOs managed by Spring. Through the use of annotations, Spring manages those sessions for the POJOs in terms of transaction life cycle. The following XML snippet shows how to create a transaction Manager:

```
<aop:aspectj-autoproxy/>
<tx:annotation-driven transaction-manager="transactionManager"/>

<bean id="ObjectGridManager"
      class="com.ibm.websphere.objectgrid.ObjectGridManagerFactory"
      factory-method="getObjectGridManager"/>

<bean id="ObjectGrid"
      factory-bean="ObjectGridManager"
      factory-method="createObjectGrid"/>

<bean id="transactionManager"
      class="com.ibm.websphere.objectgrid.spring.ObjectGridSpringFactory"
      factory-method="getLocalPlatformTransactionManager"/>
</bean>

<bean id="Service" class="com.ibm.websphere.objectgrid.spring.test.TestService">
  <property name="txManager" ref="transactionManager"/>
</bean>
```

This shows the `transactionManager` bean being declared and wired in to the `Service` bean that will use Spring transactions. We will demonstrate this using annotations and this is the reason for the `tx:annotation` clause at the beginning.

- **Obtaining an ObjectGrid session:** A POJO that has methods managed by Spring can now obtain the ObjectGrid session for the current transaction using `Session s = txManager.getSession();`

This returns the session for the POJO to use. Beans participating in the same transaction will receive the same session when they call this method. Spring will automatically handle begin for the Session and also automatically invoke commit or rollback when necessary. You can obtain an ObjectGrid EntityManager also by simply calling `getEntityManager` from the Session object.

- **Setting the ObjectGrid instance for a thread:** A single Java Virtual Machine (JVM) can host many ObjectGrid instances. Each primary shard placed in a JVM has its own ObjectGrid instance. A JVM acting as a client to a remote ObjectGrid uses an ObjectGrid instance returned from the connect method's `ClientClusterContext` to interact with that Grid. Before invoking a method on a POJO using Spring transactions for ObjectGrid, the thread must be primed with the ObjectGrid instance to use. The `TransactionManager` instance has a method allowing a specific ObjectGrid instance to be specified. Once specified then any subsequent `txManager.getSession` calls will return Sessions for that ObjectGrid instance.

The following example shows a sample main for exercising this capability:

```
ClassPathXmlApplicationContext ctx = new ClassPathXmlApplicationContext(new String[]
    {"applicationContext.xml"});
SpringLocalTxManager txManager = (SpringLocalTxManager)ctx.getBean("transactionManager");
txManager.setObjectGridForThread(og);

ITestService s = (ITestService)ctx.getBean("Service");
s.initialize();
assertEquals(s.query(), "Billy");
s.update("Bobby");
assertEquals(s.query(), "Bobby");
System.out.println("Requires new test");
s.testRequiresNew(s);
assertEquals(s.query(), "1");
```

Here we use a Spring `ApplicationContext`. The `ApplicationContext` is used to obtain a reference to the `txManager` and specify an ObjectGrid to use on this thread. The code then obtains a reference to the service and invokes methods on it. Each method call at this level causes Spring to create a Session and do begin/commit calls around the method call. Any exceptions will cause a rollback.

- **SpringLocalTxManager interface:** The `SpringLocalTxManager` interface is implemented by the ObjectGrid Platform Transaction Manager and has all public interfaces. The methods on this interface are for selecting the ObjectGrid instance to use on a thread and obtaining a Session for the thread. Any POJOs using ObjectGrid local transactions should be injected with a reference to this manager instance and only a single instance need be created, that is, its scope should be singleton. This instance is created using a static method on `ObjectGridSpringFactory`. `getLocalPlatformTransactionManager()`.

Restriction: WebSphere eXtreme Scale does not support JTA or two-phase commit for various reasons mainly to do with scalability. Thus, except at a last single-phase participant, ObjectGrid does not interact in XA or JTA type global transactions. This platform manager is intended to make using local ObjectGrid transactions as easy as possible for Spring developers.

Spring managed extension beans

You can declare plain old Java objects (POJOs) to use as extension points in the `objectgrid.xml` file. If you name the beans and then specify the class name, eXtreme Scale normally creates instances of the specified class and uses those

instances as the plug-in. WebSphere eXtreme Scale can now delegate to Spring to act as the bean factory for obtaining instances of these plug-in objects.

If an application uses Spring, POJOs have a requirement to be accessible to the rest of the application.

An application can register a Spring Bean Factory instance to use for an ObjectGrid specified by name. The application creates an instance of BeanFactory or a Spring application context and then registers it with ObjectGrid using the following static method:

```
void registerSpringBeanAdapterFactory(String objectGridName, Object springBeanFactory)
```

The previous method applies to the case when eXtreme Scale finds an extension bean whose className begins with the prefix {spring}. Such an extension bean, which could be an ObjectTransformer, Loader, TransactionCallback, and so on, uses the remainder of the name as a Spring Bean name. Then it obtains the bean instance using the Spring Bean Factory.

The eXtreme Scale deployment environment can also create a Spring bean factory from a default Spring XML configuration file. If no bean factory was registered for a given ObjectGrid, then your deployment searches for an XML file called "/<ObjectGridName>_spring.xml" automatically. For example, if your data grid is called GRID, then the XML file is called "/GRID_spring.xml" and appears in the class path in the root package. ObjectGrid constructs an ApplicationContext using the "/<ObjectGridName>_spring.xml" file and constructs beans from that bean factory.

The following is an example class name:

```
"{spring}MyLoaderBean"
```

Using the previous class name allows eXtreme Scale to use Spring to search for a bean named "MyLoaderBean". You can specify Spring-managed POJOs for any extension point if the bean factory has been registered. The Spring extensions are in the ogspring.jar file. This JAR file must be on the class path for Spring support. If a J2EE application runs in WebSphere Application Server Network Deployment augmented with WebSphere Extended Deployment, then you must place the application on the application should place the spring.jar file and its associated files in the EAR modules. The ogspring.jar must also be placed in the same location.

Spring extension beans and namespace support

WebSphere eXtreme Scale provides a feature to declare plain old Java objects (POJOs) to use as extension points in the objectgrid.xml file and a way to name the beans and then specify the class name. Normally, instances of the specified class are created, and those objects are used as the plug-ins. Now, eXtreme Scale can delegate to Spring to obtain instances of these plug-in objects. If an application uses Spring then typically such POJOs have a requirement to be wired in to the rest of the application.

In some scenarios, you must use Spring to configure a plug-in, as in the following example:

```
<objectGrid name="Grid">
  <bean id="TransactionCallback" className="com.ibm.websphere.objectgrid.jpa.JPATxCallback">
    <property name="persistenceUnitName" type="java.lang.String" value="employeePU" />
  </bean>
  ...
</objectGrid>
```

The built-in TransactionCallback implementation, the `com.ibm.websphere.objectgrid.jpa.JPATxCallback` class, is configured as the TransactionCallback class. This class is configured with the **persistenceUnitName** property as shown in the previous example. The JPATxCallback class also has the JPAPropertyFactory attribute, which is of type `java.lang.Object`. The ObjectGrid XML configuration cannot support this type of configuration.

The eXtreme Scale Spring integration solves this problem by delegating the bean creation to the Spring framework. The revised configuration follows:

```
<objectGrid name="Grid">
  <bean id="TransactionCallback" className="{spring}jpaTxCallback"/>
  ...
</objectGrid>
```

The spring file for the "Grid" object contains the following information:

```
<bean id="jpaTxCallback" class="com.ibm.websphere.objectgrid.jpa.JPATxCallback" scope="shard">
  <property name="persistenceUnitName" value="employeeEMPU"/>
  <property name="JPAPropertyFactory" ref="jpaPropFactory"/>
</bean>

<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.
JPAPropFactoryImpl" scope="shard">
</bean>
```

Here, the TransactionCallback is specified as `{spring}jpaTxCallback`, and the `jpaTxCallback` and `jpaPropFactory` bean are configured in the spring file as shown in the previous example. The Spring configuration makes it possible to configure a JPAPropertyFactory bean as a parameter of the JPATxCallback object.

Default Spring bean factory

When eXtreme Scale finds a plug-in or an extension bean (such as an ObjectTransformer, Loader, TransactionCallback, and so on) with a classname value that begins with the prefix `{spring}`, then eXtreme Scale uses the remainder of the name as a Spring Bean name and obtain the bean instance using the Spring Bean Factory.

By default, if no bean factory was registered for a given ObjectGrid, then it tries to find an `ObjectGridName_spring.xml` file. For example, if your data grid is called "Grid" then the XML file is called `/Grid_spring.xml`. This file should be in the class path or in a META-INF directory which is in the class path. If this file is found, then eXtreme Scale constructs an ApplicationContext using that file and constructs beans from that bean factory.

Custom Spring bean factory

WebSphere eXtreme Scale also provides an `ObjectGridSpringFactory` API to register a Spring Bean Factory instance to use for a specific named ObjectGrid. This API registers an instance of `BeanFactory` with eXtreme Scale using the following static method:

```
void registerSpringBeanAdapterFactory(String objectGridName, Object
springBeanFactory)
```

Namespace support

Since version 2.0, Spring has a mechanism for schema-based extensions to the basic Spring XML format for defining and configuring beans. ObjectGrid uses this new feature to define and configure ObjectGrid beans. With Spring XML schema

extension, some of the built-in implementations of eXtreme Scale plug-ins and some ObjectGrid beans are predefined in the "objectgrid" namespace. When writing the Spring configuration files, you do not have to specify the full class name of the built-in implementations. Instead, you can reference the predefined beans.

Also, with the attributes of the beans defined in the XML schema, you are less likely to provide a wrong attribute name. XML validation based on the XML schema can catch these kind of errors earlier in the development cycle.

These beans defined in the XML schema extensions are:

- transactionManager
- register
- server
- catalog
- catalogServerProperties
- container
- JPALoader
- JPATxCallback
- JPAEntityLoader
- LRUEvictor
- LFUEvictor
- HashIndex

These beans are defined in the objectgrid.xsd XML schema. This XSD file is shipped as com/ibm/ws/objectgrid/spring/namespace/objectgrid.xsd file in the ogspring.jar file . For detailed descriptions of the XSD file and the beans defined in the XSD file, see Spring descriptor XML filethe information about the Spring descriptor file in the *Administration Guide*.

Use the JPATxCallback example from the previous section. In the previous section, the JPATxCallback bean is configured as the following:

```
<bean id="jpaTxCallback" class="com.ibm.websphere.objectgrid.jpa.JPATxCallback" scope="shard">
  <property name="persistenceUnitName" value="employeeEMPU"/>
  <property name="JPAPropertyFactory" ref="jpaPropFactory"/>
</bean>

<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl" scope="shard">
</bean>
```

Using this namespace feature, the spring XML configuration can be written as the following:

```
<objectgrid:JPATxCallback id="jpaTxCallback" persistenceUnitName="employeeEMPU"
  jpaPropertyFactory="jpaPropFactory" />

<bean id="jpaPropFactory" class="com.ibm.ws.objectgrid.jpa.plugins.JPAPropFactoryImpl"
  scope="shard">
</bean>
```

Notice here that instead of specifying the com.ibm.websphere.objectgrid.jpa.JPATxCallback class as in the previous example, we directly use the pre-defined objectgrid:JPATxCallback bean. As you can see, this configuration is less verbose and more friendly to error checking.

For a description of working with Spring beans, consult "Starting a container server with Spring" on page 386.

Starting a container server with Spring

You can start a container server using Spring managed extension beans and namespace support.

About this task

With several XML files configured for Spring, you can start basic eXtreme Scale container servers.

Procedure

1. ObjectGrid XML file:

First of all, define a very simple ObjectGrid XML file which contains one ObjectGrid "Grid" and one map "Test". The ObjectGrid has an ObjectGridEventListener plug-in called "partitionListener", and the map "Test" has an Evictor plugged in called "testLRUEvictor". Notice both the ObjectGridEventListener plug-in and Evictor plug-in are configured using Spring as their names contain "{spring}".

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="Grid">
      <bean id="ObjectGridEventListener" className="{spring}partitionListener" />
      <backingMap name="Test" pluginCollectionRef="test" />
    </objectGrid>
  </objectGrids>

  <backingMapPluginCollections>
    <backingMapPluginCollection id="test">
      <bean id="Evictor" className="{spring}testLRUEvictor"/>
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

2. ObjectGrid deployment XML file:

Now, create a simple ObjectGrid deployment XML file as follows. It partitions the ObjectGrid into 5 partitions, and no replica is required.

```
<?xml version="1.0" encoding="UTF-8"?>
<deploymentPolicy xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/deploymentPolicy ../deploymentPolicy.xsd"
  xmlns="http://ibm.com/ws/objectgrid/deploymentPolicy">
  <objectgridDeployment objectgridName="Grid">
    <mapSet name="mapSet" numInitialContainers="1" numberOfPartitions="5" minSyncReplicas="0"
      maxSyncReplicas="1" maxAsyncReplicas="0">
      <map ref="Test" />
    </mapSet>
  </objectgridDeployment>
</deploymentPolicy>
```

3. ObjectGrid Spring XML file:

Now we will use both ObjectGrid Spring managed extension beans and namespace support features to configure the ObjectGrid beans. The spring xml file is named Grid_spring.xml. Notice two schemas are included in the XML file: spring-beans-2.0.xsd is for using the Spring managed beans, and objectgrid.xsd is for using the beans predefined in the objectgrid namespace.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:objectgrid="http://www.ibm.com/schema/objectgrid"
  xsi:schemaLocation="
    http://www.ibm.com/schema/objectgrid
    http://www.ibm.com/schema/objectgrid/objectgrid.xsd
```

```

    http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.0.xsd">

    <objectgrid:register id="ogregister" gridname="Grid"/>

    <objectgrid:server id="server" isCatalog="true" name="server">
        <objectgrid:catalog host="localhost" port="2809"/>
    </objectgrid:server>

    <objectgrid:container id="container"
    objectgridxml="com/ibm/ws/objectgrid/test/springshard/objectgrid.xml"
    deploymentxml="com/ibm/ws/objectgrid/test/springshard/deployment.xml"
    server="server"/>

    <objectgrid:LRUEvictor id="testLRUEvictor" numberOfLRUQueues="31"/>

    <bean id="partitionListener"
    class="com.ibm.websphere.objectgrid.springshard.ShardListener" scope="shard"/>
</beans>

```

There were six beans defined in this spring XML file:

- a. *objectgrid:register*: This register the default bean factory for the ObjectGrid "Grid".
- b. *objectgrid:server*: This defines an ObjectGrid server with name "server". This server will also provide catalog service since it has an *objectgrid:catalog* bean nested in it.
- c. *objectgrid:catalog*: This defines an ObjectGrid catalog service endpoint, which is set to "localhost:2809".
- d. *objectgrid:container*: This defines an ObjectGrid container with specified objectgrid XML file and deployment XML file as we discussed before. The server property specifies which server this container is hosted in.
- e. *objectgrid:LRUEvictor*: This defines an LRUEvictor with the number of LRU queues to use set to 31.
- f. *bean partitionListener*: This defines a ShardListener plug-in. You must provide an implementation for this plug-in, so it cannot use the pre-defined beans. Also this scope of the bean is set to "shard", which means there is only one instance of this ShardListener per ObjectGrid shard.

4. Starting the server:

The snippet below starts the ObjectGrid server, which hosts both the container service and the catalog service. As we can see, the only method we need to call to start the server is to get a bean "container" from the bean factory. This simplifies the programming complexity by moving most of the logic into Spring configuration.

```

public class ShardServer extends TestCase
{
    Container container;
    org.springframework.beans.factory.BeanFactory bf;

    public void startServer(String cep)
    {
        try
        {
            bf = new org.springframework.context.support.ClassPathXmlApplicationContext(
                "/com/ibm/ws/objectgrid/test/springshard/Grid_spring.xml", ShardServer.class);
            container = (Container)bf.getBean("container");
        }
        catch(Exception e)
        {
            throw new ObjectGridRuntimeException("Cannot start OG container", e);
        }
    }

    public void stopServer()
    {

```

```

        if(container != null)
            container.teardown();
    }
}

```

Configuring clients in the Spring framework

You can override client-side ObjectGrid settings with the Spring Framework.

About this task

The following example XML file shows how to build an ObjectGridConfiguration element, and use it to override some client side settings. You can create a similar configuration using programmatic configuration or by configuring the ObjectGrid descriptor XML file.

8.5+ For information about how to use the ObjectGridClientBean and ObjectGridCatalogServiceDomainBean beans to support the Spring Framework Version 3.1 cache abstraction, see Configuring a Spring cache provider.

Procedure

1. Create an XML file to configure clients with the Spring framework.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
  <bean id="companyGrid" factory-bean="manager" factory-method="getObjectGrid"
    singleton="true">
    <constructor-arg type="com.ibm.websphere.objectgrid.ClientClusterContext">
      <ref bean="client" />
    </constructor-arg>
    <constructor-arg type="java.lang.String" value="CompanyGrid" />
  </bean>

  <bean id="manager" class="com.ibm.websphere.objectgrid.ObjectGridManagerFactory"
    factory-method="getObjectGridManager" singleton="true">
    <property name="overrideObjectGridConfigurations">
      <map>
        <entry key="DefaultDomain">
          <list>
            <ref bean="ogConfig" />
          </list>
        </entry>
      </map>
    </property>
  </bean>

  <bean id="ogConfig"
    class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
    factory-method="createObjectGridConfiguration">
    <constructor-arg type="java.lang.String">
      <value>CompanyGrid</value>
    </constructor-arg>
    <property name="plugins">
      <list>
        <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
          factory-method="createPlugin">
          <constructor-arg type="com.ibm.websphere.objectgrid.config.PluginType"
            value="TRANSACTION_CALLBACK" />
          <constructor-arg type="java.lang.String"
            value="com.company.MyClientTxCallback" />
        </bean>
        <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
          factory-method="createPlugin">
          <constructor-arg type="com.ibm.websphere.objectgrid.config.PluginType"
            value="OBJECTGRID_EVENT_LISTENER" />
          <constructor-arg type="java.lang.String" value="" />
        </bean>
      </list>
    </property>
    <property name="backingMapConfigurations">
      <list>
        <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
          factory-method="createBackingMapConfiguration">

```

```

        <constructor-arg type="java.lang.String" value="Customer" />
        <property name="plugins">
    <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
        factory-method="createPlugin">
        <constructor-arg type="com.ibm.websphere.objectgrid.config.PluginType"
value="EVICTOR" />
    <constructor-arg type="java.lang.String"
value="com.ibm.websphere.objectgrid.plugins.builtins.LRUEvictor" />
        </bean>
    </property>
    <property name="numberOfBuckets" value="1429" />
    </bean>
    <bean class="com.ibm.websphere.objectgrid.config.ObjectGridConfigFactory"
        factory-method="createBackingMapConfiguration">
        <constructor-arg type="java.lang.String" value="OrderLine" />
        <property name="numberOfBuckets" value="701" />
    <property name="timeToLive" value="800" />
    <property name="ttlEvictorType">
        <value type="com.ibm.websphere.objectgrid.
TTLType">LAST_ACCESS_TIME</value>
    </property>
    </bean>
    </list>
    </property>
    </bean>

    <bean id="client" factory-bean="manager" factory-method="connect"
        singleton="true">
        <constructor-arg type="java.lang.String">
    <value>localhost:2809</value>
        </constructor-arg>
    <constructor-arg
        type="com.ibm.websphere.objectgrid.security.
config.ClientSecurityConfiguration">
        <null />
        </constructor-arg>
    <constructor-arg type="java.net.URL">
        <null />
        </constructor-arg>
    </bean>
</beans>

```

2. Load the XML file you created and build the ObjectGrid.

```

BeanFactory beanFactory = new XmlBeanFactory(new UrlResource
("file:test/companyGridSpring.xml"));
ObjectGrid companyGrid = (ObjectGrid) beanFactory.getBean("companyGrid");

```

Read about the “Spring framework overview” on page 121 for more information on creating an XML descriptor file.

Chapter 6. Tuning performance



You can tune settings in your environment to increase the overall performance of your WebSphere eXtreme Scale environment.

Tuning the cache sizing agent for accurate memory consumption estimates

WebSphere eXtreme Scale supports sizing the memory consumption of BackingMap instances in distributed data grids. Memory consumption sizing is not supported for local data grid instances. The value that is reported by WebSphere eXtreme Scale for a given map is very close to the value that is reported by heap dump analysis. If map object is complex, the sizings might be less accurate. The CWOBJ4543 message is displayed in the log for any cache entry object that cannot be accurately sized because it is overly complex. You can get a more accurate measurement by avoiding unnecessary map complexity.

Procedure

- Enable the sizing agent.

If you are using a Java 5 or higher Java virtual machine (JVM), use the sizing agent. With the sizing agent, WebSphere eXtreme Scale can obtain additional information from the JVM to improve its estimates. The agent can be loaded by adding the following argument to the JVM command line:

```
-javaagent:WXS lib directory/wxssizeagent.jar
```

For an embedded topology, add the argument to the command line of the WebSphere Application Server process.

For a distributed topology, add the argument to command line of the eXtreme Scale processes (containers) and the WebSphere Application Server process.

When loaded correctly, the following message is written to the SystemOut.log file.

```
CWOBJ4541I: Enhanced BackingMap memory sizing is enabled.
```

- Prefer Java data types over custom data types, where possible.

WebSphere eXtreme Scale can accurately size the memory cost of the following types:

- java.lang.String and arrays where String is the component class (String[])
- All primitive wrapper types (Byte, Short, Character, Boolean, Long, Double, Float, Integer) and arrays where primitive wrappers are the component type (for example, Integer[], Character[])
- java.math.BigDecimal and java.math.BigInteger, and arrays where these two classes are the component type (BigInteger[] and BigDecimal[])
- Temporal types (java.util.Date, java.sql.Date, java.util.Time, java.sql.Timestamp)
- java.util.Calendar and java.util.GregorianCalendar

- Avoid object internment, when possible.

When an object is inserted into a map, WebSphere eXtreme Scale assumes that it holds the only reference to the object and all the objects to which the object directly refers. If you insert 1000 custom Objects into a map, and each one has a

reference to the same string instance, then WebSphere eXtreme Scale sizes that string instance 1000 times, overestimating the actual size of the map on the heap. However, WebSphere eXtreme Scale correctly compensates for the following common internment scenarios:

- References to Java 5 Enums
- References to Classes that follow the Typesafe Enum Pattern. Classes following this pattern only have only private constructors defined, have at least one private static final field of its own type, and if they implement `Serializable`, the class implements the `readResolve()` method.
- Java 5 Primitive wrapper internment. For example, using `Integer.valueOf(1)` instead of `new Integer(1)`

If you must use internment, use one of the preceding techniques to get more accurate estimates.

- Use custom types thoughtfully.

When using custom types, prefer primitive data types for fields vs Object types. Also, prefer the Object types listed in entry 2 over your own custom implementations.

When using custom types, keep the Object tree to one level. When inserting a custom Object into a map, WebSphere eXtreme Scale will only calculate the cost of the inserted Object, which includes any primitive fields, and all the Objects it directly references. WebSphere eXtreme Scale will not follow references further down into the Object tree. If you insert an Object into the map, and WebSphere eXtreme Scale detects references that were not followed during the sizing process, a message coded `CWOBJ4543` that includes the name of the Class that could not be fully sized results. When this error occurs, treat the size statistics on the map as trend data, rather than relying on the size statistics as an accurate total.

- Use the `CopyMode.COPY_TO_BYTES` copy mode if possible.

Use the `CopyMode.COPY_TO_BYTES` copy mode to remove any uncertainty from sizing the value Objects being inserted into the map, even when an Object tree has too many levels to be sized normally (resulting in the `CWOBJ4543` message).

Cache memory consumption sizing

WebSphere eXtreme Scale can accurately estimate the Java heap memory usage of a given `BackingMap` in bytes. Use this capability to help correctly size your Java virtual machine heap settings and eviction policies. The behavior of this feature varies with the complexity of the Objects being placed in the backing map and how the map is configured. Currently, this feature is supported only for distributed data grids. Local data grid instances do not support used bytes sizing.

Heap consumption considerations

eXtreme Scale stores all of its data inside the heap space of the JVM processes that make up the data grid. For a given map, the heap space it consumes can be broken down into the following components:

- The size all the key objects currently in the map
- The size of all the value objects currently in the map
- The size of all the `EvictorData` objects that are in use by the Evictor plug-ins on the map
- The overhead of the underlying data structure

The number of used bytes that is reported by the sizing statistics is the sum of these four components. These values are calculated on a per entry basis on the insert, update, and remove map operations, meaning that eXtreme Scale always has a current value for the number of bytes that a given backing map is consuming.

When data grids are partitioned, each partition contains a piece of the backing map. Because the sizing statistics are calculated at the lowest level of the eXtreme Scale code, each partition of a backing map tracks its own size. You can use the eXtreme Scale Statistics APIs to track the cumulative size of the map, as well as the size of its individual partitions.

In general, use the sizing data as a measure of the trends of data over time, not as an accurate measurement of the heap space that is being used by the map. For example, if the reported size of a map doubles from 5 MB to 10 MB, then view the memory consumption of the map as having doubled. The actual measurement of 10 MB might be inaccurate for a number of reasons. If you take the reasons into account and follow the best practices, then the accuracy of the size measurements approaches that of post-processing a Java heap dump.

The main issue with accuracy is that the Java Memory Model is not restrictive enough to allow for memory measurements that are certain to be accurate. The fundamental problem is that an object can be live on the heap due to multiple references. For example, if the same 5 KB object instance is inserted into three separate maps, then any of those three maps prevent the object from being garbage collected. In this situation, any of the following measurements would be justifiable:

- The size of each map is increased by 5 KB.
- The size of the first map the Object is placed into is increased by 5 KB.
- The other two maps are not increased in size. The size of each map is increased by a fraction of the size of the object.

This ambiguity is why these measurements should be considered trend data, unless you have removed the ambiguity through design choices, best practices, and understanding of the implementation choices that can provide more accurate statistics.

eXtreme Scale assumes that a given map holds the only long-lived reference to the key and value Objects that it contains. If the same 5 KB object is put into three maps, then the size of each map is increased by 5 KB. The increase usually is not a problem, because the feature is supported only for distributed data grids. If you insert the same Object into three different maps on a remote client, each map receives its own copy of the Object. The default transactional COPY MODE settings also usually guarantee that each map has its own copy of a given Object.

Object interning

Object interning can cause a challenge with estimating heap memory usage. When you implement object interning, your application code purposely ensures that all references to a given object value actually point to the same object instance on the heap, and therefore the same location in memory. An example of this might be the following class:

```
public class ShippingOrder implements Serializable,Cloneable{

    public static final STATE_NEW = "new";
    public static final STATE_PROCESSING = "processing";
    public static final STATE_SHIPPED = "shipped";
```

```

        private String state;
        private int orderNumber;
private int customerNumber;

public Object clone(){
    ShippingOrder toReturn = new ShippingOrder();
    toReturn.state = this.state;
    toReturn.orderNumber = this.orderNumber;
    toReturn.customerNumber = this.customerNumber;
    return toReturn;
}

private void readResolve(){
    if (this.state.equalsIgnoreCase("new")
        this.state = STATE_NEW;
    else if (this.state.equalsIgnoreCase("processing")
        this.state = STATE_PROCESSING;
    else if (this.state.equalsIgnoreCase("shipped")
        this.state = STATE_SHIPPED;
}
}

```

Object interning causes overestimation by the sizing statistics because eXtreme Scale assumes that the objects are using different memory locations. If a million `ShippingOrder` objects exist, the sizing statistics display the cost of a million Strings holding the state information. In reality, only three Strings exist that are static class members. The memory cost for the static class members never should be added to any eXtreme Scale map. However, this situation cannot be detected at runtime. There are dozens of ways that similar object interning can be implemented, which is why it is so hard to detect. It is not practical for eXtreme Scale to protect against all possible implementations. However, eXtreme Scale does protect against the most commonly used types of object interning. To optimize memory usage with Object interning, implement interning only on custom objects that fall into the following two categories to enhance the accuracy of the memory consumption statistics:

- eXtreme Scale automatically adjusts for Java 5 enums and the Typesafe Enum pattern, as described at Java 2 Platform Standard Edition 5.0 Overview: Enums.
- eXtreme Scale automatically accounts for the automatic interning of primitive wrapper types, such as Integer. Automatic interning for primitive wrapper types was introduced in Java 5 through the use of static `valueOf` methods.

Memory consumption statistics

Use one of the following methods to access the memory consumption statistics.

Statistics API

Use the `MapStatsModule.getUsedBytes()` method, which provides statistics for a single map, including the number of entries and hit rate.

For details, see Statistics modules.

Managed Beans (MBeans)

Use the `MapUsedBytes` managed MBean statistic. You can use several different types of Java Management Extensions (JMX) MBeans to administer and monitor deployments. Each MBean refers to a specific entity, such as a map, eXtreme Scale, server, replication group, or replication group member.

For details, see Administering with Managed Beans (MBeans).

Performance monitoring infrastructure (PMI) modules

You can monitor the performance of your applications with the PMI modules. Specifically, use the map PMI module for containers embedded in WebSphere Application Server.

For details, see PMI modules.

WebSphere eXtreme Scale console

With the console, you can view the memory consumption statistics. See [Monitoring with the web console](#).

All of these methods access the same underlying measurement of the memory consumption of a given BaseMap instance. The WebSphere eXtreme Scale runtime attempts with a best effort to calculate the number of bytes of heap memory that is consumed by the key and value objects that are stored in the map, as well as the overhead of the map itself. You can see how much heap memory each map is consuming across the whole distributed data grid.

In most cases the value reported by WebSphere eXtreme Scale for a given map is very close to the value reported by heap dump analysis. WebSphere eXtreme Scale accurately sizes its own overhead, but cannot account for every possible object that might be put into a map. Following the best practices described in “[Tuning the cache sizing agent for accurate memory consumption estimates](#)” on page 391 can enhance the accuracy of the size in bytes measurements provided by WebSphere eXtreme Scale.

Tuning and performance for application development

To improve performance for your in-memory data grid or database processing space, you can investigate several considerations such using the best practices for product features such as locking, serialization, and query performance.

Tuning the copy mode

WebSphere eXtreme Scale makes a copy of the value based on the available CopyMode settings. Determine which setting works best for your deployment requirements.

You can use the BackingMap API `setCopyMode(CopyMode, valueInterfaceClass)` method to set the copy mode to one of the following final static fields that are defined in the `com.ibm.websphere.objectgrid.CopyMode` class.

When an application uses the ObjectMap interface to obtain a reference to a map entry, use that reference only within the data grid transaction that obtained the reference. Using the reference in a different transaction can lead to errors. For example, if you use the pessimistic locking strategy for the BackingMap, a `get` or `getForUpdate` method call acquires an S (shared) or U (update) lock, depending on the transaction. The `get` method returns the reference to the value and the lock that is obtained is released when the transaction completes. The transaction must call the `get` or `getForUpdate` method to lock the map entry in a different transaction. Each transaction must obtain its own reference to the value by calling the `get` or `getForUpdate` method instead of reusing the same value reference in multiple transactions.

CopyMode for entity maps

When using a map associated with an EntityManager API entity, the map always returns the entity Tuple objects directly without making a copy unless you are using COPY_TO_BYTES copy mode. It is important that the CopyMode is updated or the Tuple is copied appropriately when making changes.

COPY_ON_READ_AND_COMMIT

The COPY_ON_READ_AND_COMMIT mode is the default mode. The valueInterfaceClass argument is ignored when this mode is used. This mode ensures that an application does not contain a reference to the value object that is in the BackingMap. Instead, the application is always working with a copy of the value that is in the BackingMap. The COPY_ON_READ_AND_COMMIT mode ensures that the application can never inadvertently corrupt the data that is cached in the BackingMap. When an application transaction calls an ObjectMap.get method for a given key, and it is the first access of the ObjectMap entry for that key, a copy of the value is returned. When the transaction is committed, any changes that are committed by the application are copied to the BackingMap to ensure that the application does not have a reference to the committed value in the BackingMap.

COPY_ON_READ

The COPY_ON_READ mode improves performance over the COPY_ON_READ_AND_COMMIT mode by eliminating the copy that occurs when a transaction is committed. The valueInterfaceClass argument is ignored when this mode is used. To preserve the integrity of the BackingMap data, the application ensures that every reference that it has for an entry is destroyed after the transaction is committed. With this mode, the ObjectMap.get method returns a copy of the value instead of a reference to the value to ensure that changes that are made by the application to the value does not affect the BackingMap value until the transaction is committed. However, when the transaction does commit, a copy of changes is not made. Instead, the reference to the copy that was returned by the ObjectMap.get method is stored in the BackingMap. The application destroys all map entry references after the transaction is committed. If application does not destroy the map entry references, the application might cause the data cached in BackingMap to become corrupted. If an application is using this mode and is having problems, switch to COPY_ON_READ_AND_COMMIT mode to see if the problem still exists. If the problem goes away, then the application is failing to destroy all of its references after the transaction has committed.

COPY_ON_WRITE

The COPY_ON_WRITE mode improves performance over the COPY_ON_READ_AND_COMMIT mode by eliminating the copy that occurs when the ObjectMap.get method is called for the first time by a transaction for a given key. The ObjectMap.get method returns a proxy to the value instead of a direct reference to the value object. The proxy ensures that a copy of the value is not made unless the application calls a set method on the value interface that is specified by the valueInterfaceClass argument. The proxy provides a copy on write implementation. When a transaction commits, the BackingMap examines the proxy to determine if any copy was made as a result of a set method being called. If a copy was made, then the reference to that copy is stored in the BackingMap. The big advantage of this mode is that a value is never copied on a read or at a commit when the transaction never calls a set method to change the value.

The `COPY_ON_READ_AND_COMMIT` and `COPY_ON_READ` modes both make a deep copy when a value is retrieved from the `ObjectMap`. If an application only updates some of the values that are retrieved in a transaction then this mode is not optimal. The `COPY_ON_WRITE` mode supports this behavior efficiently but requires that the application uses a simple pattern. The value objects are required to support an interface. The application must use the methods on this interface when it is interacting with the value in a session. If this is the case, then proxies are created for the values that are returned to the application. The proxy has a reference to the real value. If the application performs read operations only, the read operations always run against the real copy. If the application modifies an attribute on the object, the proxy makes a copy of the real object and then modifies the copy. The proxy then uses the copy from that point on. Using the copy allows the copy operation to be avoided completely for objects that are only read by the application. All modify operations must start with the `set` prefix. Enterprise JavaBeans normally are coded to use this style of method naming for methods that modify the objects attributes. This convention must be followed. Any objects that are modified are copied at the time that they are modified by the application. This read and write scenario is the most efficient scenario supported by eXtreme Scale. To configure a map to use `COPY_ON_WRITE` mode, use the following example. In this example, the application stores `Person` objects that are keyed using the name in the `Map`. The person object is represented in the following code snippet.

```
class Person {
    String name;
    int age;
    public Person() {
    }
    public void setName(String n) {
        name = n;
    }
    public String getName() {
        return name;
    }
    public void setAge(int a) {
        age = a;
    }
    public int getAge() {
        return age;
    }
}
```

The application uses the `IPerson` interface only when it interacts with values that are retrieved from a `ObjectMap`. Modify the object to use an interface as in the following example.

```
interface IPerson
{
    void setName(String n);
    String getName();
    void setAge(int a);
    int getAge();
}
// Modify Person to implement IPerson interface
class Person implements IPerson {
    ...
}
```

The application then needs to configure the `BackingMap` to use `COPY_ON_WRITE` mode, like in the following example:

```
ObjectGrid dg = ...;
BackingMap bm = dg.defineMap("PERSON");
// use COPY_ON_WRITE for this Map with
```

```

// IPerson as the valueProxyInfo Class
bm.setCopyMode(CopyMode.COPY_ON_WRITE,IPerson.class);
// The application should then use the following
// pattern when using the PERSON Map.
Session sess = ...;
ObjectMap person = sess.getMap("PERSON");
...
sess.begin();
// the application casts the returned value to IPerson and not Person
IPerson p = (IPerson)person.get("Billy");
p.setAge(p.getAge()+1);
...
// make a new Person and add to Map
Person p1 = new Person();
p1.setName("Bobby");
p1.setAge(12);
person.insert(p1.getName(), p1);
sess.commit();
// the following snippet WON'T WORK. Will result in ClassCastException
sess.begin();
// the mistake here is that Person is used rather than
// IPerson
Person a = (Person)person.get("Bobby");
sess.commit();

```

The first section of the application retrieves a value that was named Billy in the map. The application casts the returned value to the IPerson object, not the Person object because the proxy that is returned implements two interfaces:

- The interface specified in the BackingMap.setCopyMode method call
- The com.ibm.websphere.objectgrid.ValueProxyInfo interface

You can cast the proxy to two types. The last part of the preceding code snippet demonstrates what is not allowed in COPY_ON_WRITE mode. The application retrieves the Bobby record and tries to cast the record to a Person object. This action fails with a class cast exception because the proxy that is returned is not a Person object. The returned proxy implements the IPerson object and ValueProxyInfo.

ValueProxyInfo interface and partial update support: This interface allows an application to retrieve either the committed read-only value referenced by the proxy or the set of attributes that have been modified during this transaction.

```

public interface ValueProxyInfo {
    List /**/ ibmGetDirtyAttributes();
    Object ibmGetRealValue();
}

```

The ibmGetRealValue method returns a read-only copy of the object. The application must not modify this value. The ibmGetDirtyAttributes method returns a list of strings that represent the attributes that were modified by the application during this transaction. The main use case for the ibmGetDirtyAttributes method is in a Java database connectivity (JDBC) or CMP-based loader. Only the attributes that are named in the list need be updated on either the SQL statement or object mapped to the table. This practice leads to more efficient SQL generated by the Loader. When a copy on write transaction is committed and if a loader is plugged in, the loader can cast the values of the modified objects to the ValueProxyInfo interface to obtain this information.

Handling the equals method when using COPY_ON_WRITE or proxies: For example, the following code constructs a Person object and then inserts it to an ObjectMap. Next, it retrieves the same object using the ObjectMap.get method. The

value is cast to the interface. If the value is cast to the Person interface, a ClassCastException exception results because the returned value is a proxy that implements the IPerson interface and is not a Person object. The equality check fails when using the == operation because they are not the same object.

```
session.begin();
// new the Person object
Person p = new Person(...);
personMap.insert(p.getName, p);
// retrieve it again, remember to use the interface for the cast
IPerson p2 = personMap.get(p.getName());
if(p2 == p) {
    // they are the same
} else {
    // they are not
}
```

Another consideration is when you must override the equals method. The equals method must verify that the argument is an object that implements the IPerson interface and cast the argument to be an IPerson object. Because the argument might be a proxy that implements the IPerson interface, you must use the getAge and getName methods when comparing instance variables for equality. See the following example:

```
{
    if ( obj == null ) return false;
    if ( obj instanceof IPerson ) {
        IPerson x = (IPerson) obj;
        return ( age.equals( x.getAge() ) && name.equals( x.getName() ) )
    }
    return false;
}
```

ObjectQuery and HashIndex configuration requirements: When you are using COPY_ON_WRITE with ObjectQuery or a HashIndex plug-ins, you must configure the ObjectQuery schema and HashIndex plug-in to access the objects using property methods, which is the default. If you configured field access, the query engine and index attempts to access the fields in the proxy object, which always returns null or 0 because the object instance is a proxy.

NO_COPY

The NO_COPY mode allows an application to obtain performance improvements, but requires that application to never modify a value object that is obtained using an ObjectMap.get method. The valueInterfaceClass argument is ignored when this mode is used. If this mode is used, no copy of the value is ever made. If the application modifies any value object instances that are retrieved from or added to the ObjectMap, then the data in the BackingMap is corrupted. The NO_COPY mode is primarily useful for read-only maps where data is never modified by the application. If the application is using this mode and it is having problems, then switch to the COPY_ON_READ_AND_COMMIT mode to see if the problem still exists. If the problem goes away, then the application is modifying the value returned by ObjectMap.get method, either during transaction or after transaction has committed. All maps associated with EntityManager API entities automatically use this mode regardless of what is specified in the eXtreme Scale configuration.

All maps associated with EntityManager API entities automatically use this mode regardless of what is specified in the eXtreme Scale configuration.

COPY_TO_BYTES

You can store objects in a serialized format instead of POJO format. By using the `COPY_TO_BYTES` setting, you can reduce the memory footprint that a large graph of objects can consume. For more information, see “Improving performance with byte array maps.”

COPY_TO_BYTES_RAW

With `COPY_TO_BYTES_RAW`, you can directly access the serialized form of your data. This copy mode offers an efficient way for you to interact with serialized bytes, which allows you to bypass the deserialization process to access objects in memory.

In the ObjectGrid descriptor XML file, you can set the copy mode to `COPY_TO_BYTES`, and programmatically set the copy mode to `COPY_TO_BYTES_RAW` in the instances where you want to access the raw, serialized data. Set the copy mode to `COPY_TO_BYTES_RAW` in the ObjectGrid descriptor XML file only when your application uses the raw data as a part of a main application process.

Incorrect use of CopyMode

Errors occur when an application attempts to improve performance by using the `COPY_ON_READ`, `COPY_ON_WRITE`, or `NO_COPY` copy mode, as described above. The intermittent errors do not occur when you change the copy mode to the `COPY_ON_READ_AND_COMMIT` mode.

Problem

The problem might be due to corrupted data in the ObjectGrid map, which is a result of the application violating the programming contract of the copy mode that is being used. Data corruption can cause unpredictable errors to occur intermittently or in an unexplained or unexpected fashion.

Solution

The application must comply with the programming contract that is stated for the copy mode being used. For the `COPY_ON_READ` and `COPY_ON_WRITE` copy modes, the application uses a reference to a value object outside of the transaction scope from which the value reference was obtained. To use these modes, the application must delete the reference to the value object after the transaction completes, and obtain a new reference to the value object in each transaction that accesses the value object. For the `NO_COPY` copy mode, the application must never change the value object. In this case, either write the application so that it does not change the value object, or set the application to use a different copy mode.

Improving performance with byte array maps

You can store values in your maps in a byte array instead of POJO form, which reduces the memory footprint that a large graph of objects can consume.

Advantages

The amount of memory that is consumed increases with the number of objects in a graph of objects. By reducing a complicated graph of objects to a byte array, only

one object is maintained in the heap instead of several objects. With this reduction of the number of objects in the heap, the Java run time has fewer objects to search for during garbage collection.

The default copy mechanism used by WebSphere eXtreme Scale is serialization, which is expensive. For instance, if using the default copy mode of `COPY_ON_READ_AND_COMMIT`, a copy is made both at read time and at get time. Instead of making a copy at read time, with byte arrays, the value is inflated from bytes, and instead of making a copy at commit time, the value is serialized to bytes. Using byte arrays results in equivalent data consistency to the default setting with a reduction of memory used.

When using byte arrays, note that having an optimized serialization mechanism is critical to seeing a reduction of memory consumption. For more information, see “Tuning serialization performance” on page 407.

Configuring byte array maps

You can enable byte array maps with the ObjectGrid XML file by modifying the `CopyMode` attribute that is used by a map to the setting `COPY_TO_BYTES`, shown in the following example:

```
<backingMap name="byteMap" copyMode="COPY_TO_BYTES" />
```

Considerations

You must consider whether or not to use byte array maps in a given scenario. Although you can reduce your memory use, processor use can increase when you use byte arrays.

The following list outlines several factors that should be considered before choosing to use the byte array map function.

Object type

Comparatively, memory reduction may not be possible when using byte array maps for some object types. Consequently, several types of objects exist for which you should not use byte array maps. If you are using any of the Java primitive wrappers as values, or a POJO that does not contain references to other objects (only storing primitive fields), the number of Java Objects is already as low as possible—there is only one. Since the amount of memory used by the object is already optimized, using a byte array map for these types of objects is not recommended. Byte array maps are more suitable to object types that contain other objects or collections of objects where the total number of POJO objects is greater than one.

For example, if you have a Customer object that had a business Address and a home Address, as well as a collection of Orders, the number of objects in the heap and the number of bytes used by those objects can be reduced by using byte array maps.

Local access

When using other copy modes, applications can be optimized when copies are made if objects are Cloneable with the default ObjectTransformer or when a custom ObjectTransformer is provided with an optimized copyValue method. Compared to the other copy modes, copying on reads, writes, or commit

operations will have additional cost when accessing objects locally. For example, if you have a near cache in a distributed topology or are directly accessing a local or server ObjectGrid instance, the access and commit time will increase when using byte array maps due to the cost of serialization. You will see a similar cost in a distributed topology if you use data grid agents or you access the server primary when using the ObjectGridEventGroup.ShardEvents plug-in.

Plug-in interactions

With byte array maps, objects are not inflated when communicating from a client to a server unless the server needs the POJO form. Plug-ins that interact with the map value will experience a reduction in performance due to the requirement to inflate the value.

Any plug-in that uses LogElement.getCacheEntry or LogElement.getCurrentValue will see this additional cost. If you want to get the key, you can use LogElement.getKey, which avoids the additional overhead associated with the LogElement.getCacheEntry().getKey method. The following sections discuss plug-ins in light of the usage of byte arrays.

Indexes and queries

When objects are stored in POJO format, the cost of doing indexing and querying is minimal because the object does not need to be inflated. When using a byte array map you will have the additional cost of inflating the object. In general if your application uses indexes or queries, it is not recommended to use byte array maps unless you only run queries on key attributes.

Optimistic locking

When using the optimistic locking strategy, you will have the additional cost during updates and invalidate operations. This comes from having to inflate the value on the server to get the version value to do optimistic collision checking. If you are just using optimistic locking to guarantee fetch operations and do not need optimistic collision checking, you can use the com.ibm.websphere.objectgrid.plugins.builtins.NoVersioningOptimisticCallback to disable version checking.

Loader


With a Loader, you will also have the cost in the eXtreme Scale run time from inflating and reserializing the value when it is used by the Loader. You can still use byte array maps with Loaders, but consider the cost of making changes to the value in such a scenario. For example, you can use the byte array feature in the context of a read mostly cache. In this case, the benefit of having less objects in the heap and less memory used will outweigh the cost incurred from using byte arrays on insert and update operations.

ObjectGridEventListener

When using the transactionEnd method in the ObjectGridEventListener plug-in, you will have an additional cost on the server side for remote requests when accessing a LogElement's CacheEntry or current value. If the implementation of the method does not access these fields, then you will not have the additional cost.

Tuning copy operations with the ObjectTransformer interface

The ObjectTransformer interface uses callbacks to the application to provide custom implementations of common and expensive operations such as object serialization and deep copies on objects.

 The ObjectTransformer interface has been replaced by the DataSerializer plug-ins, which you can use to efficiently store arbitrary data in WebSphere eXtreme Scale so that existing product APIs can efficiently interact with your data.

Overview

Copies of values are always made except when the NO_COPY mode is used. The default copying mechanism that is employed in eXtreme Scale is serialization, which is known as an expensive operation. The ObjectTransformer interface is used in this situation. The ObjectTransformer interface uses callbacks to the application to provide a custom implementation of common and expensive operations, such as object serialization and deep copies on objects.

An application can provide an implementation of the ObjectTransformer interface to a map, and eXtreme Scale then delegates to the methods on this object and relies on the application to provide an optimized version of each method in the interface. The ObjectTransformer interface follows:

```
public interface ObjectTransformer {
    void serializeKey(Object key, ObjectOutputStream stream) throws IOException;
    void serializeValue(Object value, ObjectOutputStream stream) throws IOException;
    Object inflateKey(ObjectInputStream stream) throws IOException, ClassNotFoundException;
    Object inflateValue(ObjectInputStream stream) throws IOException, ClassNotFoundException;
    Object copyValue(Object value);
    Object copyKey(Object key);
}
```

You can associate an ObjectTransformer interface with a BackingMap by using the following example code:

```
ObjectGrid g = ...;
BackingMap bm = g.defineMap("PERSON");
MyObjectTransformer ot = new MyObjectTransformer();
bm.setObjectTransformer(ot);
```

Tune deep copy operations

After an application receives an object from an ObjectMap, eXtreme Scale performs a deep copy on the object value to ensure that the copy in the BaseMap map maintains data integrity. The application can then modify the object value safely. When the transaction commits, the copy of the object value in the BaseMap map is updated to the new modified value and the application stops using the value from that point on. You could have copied the object again at the commit phase to make a private copy. However, in this case the performance cost of this action was traded off against requiring the application programmer not to use the value after the transaction commits. The default ObjectTransformer attempts to use either a clone or a serialize and inflate pair to generate a copy. The serialize and inflate pair is the worst case performance scenario. If profiling reveals that serialize and inflate is a problem for your application, write an appropriate clone method to create a deep copy. If you cannot alter the class, then create a custom ObjectTransformer plug-in and implement more efficient copyValue and copyKey methods.

Tuning evictors

If you use plug-in evictors, they are not active until you create them and associate them with a backing map. The following best practices increase performance for least frequently used (LFU) and least recently used (LRU) evictors.

Least frequently used (LFU) evictor

The concept of a LFU evictor is to remove entries from the map that are used infrequently. The entries of the map are spread over a set amount of binary heaps. As the usage of a particular cache entry grows, it becomes ordered higher in the heap. When the evictor attempts a set of evictions it removes only the cache entries that are located lower than a specific point on the binary heap. As a result, the least frequently used entries are evicted.

Least recently used (LRU) evictor

The LRU Evictor follows the same concepts of the LFU Evictor with a few differences. The main difference is that the LRU uses a first in, first out queue (FIFO) instead of a set of binary heaps. Every time a cache entry is accessed, it moves to the head of the queue. Consequently, the front of the queue contains the most recently used map entries and the end becomes the least recently used map entries. For example, the A cache entry is used 50 times, and the B cache entry is used only once right after the A cache entry. In this situation, the B cache entry is at the front of the queue because it was used most recently, and the A cache entry is at the end of the queue. The LRU evictor evicts the cache entries that are at the tail of the queue, which are the least recently used map entries.

LFU and LRU properties and best practices to improve performance

Number of heaps

When using the LFU evictor, all of the cache entries for a particular map are ordered over the number of heaps that you specify, improving performance drastically and preventing all of the evictions from synchronizing on one binary heap that contains all of the ordering for the map. More heaps also speeds up the time that is required for reordering the heaps because each heap has fewer entries. Set the number of heaps to 10% of the number of entries in your BaseMap.

Number of queues

When using the LRU evictor, all of the cache entries for a particular map are ordered over the number of LRU queues that you specify, improving performance drastically and preventing all of the evictions from synchronizing on one queue that contains all of the ordering for the map. Set the number of queues to 10% of the number of entries in your BaseMap.

MaxSize property

When an LFU or LRU evictor begins evicting entries, it uses the MaxSize evictor property to determine how many binary heaps or LRU queue elements to evict. For example, assume that you set the number of heaps or queues to have about 10 map entries in each map queue. If your MaxSize property is set to 7, the evictor evicts 3 entries from each heap or queue object to bring the size of each heap or queue back down to 7. The evictor only evicts map entries from a heap or queue

when that heap or queue has more than the MaxSize property value of elements in it. Set the MaxSize to 70% of your heap or queue size. For this example, the value is set to 7. You can get an approximate size of each heap or queue by dividing the number of BaseMap entries by the number of heaps or queues that are used.

SleepTime property

An evictor does not constantly remove entries from your map. Instead it is idle for a set amount of time, only checking the map every n number of seconds, where n refers to the SleepTime property. This property also positively affects performance: running an eviction sweep too often lowers performance because of the resources that are needed for processing too often. However, not using the evictor often can result in a map that has entries that are not needed. A map full of entries that are not needed can negatively affect both the memory requirements and processing resources that are required for your map. Setting the eviction sweep interval to fifteen seconds is a good practice for most maps. If the map is written to frequently and is used at a high transaction rate, consider setting the value to a lower time. If the map is accessed infrequently, you can set the time to a higher value.

Example

The following example defines a map, creates a new LFU evictor, sets the evictor properties, and sets the map to use the evictor:

```
//Use ObjectGridManager to create/get the ObjectGrid. Refer to
// the ObjectGridManger section
ObjectGrid objGrid = ObjectGridManager.create.....
BackingMap bMap = objGrid.defineMap("SomeMap");

//Set properties assuming 50,000 map entries
LFUEvictor someEvictor = new LFUEvictor();
someEvictor.setNumberOfHeaps(5000);
someEvictor.setMaxSize(7);
someEvictor.setSleepTime(15);
bMap.setEvictor(someEvictor);
```

Using the LRU evictor is very similar to using an LFU evictor. An example follows:

```
ObjectGrid objGrid = new ObjectGrid;
BackingMap bMap = objGrid.defineMap("SomeMap");

//Set properties assuming 50,000 map entries
LRUEvictor someEvictor = new LRUEvictor();
someEvictor.setNumberOfLRUQueues(5000);
someEvictor.setMaxSize(7);
someEvictor.setSleepTime(15);
bMap.setEvictor(someEvictor);
```

Notice that only two lines are different from the LFUEvictor example.

Tuning locking performance

Locking strategies and transaction isolation settings affect the performance of your applications.

Retrieve a cached instance

For more information, see “Lock manager” on page 220:

Pessimistic locking strategy

Use the pessimistic locking strategy for read and write map operations where keys often collide. The pessimistic locking strategy has the greatest impact on performance.

Read committed and read uncommitted transaction isolation

When you are using pessimistic locking strategy, set the transaction isolation level using the `Session.setTransactionIsolation` method. For read committed or read uncommitted isolation, use the `Session.TRANSACTION_READ_COMMITTED` or `Session.TRANSACTION_READ_UNCOMMITTED` arguments depending on the isolation. To reset the transaction isolation level to the default pessimistic locking behavior, use the `Session.setTransactionIsolation` method with the `Session.REPEATABLE_READ` argument.

Read committed isolation reduces the duration of shared locks, which can improve concurrency and reduce the chance for deadlocks. This isolation level should be used when a transaction does not need assurances that read values remain unchanged for the duration of the transaction.

Use an uncommitted read when the transaction does not need to see the committed data.

Optimistic locking strategy

Optimistic locking is the default configuration. This strategy improves both performance and scalability compared to the pessimistic strategy. Use this strategy when your applications can tolerate some optimistic update failures, while still performing better than the pessimistic strategy. This strategy is excellent for read operations and infrequent update applications.

OptimisticCallback plug-in

The optimistic locking strategy makes a copy of the cache entries and compares them as needed. This operation can be expensive because copying the entry might involve cloning or serialization. To implement the fastest possible performance, implement the custom plug-in for non-entity maps.

See for more information. See the information about the `OptimisticCallback` plug-in in the *Product Overview* for more information.

Use version fields for entities

When you are using optimistic locking with entities, use the `@Version` annotation or the equivalent attribute in the Entity metadata descriptor file. The version annotation gives the ObjectGrid a very efficient way of tracking the version of an object. If the entity does not have a version field and optimistic locking is used for the entity, then the entire entity must be copied and compared.


None locking strategy

Use the none locking strategy for applications that are read only. The none locking strategy does not obtain any locks or use a lock manager. Therefore, this strategy offers the most concurrency, performance and scalability.

Tuning serialization performance

WebSphere eXtreme Scale uses multiple Java processes to hold data. These processes serialize the data: That is, they convert the data (which is in the form of Java object instances) to bytes and back to objects again as needed to move the data between client and server processes. Marshalling the data is the most expensive operation and must be addressed by the application developer when designing the schema, configuring the data grid and interacting with the data-access APIs.

The default Java serialization and copy routines are relatively slow and can consume 60 to 70 percent of the processor in a typical setup. The following sections are choices for improving the performance of the serialization.

 The ObjectTransformer interface has been replaced by the DataSerializer plug-ins, which you can use to efficiently store arbitrary data in WebSphere eXtreme Scale so that existing product APIs can efficiently interact with your data.

Write an ObjectTransformer for each BackingMap

An ObjectTransformer can be associated with a BackingMap. Your application can have a class that implements the ObjectTransformer interface and provides implementations for the following operations:

- Copying values
- Serializing and inflating keys to and from streams
- Serializing and inflating values to and from streams

The application does not need to copy keys because keys are considered immutable.

Note: The ObjectTransformer is only invoked when the ObjectGrid knows about the data that is being transformed. For example, when DataGrid API agents are used, the agents themselves as well as the agent instance data or data returned from the agent must be optimized using custom serialization techniques. The ObjectTransformer is not invoked for DataGrid API agents.

Using entities

When using the EntityManager API with entities, the ObjectGrid does not store the entity objects directly into the BackingMaps. The EntityManager API converts the entity object to Tuple objects. Entity maps are automatically associated with a highly optimized ObjectTransformer. Whenever the ObjectMap API or EntityManager API is used to interact with entity maps, the entity ObjectTransformer is invoked.

Custom serialization

Some cases exist where objects must be modified to use custom serialization, such as implementing the `java.io.Externalizable` interface or by implementing the `writeObject` and `readObject` methods for classes implementing the `java.io.Serializable` interface. Custom serialization techniques should be employed when the objects are serialized using mechanisms other than the ObjectGrid API or EntityManager API methods.

For example, when objects or entities are stored as instance data in a DataGrid API agent or the agent returns objects or entities, those objects are not transformed using an ObjectTransformer. The agent, will however, automatically use the ObjectTransformer when using EntityMixin interface. See DataGrid agents and entity based Maps for further details.

Byte arrays

When using the ObjectMap or DataGrid APIs, the key and value objects are serialized whenever the client interacts with the data grid and when the objects are replicated. To avoid the overhead of serialization, use byte arrays instead of Java objects. Byte arrays are much cheaper to store in memory since the JDK has less objects to search for during garbage collection and they are can be inflated only when needed. Byte arrays should only be used if you do not need to access the objects using queries or indexes. Since the data is stored as bytes, the data can only be accessed through its key.


WebSphere eXtreme Scale can automatically store data as byte arrays using the CopyMode.COPY_TO_BYTES map configuration option, or it can be handled manually by the client. This option will store the data efficiently in memory and can also automatically inflate the objects within the byte array for use by query and indexes on demand.

A MapSerializerPlugin plug-in can be associated with a BackingMap plug-in when you use the COPY_TO_BYTES or COPY_TO_BYTES_RAW copy modes. This association allows data to be stored in serialized form in memory, rather than the native Java object form. Storing serialized data conserves memory and improves replication and performance on the client and server. You can use a DataSerializer plug-in to develop high-performance serialization streams that can be compressed, encrypted, evolved, and queried.

Tuning serialization

The DataSerializer plug-ins expose metadata that tells WebSphere eXtreme Scale which attributes it can and cannot directly use during serialization, the path to the data that will be serialized, and the type of data that is stored in memory. You can optimize object serialization and inflation performance so that you can efficiently interact with the byte array.

Overview

 The ObjectTransformer interface has been replaced by the DataSerializer plug-ins, which you can use to efficiently store arbitrary data in WebSphere eXtreme Scale so that existing product APIs can efficiently interact with your data.

Copies of values are always made except when the NO_COPY mode is used. The default copying mechanism that is employed in eXtreme Scale is serialization, which is known as an expensive operation. The ObjectTransformer interface is used in this situation. The ObjectTransformer interface uses callbacks to the application to provide a custom implementation of common and expensive operations, such as object serialization and deep copies on objects. However, for improved performance in most cases, you can use the DataSerializer plug-ins to serialize objects. You must use either the COPY_TO_BYTES or COPY_TO_BYTES_RAW copy modes to exploit the DataSerializer plug-ins. For more information, see [Serialization using the DataSerializer plug-ins](#).

An application can provide an implementation of the `ObjectTransformer` interface to a map, and eXtreme Scale then delegates to the methods on this object and relies on the application to provide an optimized version of each method in the interface. The `ObjectTransformer` interface follows:

```
public interface ObjectTransformer {
    void serializeKey(Object key, ObjectOutputStream stream) throws IOException;
    void serializeValue(Object value, ObjectOutputStream stream) throws IOException;
    Object inflateKey(ObjectInputStream stream) throws IOException, ClassNotFoundException;
    Object inflateValue(ObjectInputStream stream) throws IOException, ClassNotFoundException;
    Object copyValue(Object value);
    Object copyKey(Object key);
}
```

You can associate an `ObjectTransformer` interface with a `BackingMap` by using the following example code:

```
ObjectGrid g = ...;
BackingMap bm = g.defineMap("PERSON");
MyObjectTransformer ot = new MyObjectTransformer();
bm.setObjectTransformer(ot);
```

Tune object serialization and inflation

Object serialization is typically the most important performance consideration with eXtreme Scale, which uses the default serializable mechanism if an `ObjectTransformer` plug-in is not supplied by the application. An application can provide implementations of either the `Serializable` `readObject` and `writeObject`, or it can have the objects implement the `Externalizable` interface, which is approximately ten times faster. If the objects in the map cannot be modified, then an application can associate an `ObjectTransformer` interface with the `ObjectMap`. The `serialize` and `inflate` methods are provided to allow the application to provide custom code to optimize these operations, given their large performance impact on the system. The `serialize` method serializes the object to the provided stream. The `inflate` method provides the input stream and expects the application to create the object, inflate it using data in the stream and return the object. Implementations of the `serialize` and `inflate` methods must mirror each other.

The `DataSerializer` plug-ins replace the `ObjectTransformer` plug-ins, which are deprecated. To serialize your data in the most efficient way, use the `DataSerializer` plug-ins to improve performance in most cases. For example, if you intend to use functions, such as `query` and `indexing`, then you can immediately take advantage of the performance improvement that the `DataSerializer` plug-ins yield without making configuration or programmatic changes to your application code.

Tuning query performance

To tune the performance of your queries, use the following techniques and tips.

Using parameters

When a query runs, the query string must be parsed and a plan developed to run the query, both of which can be costly. WebSphere eXtreme Scale caches query plans by the query string. Since the cache is a finite size, it is important to reuse query strings whenever possible. Using named or positional parameters also helps performance by fostering query plan reuse.

```
Positional Parameter Example Query q = em.createQuery("select c from
Customer c where c.surname=?1"); q.setParameter(1, "Claus");
```

Using indexes

Proper indexing on a map might have a significant impact on query performance, even though indexing has some overhead on overall map performance. Without indexing on object attributes involved in queries, the query engine performs a table scan for each attribute. The table scan is the most expensive operation during a query run. Indexing on object attributes that are involved in queries allow the query engine to avoid an unnecessary table scan, improving the overall query performance. If the application is designed to use query intensively on a read-most map, configure indexes for object attributes that are involved in the query. If the map is mostly updated, then you must balance between query performance improvement and indexing overhead on the map.

When plain old Java objects (POJO) are stored in a map, proper indexing can avoid a Java reflection. In the following example, query replaces the WHERE clause with range index search, if the budget field has an index built over it. Otherwise, query scans the entire map and evaluates the WHERE clause by first getting the budget using Java reflection and then comparing the budget with the value 50000:

```
SELECT d FROM DeptBean d WHERE d.budget=50000
```

See “Query plan” for details on how to best tune individual queries and how different syntax, object models and indexes can affect query performance.

Using pagination

In client-server environments, the query engine transports the entire result map to the client. The data that is returned should be divided into reasonable chunks. The EntityManager Query and ObjectMap ObjectQuery interfaces both support the setFirstResult and setMaxResults methods that allow the query to return a subset of the results.

Return primitive values instead of entities

With the EntityManager Query API, entities are returned as query parameters. The query engine currently returns the keys for these entities to the client. When the client iterates over these entities using the Iterator from the getResultIterator method, each entity is automatically inflated and managed as if it were created with the find method on the EntityManager interface. The entire entity graph is built from the entity ObjectMap on the client. The entity value attributes and any related entities are eagerly resolved.

To avoid building the costly graph, modify the query to return the individual attributes with path navigation.

For example:

```
// Returns an entity  
SELECT p FROM Person p  
// Returns attributes SELECT p.name, p.address.street, p.address.city, p.gender FROM Person p
```

Query plan

All eXtreme Scale queries have a query plan. The plan describes how the query engine interacts with ObjectMaps and indexes. Display the query plan to determine if the query string or indexes are being used appropriately. The query plan can also be used to explore the differences that subtle changes in a query string make in the way eXtreme Scale runs a query.

The query plan can be viewed one of two ways:

- EntityManager Query or ObjectQuery getPlan API methods
- ObjectGrid diagnostic trace

getPlan method

The getPlan method on the ObjectQuery and Query interfaces return a String that describes the query plan. This string can be displayed to standard output or a log to display a query plan.

Note: In a distributed environment, the getPlan method does not run against the server and does not reflect any defined indexes. To view the plan, use an agent to view the plan on the server.

Query plan trace

The query plan can be displayed using ObjectGrid trace. To enable query plan trace, use the following trace specification:

```
QueryEnginePlan=debug=enabled
```

See “Collecting trace” on page 469 for details on how to enable trace and locate the trace log files.

Query plan examples

Query plan uses the word for to indicate that the query is iterating through an ObjectMap collection or through a derived collection such as: q2.getEmps(), q2.dept, or a temporary collection returned by an inner loop. If the collection is from an ObjectMap, the query plan shows whether a sequential scan (denoted by INDEX SCAN), unique or non-unique index is used. Query plan uses a filter string to list the condition expressions applied to a collection.

A Cartesian product is not commonly used in object query. The following query scans the entire EmpBean map in the outer loop and scans the entire DeptBean map in the inner loop:

```
SELECT e, d FROM EmpBean e, DeptBean d
```

Plan trace:

```
for q2 in EmpBean ObjectMap using INDEX SCAN
  for q3 in DeptBean ObjectMap using INDEX SCAN
    returning new Tuple( q2, q3 )
```

The following query retrieves all employee names from a particular department by sequentially scanning the EmpBean map to get an employee object. From the employee object, the query navigates to its department object and applies the d.no=1 filter. In this example, each employee has only one department object reference, so the inner loop runs one time:

```
SELECT e.name FROM EmpBean e JOIN e.dept d WHERE d.no=1
```

Plan trace:

```
for q2 in EmpBean ObjectMap using INDEX SCAN
  for q3 in q2.dept
    filter ( q3.getNo() = 1 )
    returning new Tuple( q2.name )
```

The following query is equivalent to the previous query. However, the following query performs better because it first narrows the result down to one department object by using the unique index that is defined over the DeptBean primary key field number. From the department object, the query navigates to its employee objects to get their names:

```
SELECT e.name FROM DeptBean d JOIN d.emps e WHERE d.no=1
```

Plan trace:

```
for q2 in DeptBean ObjectMap using UNIQUE INDEX key=(1)
  for q3 in q2.getEmps()
    returning new Tuple( q3.name )
```

The following query finds all the employees that work for development or sales. The query scans the entire EmpBean map and performs additional filtering by evaluating the expressions: d.name = 'Sales' or d.name='Dev'

```
SELECT e FROM EmpBean e, in (e.dept) d WHERE d.name = 'Sales'
or d.name='Dev'
```

Plan trace:

```
for q2 in EmpBean ObjectMap using INDEX SCAN
  for q3 in q2.dept
    filter (( q3.getName() = Sales ) OR ( q3.getName() = Dev ) )
    returning new Tuple( q2 )
```

The following query is equivalent to the previous query, but this query runs a different query plan and uses the range index built over the field name. In general, this query performs better because the index over the name field is used for narrowing down the department objects, which run quickly if only a few departments are development or sales.

```
SELECT e FROM DeptBean d, in(d.emps) e WHERE d.name='Dev' or d.name='Sales'
```

Plan trace:

IteratorUnionIndex of

```
for q2 in DeptBean ObjectMap using INDEX on name = (Dev)
  for q3 in q2.getEmps()
```

```
for q2 in DeptBean ObjectMap using INDEX on name = (Sales)
  for q3 in q2.getEmps()
```

The following query finds departments that do not have any employees:

```
SELECT d FROM DeptBean d WHERE NOT EXISTS(select e from d.emps e)
```

Plan trace:

```
for q2 in DeptBean ObjectMap using INDEX SCAN
  filter ( NOT EXISTS ( correlated collection defined as
    for q3 in q2.getEmps()
      returning new Tuple( q3 )
    returning new Tuple( q2 )
```

The following query is equivalent to the previous query but uses the SIZE scalar function. This query has similar performance but is easier to write.

```

SELECT d FROM DeptBean d WHERE SIZE(d.emps)=0
for q2 in DeptBean ObjectMap using INDEX SCAN
    filter (SIZE( q2.getEmps()) = 0 )
    returning new Tuple( q2 )

```

The following example is another way of writing the same query as the previous query with similar performance, but this query is easier to write as well:

```

SELECT d FROM DeptBean d WHERE d.emps is EMPTY

```

Plan trace:

```

for q2 in DeptBean ObjectMap using INDEX SCAN
    filter ( q2.getEmps() IS EMPTY )
    returning new Tuple( q2 )

```

The following query finds any employees with a home address matching at least one of the addresses of the employee whose name equals the value of the parameter. The inner loop has no dependency on the outer loop. The query runs the inner loop one time.

```

SELECT e FROM EmpBean e WHERE e.home = any (SELECT e1.home FROM EmpBean e1
WHERE e1.name=?1)
for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( q2.home =ANY      temp collection defined as

        for q3 in EmpBean ObjectMap using INDEX on name = ( ?1)
        returning new Tuple( q3.home      )
    )
    returning new Tuple( q2 )

```

The following query is equivalent to the previous query, but has a correlated subquery; also, the inner loop runs repeatedly.

```

SELECT e FROM EmpBean e WHERE EXISTS(SELECT e1 FROM EmpBean e1 WHERE
e.home=e1.home and e1.name=?1)

```

Plan trace:

```

for q2 in EmpBean ObjectMap using INDEX SCAN
    filter ( EXISTS (      correlated collection defined as

        for q3 in EmpBean ObjectMap using INDEX on name = (?1)
        filter ( q2.home = q3.home )
        returning new Tuple( q3      )

    )

    returning new Tuple( q2 )

```

Query optimization using indexes

Defining and using indexes properly can significantly improve query performance.

WebSphere eXtreme Scale queries can use built-in HashIndex plug-ins to improve performance of queries. Indexes can be defined on entity or object attributes. The query engine will automatically use the defined indexes if its WHERE clause uses one of the following strings:

- A comparison expression with the following operators: =, <, >, <= or >= (any comparison expressions except not equals <>)
- A BETWEEN expression
- Operands of the expressions are constants or simple terms

Requirements

Indexes have the following requirements when used by Query:

- All indexes must use the built-in HashIndex plug-in.
- All indexes must be statically defined. Dynamic indexes are not supported.
- The @Index annotation may be used to automatically create static HashIndex plug-ins.
- All single-attribute indexes must have the RangeIndex property set to true.
- All composite indexes must have the RangeIndex property set to false.
- All association (relationship) indexes must have the RangeIndex property set to false.

For information about configuring the HashIndex, refer to “Plug-ins for indexing data” on page 304.

For information regarding indexing, see “Indexing” on page 103.

For a more efficient way to search for cached objects, see “Using a composite index” on page 311

Using hints to choose an index

An index can be manually selected using the setHint method on the Query and ObjectQuery interfaces with the HINT_USEINDEX constant. This can be helpful when optimizing a query to use the best performing index.

Query examples that use attribute indexes

The following examples use simple terms: e.empid, e.name, e.salary, d.name, d.budget and e.isManager. The examples assume that indexes are defined over the name, salary and budget fields of an entity or value object. The empid field is a primary key and isManager has no index defined.

The following query uses both indexes over the fields of name and salary. It returns all employees with names that equal the value of the first parameter or a salary equal to the value of the second parameter:

```
SELECT e FROM EmpBean e where e.name=?1 or e.salary=?2
```

The following query uses both indexes over the fields of name and budget. The query returns all departments named 'DEV' with a budget that is greater than 2000.

```
SELECT d FROM DeptBean dwhere d.name='DEV' and d.budget>2000
```

The following query returns all employees with a salary greater than 3000 and with an isManager flag value that equals the value of the parameter. The query uses the index that is defined over the salary field and performs additional filtering by evaluating the comparison expression: e.isManager=?1.

```
SELECT e FROM EmpBean e where e.salary>3000 and e.isManager=?1
```

The following query finds all employees who earn more than the first parameter, or any employee that is a manager. Although the salary field has an index defined,

query scans the built-in index that is built over the primary keys of the EmpBean field and evaluates the expression: e.salary>?1 or e.isManager=TRUE.

```
SELECT e FROM EmpBean e WHERE e.salary>?1 or e.isManager=TRUE
```

The following query returns employees with a name that contains the letter a. Although the name field has an index defined, query does not use the index because the name field is used in the LIKE expression.

```
SELECT e FROM EmpBean e WHERE e.name LIKE '%a%'
```

The following query finds all employees with a name that is not "Smith". Although the name field has an index defined, query does not use the index because the query uses the not equals (<>) comparison operator.

```
SELECT e FROM EmpBean e where e.name<>'Smith'
```

The following query finds all departments with a budget less than the value of the parameter, and with an employee salary greater than 3000. The query uses an index for the salary, but it does not use an index for the budget because dept.budget is not a simple term. The dept objects are derived from collection e. You do not need to use the budget index to look for dept objects.

```
SELECT dept from EmpBean e, in (e.dept) dept where e.salary>3000 and dept.budget<?
```

The following query finds all employees with a salary greater than the salary of the employees that have the empid of 1, 2, and 3. The index salary is not used because the comparison involves a subquery. The empid is a primary key, however, and is used for a unique index search because all the primary keys have a built-in index defined.

```
SELECT e FROM EmpBean e WHERE e.salary > ALL (SELECT e1.salary FROM EmpBean e1 WHERE e1.empid=1 or e1.empid =2 or e1.empid=99)
```

To check if the index is being used by the query, you can view the "Query plan" on page 410. Here is an example query plan for the previous query:

```
for q2 in EmpBean ObjectMap using INDEX SCAN
  filter ( q2.salary >ALL temp collection defined as
    IteratorUnionIndex of
      for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(1)
      )
      for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(2)
      )
      for q3 in EmpBean ObjectMap using UNIQUE INDEX key=(99)
      )
  returning new Tuple( q3.salary )
returning new Tuple( q2 )

for q2 in EmpBean ObjectMap using RANGE INDEX on salary with range(3000,)
  for q3 in q2.dept
    filter ( q3.budget < ?1 )
  returning new Tuple( q3 )
```

Indexing attributes

Indexes can be defined over any single attribute type with the constraints previously defined.

Defining entity indexes using @Index

To define an index on an entity, simply define an annotation:

Entities using annotations

```
@Entity
public class Employee {
    @Id int empid;
    @Index String name
    @Index double salary
    @ManyToOne Department dept;
}
@Entity
public class Department {
    @Id int deptid;
    @Index String name;
    @Index double budget;
    boolean isManager;
    @OneToMany Collection<Employee> employees;
}
```

With XML

Indexes can also be defined using XML:

Entities without annotations

```
public class Employee {
    int empid;
    String name
    double salary
    Department dept;
}

public class Department {
    int deptid;
    String name;
    double budget;
    boolean isManager;
    Collection employees;
}
```

ObjectGrid XML with attribute indexes

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
<objectGrid name="DepartmentGrid" entityMetadataXMLFile="entity.xml">
<backingMap name="Employee" pluginCollectionRef="Emp"/>
<backingMap name="Department" pluginCollectionRef="Dept"/>
</objectGrid>
</objectGrids>
<backingMapPluginCollections>
<backingMapPluginCollection id="Emp">
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Employee.name"/>
<property name="AttributeName" type="java.lang.String" value="name"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Employee.salary"/>
<property name="AttributeName" type="java.lang.String" value="salary"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
```

```

</bean>
</backingMapPluginCollection>
<backingMapPluginCollection id="Dept">
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Department.name"/>
<property name="AttributeName" type="java.lang.String" value="name"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Department.budget"/>
<property name="AttributeName" type="java.lang.String" value="budget"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
</backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>

```

Entity XML

```

<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ./emd.xsd">

<description>Department entities</description>
<entity class-name="acme.Employee" name="Employee" access="FIELD">
<attributes>
<id name="empid" />
<basic name="name" />
<basic name="salary" />
<many-to-one name="department"
target-entity="acme.Department"
fetch="EAGER">
<cascade><cascade-persist/></cascade>
</many-to-one>
</attributes>
</entity>
<entity class-name="acme.Department" name="Department" access="FIELD">
<attributes>
<id name="deptid" />
<basic name="name" />
<basic name="budget" />
<basic name="isManager" />
<one-to-many name="employees"
target-entity="acme.Employee"
fetch="LAZY" mapped-by="parentNode">
<cascade><cascade-persist/></cascade>
</one-to-many>
</attributes>
</entity>
</entity-mappings>

```

Defining indexes for non-entities using XML

Indexes for non-entity types are defined in XML. There is no difference when creating the MapIndexPlugin for entity maps and non-entity maps.

Java bean

```

public class Employee {
    int empid;
    String name;
    double salary;
    Department dept;

    public class Department {
        int deptid;
        String name;
        double budget;
        boolean isManager;
        Collection employees;
    }
}

```

ObjectGrid XML with attribute indexes

```

<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
xmlns="http://ibm.com/ws/objectgrid/config">
<objectGrids>
<objectGrid name="DepartmentGrid">
<backingMap name="Employee" pluginCollectionRef="Emp"/>
<backingMap name="Department" pluginCollectionRef="Dept"/>

```

```

<querySchema>
<mapSchemas>
<mapSchema mapName="Employee" valueClass="acme.Employee"
primaryKeyField="empid" />
<mapSchema mapName="Department" valueClass="acme.Department"
primaryKeyField="deptid" />
</mapSchemas>
<relationships>
<relationship source="acme.Employee"
target="acme.Department"
relationField="dept" invRelationField="employees" />
</relationships>
</querySchema>
</objectGrid>
</objectGrids>
<backingMapPluginCollections>
<backingMapPluginCollection id="Emp">
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Employee.name"/>
<property name="AttributeName" type="java.lang.String" value="name"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Employee.salary"/>
<property name="AttributeName" type="java.lang.String" value="salary"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
</backingMapPluginCollection>
<backingMapPluginCollection id="Dept">
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Department.name"/>
<property name="AttributeName" type="java.lang.String" value="name"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
<bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
<property name="Name" type="java.lang.String" value="Department.budget"/>
<property name="AttributeName" type="java.lang.String" value="budget"/>
<property name="RangeIndex" type="boolean" value="true"
description="Ranges are must be set to true for attributes." />
</bean>
</backingMapPluginCollection>
</backingMapPluginCollections>
</objectGridConfig>

```

Indexing relationships

WebSphere eXtreme Scale stores the foreign keys for related entities within the parent object. For entities, the keys are stored in the underlying tuple. For non-entity objects, the keys are explicitly stored in the parent object.

Adding an index on a relationship attribute can speed up queries that use cyclical references or use the IS NULL, IS EMPTY, SIZE and MEMBER OF query filters. Both single- and multi-valued associations may have the @Index annotation or a HashIndex plug-in configuration in an ObjectGrid descriptor XML file.

Defining entity relationship indexes using @Index

The following example defines entities with @Index annotations:

Entity with annotation

```

@Entity
public class Node {
    @ManyToOne @Index
    Node parentNode;

    @OneToMany @Index
    List<Node> childrenNodes = new ArrayList();

    @OneToMany @Index
    List<BusinessUnitType> businessUnitTypes = new ArrayList();
}

```

Defining entity relationship indexes using XML

The following example defines the same entities and indexes using XML with HashIndex plug-ins:

Entity without annotations

```
public class Node {
    int nodeId;
    Node parentNode;
    List<Node> childrenNodes = new ArrayList();
    List<BusinessUnitType> businessUnitTypes = new ArrayList();
}
```

ObjectGrid XML

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="ObjectGrid_Entity" entityMetadataXMLFile="entity.xml">
      <backingMap name="Node" pluginCollectionRef="Node"/>
      <backingMap name="BusinessUnitType" pluginCollectionRef="BusinessUnitType"/>
    </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
    <backingMapPluginCollection id="Node">
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="parentNode"/>
        <property name="AttributeName" type="java.lang.String" value="parentNode"/>
        <property name="RangeIndex" type="boolean" value="false"
description="Ranges are not supported for association indexes." />
      </bean>
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="businessUnitType"/>
        <property name="AttributeName" type="java.lang.String" value="businessUnitTypes"/>
        <property name="RangeIndex" type="boolean" value="false"
description="Ranges are not supported for association indexes." />
      </bean>
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

Entity XML

```
<?xml version="1.0" encoding="UTF-8"?>
<entity-mappings xmlns="http://ibm.com/ws/projector/config/emd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://ibm.com/ws/projector/config/emd ../emd.xsd">
  <description>My entities</description>
  <entity class-name="acme.Node" name="Account" access="FIELD">
    <attributes>
      <id name="nodeId" />
      <one-to-many name="childrenNodes"
target-entity="acme.Node"
fetch="EAGER" mapped-by="parentNode">
        <cascade><cascade-all/></cascade>
      </one-to-many>
      <many-to-one name="parentNodes"
target-entity="acme.Node"
fetch="LAZY" mapped-by="childrenNodes">
        <cascade><cascade-none/></cascade>
      </many-to-one>
      <many-to-one name="businessUnitTypes"
target-entity="acme.BusinessUnitType"
fetch="EAGER">
        <cascade><cascade-persist/></cascade>
      </many-to-one>
    </attributes>
  </entity>
  <entity class-name="acme.BusinessUnitType" name="BusinessUnitType" access="FIELD">
    <attributes>
      <id name="buId" />
      <basic name="TypeDescription" />
    </attributes>
  </entity>
</entity-mappings>
```

Using the previously defined indexes, the following entity query examples are optimized:

```
SELECT n FROM Node n WHERE n.parentNode is null
SELECT n FROM Node n WHERE n.businessUnitTypes is EMPTY
SELECT n FROM Node n WHERE size(n.businessUnitTypes)>=10
SELECT n FROM BusinessUnitType b, Node n WHERE b member of n.businessUnitTypes and b.name='TELECOM'
```

Defining non-entity relationship indexes

The following example defines a HashIndex plug-in for non-entity maps in an ObjectGrid descriptor XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<objectGridConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://ibm.com/ws/objectgrid/config ../objectGrid.xsd"
  xmlns="http://ibm.com/ws/objectgrid/config">
  <objectGrids>
    <objectGrid name="ObjectGrid_POJO">
      <backingMap name="Node" pluginCollectionRef="Node"/>
      <backingMap name="BusinessUnitType" pluginCollectionRef="BusinessUnitType"/>
      <querySchema>
        <mapSchemas>
          <mapSchema mapName="Node"
            valueClass="com.ibm.websphere.objectgrid.samples.entity.Node"
            primaryKeyField="id" />
          <mapSchema mapName="BusinessUnitType"
            valueClass="com.ibm.websphere.objectgrid.samples.entity.BusinessUnitType"
            primaryKeyField="id" />
        </mapSchemas>
        <relationships>
          <relationship source="com.ibm.websphere.objectgrid.samples.entity.Node"
            target="com.ibm.websphere.objectgrid.samples.entity.Node"
            relationField="parentNodeId" invRelationField="childrenNodeIds" />
          <relationship source="com.ibm.websphere.objectgrid.samples.entity.Node"
            target="com.ibm.websphere.objectgrid.samples.entity.BusinessUnitType"
            relationField="businessUnitTypeKeys" invRelationField="" />
        </relationships>
      </querySchema>
    </objectGrid>
  </objectGrids>
  <backingMapPluginCollections>
    <backingMapPluginCollection id="Node">
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="parentNode"/>
      </bean>
    <backingMapPluginCollection id="BusinessUnitType">
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="parentNodeId"/>
        <property name="AttributeName" type="java.lang.String" value="parentNodeId"/>
        <property name="RangeIndex" type="boolean" value="false"
          description="Ranges are not supported for association indexes." />
      </bean>
    <backingMapPluginCollection id="childrenNodeIds">
      <bean id="MapIndexPlugin" className="com.ibm.websphere.objectgrid.plugins.index.HashIndex">
        <property name="Name" type="java.lang.String" value="childrenNodeIds"/>
        <property name="AttributeName" type="java.lang.String" value="childrenNodeIds"/>
        <property name="RangeIndex" type="boolean" value="false"
          description="Ranges are not supported for association indexes." />
      </bean>
    </backingMapPluginCollection>
  </backingMapPluginCollections>
</objectGridConfig>
```

Given the above index configurations, the following object query examples are optimized:

```
SELECT n FROM Node n WHERE n.parentNodeId is null
SELECT n FROM Node n WHERE n.businessUnitTypeKeys is EMPTY
SELECT n FROM Node n WHERE size(n.businessUnitTypeKeys)>=10
SELECT n FROM BusinessUnitType b, Node n WHERE
  b member of n.businessUnitTypeKeys and b.name='TELECOM'
```

Tuning EntityManager interface performance

The EntityManager interface separates applications from the state held in its server grid data store.

The cost of using the EntityManager interface is not high and depends on the type of work being performed. Always use the EntityManager interface and optimize the crucial business logic after the application is complete. You can rework any code that uses EntityManager interfaces to use maps and tuples. Generally, this code rework might be necessary for 10 percent of the code.

If you use relationships between objects, then the performance impact is lower because an application that is using maps needs to manage those relationships similarly to the EntityManager interface.

Applications that use the EntityManager interface do not need to provide an ObjectTransformer implementation. The applications are optimized automatically.

Reworking EntityManager code for maps

A sample entity follows:

```
@Entity
public class Person
{
    @Id
    String ssn;
    String firstName;
    @Index
    String middleName;
    String surname;
}
```

Some code to find the entity and update the entity follows:

```
Person p = null;
s.begin();
p = (Person)em.find(Person.class, "1234567890");
p.middleName = String.valueOf(inner);
s.commit();
```

The same code using Maps and Tuples follows:

```
Tuple key = null;
key = map.getEntityMetadata().getKeyMetadata().createTuple();
key.setAttribute(0, "1234567890");
```

```
// The Copy Mode is always NO_COPY for entity maps if not using COPY_TO_BYTES.
// Either we need to copy the tuple or we can ask the ObjectGrid to do it for us:
map.setCopyMode(CopyMode.COPY_ON_READ);
s.begin();
Tuple value = (Tuple)map.get(key);
value.setAttribute(1, String.valueOf(inner));
map.update(key, value);
value = null;
s.commit();
```

Both of these code snippets have the same result, and an application can use either or both snippets.

The second code snippet shows how to use maps directly and how to work with the tuples (the key and value pairs). The value tuple has three attributes: **firstName**, **middleName**, and **surname**, indexed at 0, 1, and 2. The key tuple has a single attribute the ID number is indexed at zero. You can see how Tuples are created by using the EntityMetadata#getKeyMetadata or EntityMetadata#getValueMetadata methods. You must use these methods to create Tuples for an Entity. You cannot implement the Tuple interface and pass an instance of your Tuple implementation.

Entity performance instrumentation agent

You can improve the performance of field-access entities by enabling the WebSphere eXtreme Scale instrumentation agent when using Java Development Kit (JDK) Version 1.5 or later.

Enabling eXtreme Scale agent on JDK Version 1.5 or above

The ObjectGrid agent can be enabled with a Java command line option with the following syntax:

```
-javaagent:jarpath[=options]
```

The *jarpath* value is the path to an eXtreme Scale runtime Java archive (JAR) file that contains eXtreme Scale agent class and supporting classes such as the `objectgrid.jar`, `wsobjectgrid.jar`, `ogclient.jar`, `wsogclient.jar`, and `ogagent.jar` files. Typically, in a stand-alone Java program or in a Java Platform, Enterprise Edition environment that is not running WebSphere Application Server, use the `objectgrid.jar` or `ogclient.jar` file. In a WebSphere Application Server or a multi-classloaders environment, you must use the `ogagent.jar` file in the Java command line agent option. Provide the `ogagent.config` file in the class path or use agent options to specify additional information.

eXtreme Scale agent options

config

Overrides the configuration file name.

include

Specifies or overrides transformation domain definition that is the first part of the configuration file.

exclude

Specifies or overrides the @Exclude definition.

fieldAccessEntity

Specifies or overrides the @FieldAccessEntity definition.

trace Specifies a trace level. Levels can be ALL, CONFIG, FINE, FINER, FINEST, SEVERE, WARNING, INFO, and OFF.

trace.file

Specifies the location of the trace file.

The semicolon (;) is used as a delimiter to separate each option. The comma (,) is used as a delimiter to separate each element within an option. The following example demonstrates the eXtreme Scale agent option for a Java program:

```
-javaagent:objectgridRoot/lib/objectgrid.jar=config=myConfigFile;  
include=includedPackage;exclude=excludedPackage;  
fieldAccessEntity=package1,package2
```

ogagent.config file

The `ogagent.config` file is the designated eXtreme Scale agent configuration file name. If the file name is in the class path, the eXtreme Scale agent finds and parses the file. You can override the designated file name through the `config` option of eXtreme Scale agent. The following example shows how to specify the configuration file:

```
-javaagent:objectgridRoot/lib/objectgrid.jar=config=myOverrideConfigFile
```

An eXtreme Scale agent configuration file has the following parts:

- **Transformation domain:** The transformation domain part is first in the configuration file. The transformation domain is a list of packages and classes that are included in the class transformation process. This transformation domain must include all classes that are field-access entity classes, and other classes that refer to these field-access entity classes. Field-access entity classes and those classes that refer to these field-access entity classes construct the transformation domain. If you plan to specify field-access entity classes in the @FieldAccessEntity part, then you do not need to include field-access entity classes here. The transformation domain must be complete. Otherwise, you might see a FieldAccessEntityNotInstrumentedException exception.
- **@Exclude:** The @Exclude token indicates that packages and classes listed after this token are excluded from the transformation domain.
- **@FieldAccessEntity:** The @FieldAccessEntity token indicates that packages and classes listed after this token are field-access Entity packages and classes. If no line exists after the @FieldAccessEntity token, then its equivalent is "No @FieldAccessEntity specified". The eXtreme Scale agent determines that there are no field-access Entity packages and classes defined. If there are lines after the @FieldAccessEntity token, then they represent the user-specified field-access Entity packages and classes. For example, "field-access entity domain". The field-access entity domain is a sub-domain of the transformation domain. Packages and classes that are listed in the field-access entity domain are a part of the transformation domain, even when they are not listed in the transformation domain. The @Exclude token, which lists packages and classes that are excluded from transformation, has no impact on the field-access Entity domain. When @FieldAccessEntity token is specified, all field-access entities must be in this field-access Entity domain. Otherwise, a FieldAccessEntityNotInstrumentedException exception might occur.

Example agent configuration file (ogagent.config)

```
#####
# The # indicates comment line
#####
# This is an ObjectGrid agent config file (the designated file name is ogagent.config) that can be found and parsed by the ObjectGrid agent
# if it is in classpath.
# If the file name is "ogagent.config" and in classpath, Java program runs with -javaagent:objectgridroot/ogagent.jar will have
# ObjectGrid agent enabled.
# If the file name is not "ogagent.config" but in classpath, you can specify the file name in config option of ObjectGrid agent
#   -javaagent:objectgridroot/lib/objectgrid.jar=config=myOverrideConfigFile
# See comments below for more info regarding instrumentation setting override.

# The first part of the configuration is the list of packages and classes that should be included in transformation domain.
# The includes (packages/classes, construct the instrumentation domain) should be in the beginning of the file.
com.testpackage
com.testClass

# Transformation domain: The above lines are packages/classes that construct the transformation domain.
# The system will process classes with name starting with above packages/classes for transformation.
#
# @Exclude token : Exclude from transformation domain.
# The @Exclude token indicates packages/classes after that line should be excluded from transformation domain.
# It is used when user want to exclude some packages/classes from above specified included packages
#
# @FieldAccessEntity token: Field-access Entity domain.
# The @FieldAccessEntity token indicates packages/classes after that line are field-access Entity packages/classes.
# If there is no line after the @FieldAccessEntity token, it is equivalent to "No @FieldAccessEntity specified".
# The runtime will consider the user does not specify any field-access Entity packages/classes.
# The "field-access Entity domain" is a sub-domain of transformation domain.
#
# Packages/classes listed in the "field-access Entity domain" will always be part of transformation domain,
# even they are not listed in transformation domain.
# The @Exclude, which lists packages/classes excluded from transformation, has no impact on the "field-access Entity domain".
# Note: When @FieldAccessEntity is specified, all field-access entities must be in this field-access Entity domain,
# otherwise, FieldAccessEntityNotInstrumentedException may occur.
#
# The default ObjectGrid agent config file name is ogagent.config
# The runtime will look for this file as a resource in classpath and process it.
# Users can override this designated ObjectGrid agent config file name via config option of agent.
#
# e.g.
# javaagent:objectgridroot/lib/objectgrid.jar=config=myOverrideConfigFile
#
# The instrumentation definition, including transformation domain, @Exclude, and @FieldAccessEntity can be overridden individually
# by corresponding designated agent options.
# Designated agent options include:
# include      -> used to override instrumentation domain definition that is the first part of the config file
# exclude     -> used to override @Exclude definition
# fieldAccessEntity -> used to override @FieldAccessEntity definition
#
# Each agent option should be separated by ":",
# Within the agent option, the package or class should be separated by ","
#
# The following is an example that does not override the config file name:
# -javaagent:objectgridroot/lib/objectgrid.jar=include=includedPackage;exclude=excludedPackage;fieldAccessEntity=package1,package2
#
#####
@Exclude
```

```
com.excludedPackage  
com.excludedClass  
@FieldAccessEntity
```

Performance consideration

For better performance, specify the transformation domain and field-access entity domain.

Chapter 7. Security



WebSphere eXtreme Scale can secure data access, including allowing for integration with external security providers. Aspects of security include authentication, authorization, transport security, data grid security, local security, and JMX (MBean) security.

Configuring security profiles for the `xscmd` utility

By creating a security profile, you can use saved security parameters to use the `xscmd` utility with secure environments.

Before you begin

For more information about setting up the `xscmd` utility, see *Administering with the `xscmd` utility*.

About this task

You can use the `-ssp profile_name` or `--saveSecProfile profile_name` parameter with the rest of your `xscmd` command. to save a security profile. The profile can contain settings for user names and passwords, credential generators, keystores, truststores, and transport types.

The **ProfileManagement** command group in the `xscmd` utility contains commands for managing your security profiles.

Procedure

- Save a security profile.

To save a security profile, use the `-ssp profile_name` or `--saveSecProfile profile_name` parameter with the rest of your command. Adding this parameter to your command saves the following parameters:

```
-al,--alias <alias>
-arc,--authRetryCount <integer>
-ca,--credAuth <support>
-cgc,--credGenClass <className>
-cgp,--credGenProps <property>
-cxpv,--contextProvider <provider>
-ks,--keyStore <filePath>
-ksp,--keyStorePassword <password>
-kst,--keyStoreType <type>
-prot,--protocol <protocol>
-pwd,--password <password>
-ts,--trustStore <filePath>
-tsp,--trustStorePassword <password>
-tst,--trustStoreType <type>
-tt,--transportType <type>
-user,--username <username>
```

Security profiles are saved in the `user_home\.xscmd\profiles\security\profile_name.properties` directory.

Important: Do not include the `.properties` file name extension on the `profile_name` parameter. This extension is automatically added to the file name.

- Use a saved security profile.
To use a saved security profile, add the `-sp profile_name` or `--securityProfile profile_name` parameter to the command you are running. Command example:
`xscmd -c listHosts -cep myhost.mycompany.com -sp myprofile`
- List the commands in the **ProfileManagement** command group.
Run the following command: `xscmd -lc ProfileManagement`.
- List the existing security profiles.
Run the following command: `xscmd -c listProfiles -v`.
- Display the settings that are saved in a security profile.
Run the following command: `xscmd -c showProfile -pn profile_name`.
- Remove an existing security profile.
Run the following command: `xscmd -c RemoveProfile -pn profile_name`.

Securing J2C client connections

Use the Java 2 Connector (J2C) architecture to secure connections between WebSphere eXtreme Scale clients and your applications.

About this task

Applications reference the connection factory, which establishes the connection to the remote data grid. Each connection factory hosts a single eXtreme Scale client connection that is reused for all application components.

Important: Since the eXtreme Scale client connection might include a near cache, it is important that applications do not share a connection. A connection factory must exist for a single application instance to avoid problems sharing objects between applications.

You can set the credential generator with the API or in the client properties file. In the client properties file, the `securityEnabled` and `credentialGenerator` properties are used. The following code example is displayed on multiple lines for publication purposes:

```
securityEnabled=true
credentialGeneratorClass=com.ibm.websphere.objectgrid.security.plugins.builtins.
  UserPasswordCredentialGenerator
credentialGeneratorProps=operator XXXXXX
```

The credential generator and credential in the client properties file are used for the eXtreme Scale connect operation and the default J2C credentials. Therefore, the credentials that are specified with the API are used at J2C connect time for the J2C connection. However, if no credentials are specified at J2C connect time, then the credential generator in the client properties file is used.

Procedure

1. Set up secure access where the J2C connection represents the eXtreme Scale client. Use the `ClientPropertiesResource` connection factory property or the `ClientPropertiesURL` connection factory property to configure client authentication.
2. Configure the client security properties to use the connection factory that references the appropriate credential generator object for eXtreme Scale. These properties are also compatible with eXtreme Scale server security. For example, use the `WSTokenCredentialGenerator` credential generator for WebSphere credentials when eXtreme Scale is installed with WebSphere Application Server.

Alternatively, use the `UserPasswordCredentialGenerator` credential generator when you run the eXtreme Scale in a stand-alone environment. In the following example, credentials are passed programmatically using the API call instead of using the configuration in the client properties:

```
XSCConnectionSpec spec = new XSCConnectionSpec();
spec.setCredentialGenerator(new UserPasswordCredentialGenerator("operator", "xxxxxx"));
Connection conn = connectionFactory.getConnection(spec);
```

3. (Optional) Disable the near cache, if required.

All J2C connections from a single connection factory share a single near cache. Grid entry permissions and map permissions are validated on the server, but not on the near cache. When an application uses multiple credentials to create J2C connections, and the configuration uses specific permissions for grid entries and maps for those credentials, then disable the near cache. Disable the near cache using the connection factory property, `ObjectGridResource` or `ObjectGridURL`. Set the `numberOfBuckets` property to 0.

4. (Optional) Set security policy settings, if required.

If the J2EE application contains the embedded eXtreme Scale resource adapter archive (RAR) file configuration, you might be required to set additional security policy settings in the security policy file for the application. For example, these policies are required:

```
permission com.ibm.websphere.security.WebSphereRuntimePermission "accessRuntimeClasses";
permission java.lang.RuntimePermission "accessDeclaredMembers";
permission javax.management.MBeanTrustPermission "register";
permission java.lang.RuntimePermission "getClassLoader";
```

Additionally, any property or resource files used by connection factories require file or other permissions, such as `permission java.io.FilePermission "filePath";`. For WebSphere Application Server, the policy file is `META-INF/was.policy`, and it is located in the J2EE EAR file.

Results

The client security properties that you configured on the catalog service domain are used as default values. The values that you specify override any properties that are defined in the `client.properties` files.

What to do next

Use eXtreme Scale data access APIs to develop client components that you want to use transactions.

Programming for security

Use programming interfaces to handle various aspects of security in a WebSphere eXtreme Scale environment.

Security API

WebSphere eXtreme Scale adopts an open security architecture. It provides a basic security framework for authentication, authorization, and transport security, and requires users to implement plug-ins to complete the security infrastructure.

The following image shows the basic flow of client authentication and authorization for an eXtreme Scale server.

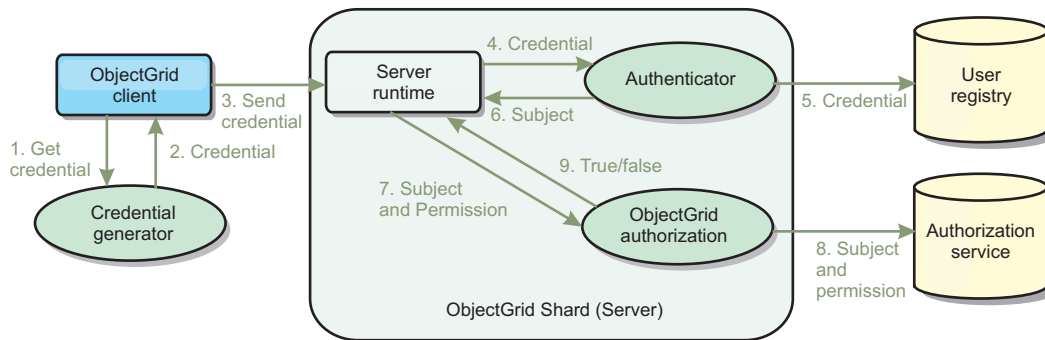


Figure 32. Flow of client authentication and authorization

The authentication flow and authorization flow are as follows.

Authentication flow

1. The authentication flow starts with an eXtreme Scale client getting a credential. This is done by the `com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator` plug-in.
2. A `CredentialGenerator` object knows how to generate a valid client credential, for example, a user ID and password pair, Kerberos ticket, and so on. This generated credential is sent back to the client.
3. After the client retrieves the `Credential` object using the `CredentialGenerator` object, this `Credential` object is sent along with the eXtreme Scale request to the eXtreme Scale server.
4. The eXtreme Scale server authenticates the `Credential` object before processing the eXtreme Scale request. Then the server uses the `Authenticator` plug-in to authenticate the `Credential` object.
5. The `Authenticator` plug-in represents an interface to the user registry, for example, a Lightweight Directory Access Protocol (LDAP) server or an operating system user registry. The `Authenticator` consults the user registry and makes authentication decisions.
6. If the authentication is successful, a `Subject` object is returned to represent this client.

Authorization flow

WebSphere eXtreme Scale adopts a permission-based authorization mechanism, and has different permission categories represented by different permission classes. For example, a `com.ibm.websphere.objectgrid.security.MapPermission` object represents permissions to read, write, insert, invalidate, and remove the data entries in an `ObjectMap`. Because WebSphere eXtreme Scale supports Java Authentication and Authorization Service (JAAS) authorization out-of-box, you can use JAAS to handle authorization by providing authorization policies.

Also, eXtreme Scale supports custom authorizations. Custom authorizations are plugged in by the plug-in `com.ibm.websphere.objectgrid.security.plugins.ObjectGridAuthorization`. The flow of the customer authorization is as follows.

7. The server runtime sends the `Subject` object and the required permission to the authorization plug-in.
8. The authorization plug-in consults the Authorization service and makes an authorization decision. If permission is granted for this `Subject` object, a value of `true` is returned, otherwise `false` is returned.
9. This authorization decision, `true` or `false`, is returned to the server runtime.

Security implementation

The topics in this section discuss how to program a secure WebSphere eXtreme Scale deployment and how to program the plug-in implementations. The section is organized based on the various security features. In each subtopic, you will learn about relevant plug-ins and how to implement the plug-ins. In the authentication section, you will see how to connect to a secure WebSphere eXtreme Scale deployment environment.

Client Authentication: The client authentication topic describes how a WebSphere eXtreme Scale client gets a credential and how a server authenticates the client. It will also discuss how a WebSphere eXtreme Scale client connects to a secure WebSphere eXtreme Scale server.

Authorization: The authorization topic explains how to use the ObjectGridAuthorization to do customer authorization besides JAAS authorization.

Grid Authentication: The data grid authentication topic discusses how you can use SecureTokenManager to securely transport server secrets.

Java Management Extensions (JMX) programming: When the WebSphere eXtreme Scale server is secured, the JMX client might need to send a JMX credential to the server.

Client authentication programming

For authentication, WebSphere eXtreme Scale provides a runtime to send the credential from the client to the server side, and then calls the authenticator plug-in to authenticate the users.

WebSphere eXtreme Scale requires you to implement the following plug-ins to complete the authentication.

- **Credential:** A Credential represents a client credential, such as a user ID and password pair.
- **CredentialGenerator:** A CredentialGenerator represents a credential factory to generate the credential.
- **Authenticator:** An Authenticator authenticates the client credential and retrieves client information.

Credential and CredentialGenerator plug-ins

When an eXtreme Scale client connects to a server that requires authentication, the client is required to provide a client credential. A client credential is represented by a `com.ibm.websphere.objectgrid.security.plugins.Credential` interface. A client credential can be a user name and password pair, a Kerberos ticket, a client certificate, or data in any format that the client and server agree upon. This interface explicitly defines the `equals(Object)` and `hashCode` methods. These two methods are important because the authenticated Subject objects are cached by using the Credential object as the key on the server side. WebSphere eXtreme Scale also provides a plug-in to generate a credential. This plug-in is represented by the `com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator` interface and is useful when the credential can expire. In this case, the `getCredential` method is called to renew a credential.

The Credential interface explicitly defines the equals(Object) and hashCode methods. These two methods are important because the authenticated Subject objects are cached by using the Credential object as the key on the server side.

You may also use the provided plug-in to generate a credential. This plug-in is represented by the com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator interface, and is useful when the credential can expire. In this case, the getCredential method is called to renew a credential. See the API documentation for more details.

There are three provided default implementations for the Credential interfaces:

- The com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential implementation, which contains a user ID and password pair.
- The com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredential implementation, which contains WebSphere Application Server-specific authentication and authorization tokens. These tokens can be used to propagate the security attributes across the application servers in the same security domain.

WebSphere eXtreme Scale also provides a plug-in to generate a credential. This plug-in is represented by the com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator interface. WebSphere eXtreme Scale provides two default built-in implementations:

- The com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator constructor takes a user ID and a password. When the getCredential method is called, it returns a UserPasswordCredential object that contains the user ID and password.
- The com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredentialGenerator represents a credential (security token) generator when running in WebSphere Application Server. When the getCredential method is called, the Subject that is associated with the current thread is retrieved. Then the security information in this Subject object is converted into a WSTokenCredential object. You can specify whether to retrieve a runAs subject or a caller subject from the thread by using the constant WSTokenCredentialGenerator.RUN_AS_SUBJECT or WSTokenCredentialGenerator.CALLER_SUBJECT.

UserPasswordCredential and UserPasswordCredentialGenerator

For testing purposes, WebSphere eXtreme Scale provides the following plug-in implementations:

1. com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredential
2. com.ibm.websphere.objectgrid.security.plugins.builtins.UserPasswordCredentialGenerator

The user password credential stores a user ID and password. The user password credential generator then contains this user ID and password.

The following example code shows how to implement these two plug-ins.

```
UserPasswordCredential.java
// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
```



```

// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007
package com.ibm.websphere.objectgrid.security.plugins.builtins;

import com.ibm.websphere.objectgrid.security.plugins.Credential;

/**
 * This class represents a credential containing a user ID and password.
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see Credential
 * @see UserPasswordCredentialGenerator#getCredential()
 */
public class UserPasswordCredential implements Credential {

    private static final long serialVersionUID = 1409044825541007228L;

    private String ivUserName;

    private String ivPassword;

    /**
     * Creates a UserPasswordCredential with the specified user name and
     * password.
     *
     * @param userName the user name for this credential
     * @param password the password for this credential
     *
     * @throws IllegalArgumentException if userName or password is <code>null</code>
     */
    public UserPasswordCredential(String userName, String password) {
        super();
        if (userName == null || password == null) {
            throw new IllegalArgumentException("User name and password cannot be null.");
        }
        this.ivUserName = userName;
        this.ivPassword = password;
    }

    /**
     * Gets the user name for this credential.
     *
     * @return the user name argument that was passed to the constructor
     *         or the <code>setUserName(String)</code>
     *         method of this class
     *
     * @see #setUserName(String)
     */
    public String getUserName() {
        return ivUserName;
    }

    /**
     * Sets the user name for this credential.
     *
     * @param userName the user name to set.
     *
     * @throws IllegalArgumentException if userName is <code>null</code>
     */
    public void setUserName(String userName) {
        if (userName == null) {
            throw new IllegalArgumentException("User name cannot be null.");
        }
        this.ivUserName = userName;
    }

    /**
     * Gets the password for this credential.
     *
     * @return the password argument that was passed to the constructor
     *         or the <code>setPassword(String)</code>
     *         method of this class
     *
     * @see #setPassword(String)
     */
    public String getPassword() {
        return ivPassword;
    }

    /**
     * Sets the password for this credential.
     *
     * @param password the password to set.
     *
     * @throws IllegalArgumentException if password is <code>null</code>
     */
    public void setPassword(String password) {
        if (password == null) {
            throw new IllegalArgumentException("Password cannot be null.");
        }
    }
}

```

```

    }
    this.ivPassword = password;
}

/**
 * Checks two UserPasswordCredential objects for equality.
 * <p>
 * Two UserPasswordCredential objects are equal if and only if their user names
 * and passwords are equal.
 *
 * @param o the object we are testing for equality with this object.
 *
 * @return <code>true</code> if both UserPasswordCredential objects are equivalent.
 *
 * @see Credential#equals(Object)
 */
public boolean equals(Object o) {
    if (this == o) {
        return true;
    }
    if (o instanceof UserPasswordCredential) {
        UserPasswordCredential other = (UserPasswordCredential) o;
        return other.ivPassword.equals(ivPassword) && other.ivUserName.equals(ivUserName);
    }
    return false;
}

/**
 * Returns the hashCode of the UserPasswordCredential object.
 *
 * @return the hash code of this object
 *
 * @see Credential#hashCode()
 */
public int hashCode() {
    return ivUserName.hashCode() + ivPassword.hashCode();
}
}

```

UserPasswordCredentialGenerator.java

```

// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007
package com.ibm.websphere.objectgrid.security.plugins.builtins;

import java.util.StringTokenizer;

import com.ibm.websphere.objectgrid.security.plugins.Credential;
import com.ibm.websphere.objectgrid.security.plugins.CredentialGenerator;

/**
 * This credential generator creates <code>UserPasswordCredential</code> objects.
 * <p>
 * UserPasswordCredentialGenerator has a one to one relationship with
 * UserPasswordCredential because it can only create a UserPasswordCredential
 * representing one identity.
 *
 * @since WAS XD 6.0.1
 * @ibm-api
 *
 * @see CredentialGenerator
 * @see UserPasswordCredential
 */
public class UserPasswordCredentialGenerator implements CredentialGenerator {

    private String ivUser;

    private String ivPwd;

    /**
     * Creates a UserPasswordCredentialGenerator with no user name or password.
     *
     * @see #setProperties(String)
     */
    public UserPasswordCredentialGenerator() {
        super();
    }

    /**
     * Creates a UserPasswordCredentialGenerator with a specified user name and
     * password
     *
     * @param user the user name
     * @param pwd the password
     */
    public UserPasswordCredentialGenerator(String user, String pwd) {

```

```

        ivUser = user;
        ivPwd = pwd;
    }

    /**
     * Creates a new <code>UserPasswordCredential</code> object using this
     * object's user name and password.
     *
     * @return a new <code>UserPasswordCredential</code> instance
     *
     * @see CredentialGenerator#getCredential()
     * @see UserPasswordCredential
     */
    public Credential getCredential() {
        return new UserPasswordCredential(ivUser, ivPwd);
    }

    /**
     * Gets the password for this credential generator.
     *
     * @return the password argument that was passed to the constructor
     */
    public String getPassword() {
        return ivPwd;
    }

    /**
     * Gets the user name for this credential.
     *
     * @return the user argument that was passed to the constructor
     *         of this class
     */
    public String getUsername() {
        return ivUser;
    }

    /**
     * Sets additional properties namely a user name and password.
     *
     * @param properties a properties string with a user name and
     *                  a password separated by a blank.
     *
     * @throws IllegalArgumentException if the format is not valid
     */
    public void setProperties(String properties) {
        StringTokenizer token = new StringTokenizer(properties, " ");
        if (token.countTokens() != 2) {
            throw new IllegalArgumentException(
                "The properties should have a user name and password and separated by a blank.");
        }

        ivUser = token.nextToken();
        ivPwd = token.nextToken();
    }

    /**
     * Checks two UserPasswordCredentialGenerator objects for equality.
     * <p>
     * Two UserPasswordCredentialGenerator objects are equal if and only if
     * their user names and passwords are equal.
     *
     * @param obj the object we are testing for equality with this object.
     *
     * @return <code>>true</code> if both UserPasswordCredentialGenerator objects
     *         are equivalent.
     */
    public boolean equals(Object obj) {
        if (obj == this) {
            return true;
        }

        if (obj != null && obj instanceof UserPasswordCredentialGenerator) {
            UserPasswordCredentialGenerator other = (UserPasswordCredentialGenerator) obj;

            boolean bothUserNull = false;
            boolean bothPwdNull = false;

            if (ivUser == null) {
                if (other.ivUser == null) {
                    bothUserNull = true;
                } else {
                    return false;
                }
            }

            if (ivPwd == null) {
                if (other.ivPwd == null) {
                    bothPwdNull = true;
                } else {
                    return false;
                }
            }

            return bothUserNull && bothPwdNull;
        }

        return false;
    }
}

```

```

        return (bothUserNull || ivUser.equals(other.ivUser)) && (bothPwdNull || ivPwd.equals(other.ivPwd));
    }
    return false;
}
/**
 * Returns the hashCode of the UserPasswordCredentialGenerator object.
 *
 * @return the hash code of this object
 */
public int hashCode() {
    return ivUser.hashCode() + ivPwd.hashCode();
}
}

```

The `UserPasswordCredential` class contains two attributes: user name and password. The `UserPasswordCredentialGenerator` serves as a factory that contains the `UserPasswordCredential` objects.

WSTokenCredential and WSTokenCredentialGenerator

When the WebSphere eXtreme Scale clients and servers are all deployed in WebSphere Application Server, the client application can use these two built-in implementations when the following conditions are satisfied:

1. WebSphere Application Server global security is turned on.
2. All WebSphere eXtreme Scale clients and servers are running in WebSphere Application Server Java virtual machines.
3. The application servers are in the same security domain.
4. The client is already authenticated in WebSphere Application Server.

In this situation, the client can use the `com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredentialGenerator` class to generate a credential. The server uses the `WSAuthenticator` implementation class to authenticate the credential.

This scenario takes advantage of the fact that the eXtreme Scale client has already been authenticated. Because the application servers that have the servers are in the same security domain as the application servers that house the clients, the security tokens can be propagated from the client to the server so that the same user registry does not need to be authenticated again.

Note: Do not assume that a `CredentialGenerator` always generates the same credential. For an expirable and refreshable credential, the `CredentialGenerator` should be able to generate the latest valid credential to make sure the authentication succeeds. One example is using the Kerberos ticket as a `Credential` object. When the Kerberos ticket refreshes, the `CredentialGenerator` should retrieve the refreshed ticket when `CredentialGenerator.getCredential` is called.

Authenticator plug-in

After the eXtreme Scale client retrieves the `Credential` object using the `CredentialGenerator` object, this client `Credential` object is sent along with the client request to the eXtreme Scale server. The server authenticates the `Credential` object before processing the request. If the `Credential` object is authenticated successfully, a `Subject` object is returned to represent this client.

This `Subject` object is then cached, and it expires after its lifetime reaches the session timeout value. The login session timeout value can be set by using the

loginSessionExpirationTime property in the cluster XML file. For example, setting loginSessionExpirationTime="300" makes the Subject object expire in 300 seconds.

This Subject object is then used for authorizing the request, which is shown later. An eXtreme Scale server uses the Authenticator plug-in to authenticate the Credential object. See the information about the Authenticator in the API documentation for more details.

The Authenticator plug-in is where the eXtreme Scale runtime authenticates the Credential object from the client user registry, for example, a Lightweight Directory Access Protocol (LDAP) server.

WebSphere eXtreme Scale does not provide an immediately available user registry configuration. The configuration and management of user registry is left outside of WebSphere eXtreme Scale for simplicity and flexibility. This plug-in implements connecting and authenticating to the user registry. For example, an Authenticator implementation extracts the user ID and password from the credential, uses them to connect and validate to an LDAP server, and creates a Subject object as a result of the authentication. The implementation might use JAAS login modules. A Subject object is returned as a result of authentication.

Notice that this method creates two exceptions: InvalidCredentialException and ExpiredCredentialException. The InvalidCredentialException exception indicates that the credential is not valid. The ExpiredCredentialException exception indicates that the credential expired. If one of these two exceptions result from the authenticate method, the exceptions are sent back to the client. However, the client runtime handles these two exceptions differently:

- If the error is an InvalidCredentialException exception, the client run time displays this exception. Your application must handle the exception. You can correct the CredentialGenerator, for example, and then retry the operation.
- If the error is an ExpiredCredentialException exception, and the retry count is not 0, the client run time calls the CredentialGenerator.getCredential method again, and sends the new Credential object to the server. If the new credential authentication succeeds, the server processes the request. If the new credential authentication fails, the exception is sent back to the client. If the number of authentication retries reaches the supported value and the client still gets an ExpiredCredentialException exception, the ExpiredCredentialException exception results. Your application must handle the error.

The Authenticator interface provides great flexibility. You can implement the Authenticator interface in your own specific way. For example, you can implement this interface to support two different user registries.

WebSphere eXtreme Scale provides sample authenticator plug-in implementations. Except for the WebSphere Application Server authenticator plug-in, the other implementations are only samples for testing purposes.

KeyStoreLoginAuthenticator

This example uses an eXtreme Scale built-in implementation: KeyStoreLoginAuthenticator, which is for testing and sample purposes (a key store is a simple user registry and should not be used for a production environment). Again, the class is displayed to further demonstrate how to implement an authenticator.

```

KeyStoreLoginAuthenticator.java
// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007

package com.ibm.websphere.objectgrid.security.plugins.builtins;

import javax.security.auth.Subject;
import javax.security.auth.login.LoginContext;
import javax.security.auth.login.LoginException;

import com.ibm.websphere.objectgrid.security.plugins.Authenticator;
import com.ibm.websphere.objectgrid.security.plugins.Credential;
import com.ibm.websphere.objectgrid.security.plugins.ExpiredCredentialException;
import com.ibm.websphere.objectgrid.security.plugins.InvalidCredentialException;
import com.ibm.ws.objectgrid.Constants;
import com.ibm.ws.objectgrid.ObjectGridManagerImpl;
import com.ibm.ws.objectgrid.security.auth.callback.UserPasswordCallbackHandlerImpl;

/**
 * This class is an implementation of the <code>Authenticator</code> interface
 * when a user name and password are used as a credential.
 * <p>
 * When user ID and password authentication is used, the credential passed to the
 * <code>authenticate(Credential)</code> method is a UserPasswordCredential object.
 * <p>
 * This implementation will use a <code>KeyStoreLoginModule</code> to authenticate
 * the user into the key store using the JAAS login module "KeyStoreLogin". The key
 * store can be configured as an option to the <code>KeyStoreLoginModule</code>
 * class. Please see the <code>KeyStoreLoginModule</code> class for more details
 * about how to set up the JAAS login configuration file.
 * <p>
 * This class is only for sample and quick testing purpose. Users should
 * write your own Authenticator implementation which can fit better into
 * the environment.
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see Authenticator
 * @see KeyStoreLoginModule
 * @see UserPasswordCredential
 */
public class KeyStoreLoginAuthenticator implements Authenticator {

    /**
     * Creates a new KeyStoreLoginAuthenticator.
     */
    public KeyStoreLoginAuthenticator() {
        super();
    }

    /**
     * Authenticates a <code>UserPasswordCredential</code>.
     * <p>
     * Uses the user name and password from the specified UserPasswordCredential
     * to login to the KeyStoreLoginModule named "KeyStoreLogin".
     *
     * @throws InvalidCredentialException if credential isn't a
     *         UserPasswordCredential or some error occurs during processing
     *         of the supplied UserPasswordCredential
     *
     * @throws ExpiredCredentialException if credential is expired. This exception
     *         is not used by this implementation
     *
     * @see Authenticator#authenticate(Credential)
     * @see KeyStoreLoginModule
     */
    public Subject authenticate(Credential credential) throws InvalidCredentialException,
        ExpiredCredentialException {

        if (credential == null) {
            throw new InvalidCredentialException("Supplied credential is null");
        }

        if (!(credential instanceof UserPasswordCredential)) {
            throw new InvalidCredentialException("Supplied credential is not a UserPasswordCredential");
        }

        UserPasswordCredential cred = (UserPasswordCredential) credential;
        LoginContext lc = null;
        try {
            lc = new LoginContext("KeyStoreLogin",
                new UserPasswordCallbackHandlerImpl(cred.getUserName(), cred.getPassword().toCharArray()));
        } catch (LoginException e) {
            throw new InvalidCredentialException("Login failed: " + e.getMessage());
        }

        lc.login();
    }
}

```

```

        Subject subject = lc.getSubject();

        return subject;
    }
    catch (LoginException le) {
        throw new InvalidCredentialException(le);
    }
    catch (IllegalArgumentException ile) {
        throw new InvalidCredentialException(ile);
    }
}
}
}

```

KeyStoreLoginModule.java

```

// This sample program is provided AS IS and may be used, executed, copied and modified
// without royalty payment by customer
// (a) for its own instruction and study,
// (b) in order to develop applications designed to run with an IBM WebSphere product,
// either for customer's own internal use or for redistribution by customer, as part of such an
// application, in customer's own products.
// Licensed Materials - Property of IBM
// 5724-J34 © COPYRIGHT International Business Machines Corp. 2007
package com.ibm.websphere.objectgrid.security.plugins.builtins;

import java.io.File;
import java.io.FileInputStream;
import java.security.KeyStore;
import java.security.KeyStoreException;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.UnrecoverableKeyException;
import java.security.cert.Certificate;
import java.security.cert.CertificateException;
import java.security.cert.CertificateFactory;
import java.security.cert.X509Certificate;
import java.util.Arrays;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

import javax.security.auth.Subject;
import javax.security.auth.callback.Callback;
import javax.security.auth.callback.CallbackHandler;
import javax.security.auth.callback.NameCallback;
import javax.security.auth.callback.PasswordCallback;
import javax.security.auth.login.LoginException;
import javax.security.auth.spi.LoginModule;
import javax.security.auth.x500.X500Principal;
import javax.security.auth.x500.X500PrivateCredential;

import com.ibm.websphere.objectgrid.ObjectGridRuntimeException;
import com.ibm.ws.objectgrid.Constants;
import com.ibm.ws.objectgrid.ObjectGridManagerImpl;
import com.ibm.ws.objectgrid.util.ObjectGridUtil;

/**
 * A KeyStoreLoginModule is keystore authentication login module based on
 * JAAS authentication.
 * <p>
 * A login configuration should provide an option "<code>keyStoreFile</code>" to
 * indicate where the keystore file is located. If the <code>keyStoreFile</code>
 * value contains a system property in the form, <code>${system.property}</code>,
 * it will be expanded to the value of the system property.
 * <p>
 * If an option "<code>keyStoreFile</code>" is not provided, the default keystore
 * file name is <code>${java.home}${/}.keystore</code>.
 * <p>
 * Here is a Login module configuration example:
 * <pre><code>
 *   KeyStoreLogin {
 *       com.ibm.websphere.objectgrid.security.plugins.builtins.KeystoreLoginModule required
 *       keyStoreFile="${user.dir}${/}security${/}.keystore";
 *   };
 * </code></pre>
 *
 * @ibm-api
 * @since WAS XD 6.0.1
 *
 * @see LoginModule
 */
public class KeyStoreLoginModule implements LoginModule {

    private static final String CLASS_NAME = KeyStoreLoginModule.class.getName();

    /**
     * Key store file property name
     */
    public static final String KEY_STORE_FILE_PROPERTY_NAME = "keyStoreFile";

    /**
     * Key store type. Only JKS is supported

```

```

    */
    public static final String KEYSTORE_TYPE = "JKS";

    /**
     * The default key store file name
     */
    public static final String DEFAULT_KEY_STORE_FILE = "${java.home}${/}.keystore";

    private CallbackHandler handler;

    private Subject subject;

    private boolean debug = false;

    private Set principals = new HashSet();

    private Set publicCreds = new HashSet();

    private Set privateCreds = new HashSet();

    protected KeyStore keyStore;

    /**
     * Creates a new KeyStoreLoginModule.
     */
    public KeyStoreLoginModule() {
    }

    /**
     * Initializes the login module.
     *
     * @see LoginModule#initialize(Subject, CallbackHandler, Map, Map)
     */
    public void initialize(Subject sub, CallbackHandler callbackHandler,
        Map mapSharedState, Map mapOptions) {

        // initialize any configured options
        debug = "true".equalsIgnoreCase((String) mapOptions.get("debug"));

        if (sub == null)
            throw new IllegalArgumentException("Subject is not specified");

        if (callbackHandler == null)
            throw new IllegalArgumentException(
                "CallbackHandler is not specified");

        // Get the key store path
        String sKeyStorePath = (String) mapOptions
            .get(KEY_STORE_FILE_PROPERTY_NAME);

        // If there is no key store path, the default one is the .keystore
        // file in the java home directory
        if (sKeyStorePath == null) {
            sKeyStorePath = DEFAULT_KEY_STORE_FILE;
        }

        // Replace the system environment variable
        sKeyStorePath = ObjectGridUtil.replaceVar(sKeyStorePath);

        File fileKeyStore = new File(sKeyStorePath);

        try {
            KeyStore store = KeyStore.getInstance("JKS");
            store.load(new FileInputStream(fileKeyStore), null);

            // Save the key store
            keyStore = store;

            if (debug) {
                System.out.println("[KeyStoreLoginModule] initialize: Successfully loaded key store");
            }
        }
        catch (Exception e) {
            ObjectGridRuntimeException re = new ObjectGridRuntimeException(
                "Failed to load keystore: " + fileKeyStore.getAbsolutePath());
            re.initCause(e);
            if (debug) {
                System.out.println("[KeyStoreLoginModule] initialize: Key store loading failed with exception "
                    + e.getMessage());
            }
        }

        this.subject = sub;
        this.handler = callbackHandler;
    }

    /**
     * Authenticates a user based on the keystore file.
     *
     * @see LoginModule#login()
     */

```



```

public boolean login() throws LoginException {
    if (debug) {
        System.out.println("[KeyStoreLoginModule] login: entry");
    }

    String name = null;
    char pwd[] = null;

    if (keyStore == null || subject == null || handler == null) {
        throw new LoginException("Module initialization failed");
    }

    NameCallback nameCallback = new NameCallback("Username:");
    PasswordCallback pwdCallback = new PasswordCallback("Password:", false);

    try {
        handler.handle(new Callback[] { nameCallback, pwdCallback });
    }
    catch (Exception e) {
        throw new LoginException("Callback failed: " + e);
    }

    name = nameCallback.getName();
    char[] tempPwd = pwdCallback.getPassword();

    if (tempPwd == null) {
        // treat a NULL password as an empty password
        tempPwd = new char[0];
    }
    pwd = new char[tempPwd.length];
    System.arraycopy(tempPwd, 0, pwd, 0, tempPwd.length);

    pwdCallback.clearPassword();

    if (debug) {
        System.out.println("[KeyStoreLoginModule] login: "
            + "user entered user name: " + name);
    }

    // Validate the user name and password
    try {
        validate(name, pwd);
    }
    catch (SecurityException se) {
        principals.clear();
        publicCreds.clear();
        privateCreds.clear();
        LoginException le = new LoginException(
            "Exception encountered during login");
        le.initCause(se);

        throw le;
    }

    if (debug) {
        System.out.println("[KeyStoreLoginModule] login: exit");
    }
    return true;
}

/**
 * Indicates the user is accepted.
 * <p>
 * This method is called only if the user is authenticated by all modules in
 * the login configuration file. The principal objects will be added to the
 * stored subject.
 *
 * @return false if for some reason the principals cannot be added; true
 *         otherwise
 *
 * @exception LoginException
 *         LoginException is thrown if the subject is readonly or if
 *         any unrecoverable exceptions is encountered.
 *
 * @see LoginModule#commit()
 */
public boolean commit() throws LoginException {
    if (debug) {
        System.out.println("[KeyStoreLoginModule] commit: entry");
    }

    if (principals.isEmpty()) {
        throw new IllegalStateException("Commit is called out of sequence");
    }

    if (subject.isReadOnly()) {
        throw new LoginException("Subject is Readonly");
    }

    subject.getPrincipals().addAll(principals);
}

```

```

subject.getPublicCredentials().addAll(publicCreds);
subject.getPrivateCredentials().addAll(privateCreds);

principals.clear();
publicCreds.clear();
privateCreds.clear();

if (debug) {
    System.out.println("[KeyStoreLoginModule] commit: exit");
}
return true;
}

/**
 * Indicates the user is not accepted
 *
 * @see LoginModule#abort()
 */
public boolean abort() throws LoginException {
    boolean b = logout();
    return b;
}

/**
 * Logs the user out. Clear all the maps.
 *
 * @see LoginModule#logout()
 */
public boolean logout() throws LoginException {

    // Clear the instance variables
    principals.clear();
    publicCreds.clear();
    privateCreds.clear();

    // clear maps in the subject
    if (!subject.isReadOnly()) {
        if (subject.getPrincipals() != null) {
            subject.getPrincipals().clear();
        }

        if (subject.getPublicCredentials() != null) {
            subject.getPublicCredentials().clear();
        }

        if (subject.getPrivateCredentials() != null) {
            subject.getPrivateCredentials().clear();
        }
    }
    return true;
}

/**
 * Validates the user name and password based on the keystore.
 *
 * @param userName user name
 * @param password password
 * @throws SecurityException if any exceptions encountered
 */
private void validate(String userName, char password[])
    throws SecurityException {

    PrivateKey privateKey = null;

    // Get the private key from the keystore
    try {
        privateKey = (PrivateKey) keyStore.getKey(userName, password);
    }
    catch (NoSuchAlgorithmException nsae) {
        SecurityException se = new SecurityException();
        se.initCause(nsae);
        throw se;
    }
    catch (KeyStoreException kse) {
        SecurityException se = new SecurityException();
        se.initCause(kse);
        throw se;
    }
    catch (UnrecoverableKeyException uke) {
        SecurityException se = new SecurityException();
        se.initCause(uke);
        throw se;
    }

    if (privateKey == null) {
        throw new SecurityException("Invalid name: " + userName);
    }

    // Check the certificats
    Certificate certs[] = null;

```



```

        lc.login();

        Subject subject = lc.getSubject();

        return subject;
    }
    catch (LoginException le) {
        throw new InvalidCredentialException(le);
    }
    catch (IllegalArgumentException ile) {
        throw new InvalidCredentialException(ile);
    }
}

```

Also, eXtreme Scale ships a login module `com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPLoginModule` for this purpose. You must provide the following two options in the JAAS login configuration file.

- `providerURL`: The LDAP server provider URL
- `factoryClass`: The LDAP context factory implementation class

The `LDAPLoginModule` module calls the `com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPAuthenticationHelper.authenticate` method. The following code snippet shows how you can implement the `authenticate` method of the `LDAPAuthenticationHelper`.

```

/**
 * Authenticate the user to the LDAP directory.
 * @param user the user ID, e.g., uid=xxxxxx,c=us,ou=bluepages,o=ibm.com
 * @param pwd the password
 *
 * @throws NamingException
 */
public String[] authenticate(String user, String pwd)
throws NamingException {
    Hashtable env = new Hashtable();
    env.put(Context.INITIAL_CONTEXT_FACTORY, factoryClass);
    env.put(Context.PROVIDER_URL, providerURL);
    env.put(Context.SECURITY_PRINCIPAL, user);
    env.put(Context.SECURITY_CREDENTIALS, pwd);
    env.put(Context.SECURITY_AUTHENTICATION, "simple");

    InitialContext initialContext = new InitialContext(env);

    // Look up for the user
    DirContext dirCtx = (DirContext) initialContext.lookup(user);

    String uid = null;
    int iComma = user.indexOf(",");
    int iEqual = user.indexOf("=");
    if (iComma > 0 && iComma > 0) {
        uid = user.substring(iEqual + 1, iComma);
    }
    else {
        uid = user;
    }

    Attributes attributes = dirCtx.getAttributes("");

    // Check the UID
    String thisUID = (String) (attributes.get(UID).get());

    String thisDept = (String) (attributes.get(HR_DEPT).get());

    if (thisUID.equals(uid)) {
        return new String[] { thisUID, thisDept };
    }
}

```

```

    }
    else {
        return null;
    }
}

```

If authentication succeeds, the ID and password are considered valid. Then the login module gets the ID information and department information from this authenticate method. The login module creates two principals: SimpleUserPrincipal and SimpleDeptPrincipal. You can use the authenticated subject for group authorization (in this case, the department is a group) and individual authorization.

The following example shows a login module configuration that is used to log in to the LDAP server:

```

LDAPLogin { com.ibm.websphere.objectgrid.security.plugins.builtins.LDAPLoginModule required
    providerURL="ldap://directory.acme.com:389/"
    factoryClass="com.sun.jndi.ldap.LdapCtxFactory";
};

```

In the previous configuration, the LDAP server points to the ldap://directory.acme.com:389/server. Change this setting to your LDAP server. This login module uses the provided ID and password to connect to the LDAP server. This implementation is for testing purposes only.

Using the WebSphere Application Server authenticator plug-in

Also, eXtreme Scale provides the com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenAuthenticator built-in implementation to use the WebSphere Application Server security infrastructure. This built-in implementation can be used when the following conditions are true.

1. WebSphere Application Server global security is turned on.
2. All eXtreme Scale clients and servers are launched in WebSphere Application Server JVMs.
3. These application servers are in the same security domain.
4. The eXtreme Scale client is already authenticated in WebSphere Application Server.

The client can use the com.ibm.websphere.objectgrid.security.plugins.builtins.WSTokenCredentialGenerator class to generate a credential. The server uses this Authenticator implementation class to authenticate the credential. If the token is authenticated successfully, a Subject object returns.

This scenario takes advantage of the fact that the client has already been authenticated. Because the application servers that have the servers are in the same security domain as the application servers that house the clients, the security tokens can be propagated from the client to the server so that the same user registry does not need to be authenticated again.

Using the Tivoli® Access Manager authenticator plug-in

Tivoli Access Manager is used widely as a security server. You can also implement Authenticator using the Tivoli Access Manager's provided login modules.

To authenticate a user for Tivoli Access Manager, apply the the `com.tivoli.mts.PDLoginModule` login module, which requires that the calling application provide the following information:

1. A principal name, specified as either a short name or an X.500 name (DN)
2. A password

The login module authenticates the principal and returns the Tivoli Access Manager credential. The login module expects the calling application to provide the following information:

1. The user name, through a `javax.security.auth.callback.NameCallback` object.
2. The password, through a `javax.security.auth.callback.PasswordCallback` object.

When the Tivoli Access Manager credential is successfully retrieved, the JAAS LoginModule creates a Subject and a PDPrincipal. No built-in for Tivoli Access Manager authentication is provided, because it is just with the PDLoginModule module. See the IBM Tivoli Access Manager Authorization Java Classes Developer Reference for more details.

Connecting to WebSphere eXtreme Scale securely

To connect an eXtreme Scale client to a server securely, you can use any connect method in the ObjectGridManager interface which takes a ClientSecurityConfiguration object. The following is a brief example.

```
public ClientClusterContext connect(String catalogServerEndpoints,  
    ClientSecurityConfiguration securityProps,  
    URL overRideObjectGridXml) throws ConnectException;
```

This method takes a parameter of the ClientSecurityConfiguration type, which is an interface representing a client security configuration. You can use `com.ibm.websphere.objectgrid.security.config.ClientSecurityConfigurationFactory` public API to create an instance with default values, or you can create an instance by passing the WebSphere eXtreme Scale client property file. This file contains the following properties that are related to authentication. The value marked with a plus sign (+) is the default.

- `securityEnabled (true, false+)`: This property indicates if security is enabled. When a client connects to a server, the `securityEnabled` value on the client and server side must be both true or both false. For example, if the connected server security is enabled, the client has to set this property to true to connect to the server.
- `authenticationRetryCount (an integer value, 0+)`: This property determines how many retries are attempted for login when a credential is expired. If the value is 0, no retries are attempted. The authentication retry only applies to the case when the credential is expired. If the credential is not valid, there is no retry. Your application is responsible for retrying the operation.

After you create a `com.ibm.websphere.objectgrid.security.config.ClientSecurityConfiguration` object, set the `CredentialGenerator` object on the client using the following method:

```
/**  
 * Set the {@link CredentialGenerator} object for this client.  
 * @param generator the CredentialGenerator object associated with this client  
 */  
void setCredentialGenerator(CredentialGenerator generator);
```

You can set the CredentialGenerator object in the WebSphere eXtreme Scale client property file too, as follows.

- credentialGeneratorClass: The class implementation name for the CredentialGenerator object. It must have a default constructor.
- credentialGeneratorProps: The properties for the CredentialGenerator class. If the value is not null, it is set to the constructed CredentialGenerator object using the setProperties(String) method.

Here is a sample to instantiate a ClientSecurityConfiguration and then use it to connect to the server.

```
/**
 * Get a secure ClientClusterContext
 * @return a secure ClientClusterContext object
 */
protected ClientClusterContext connect() throws ConnectException {
    ClientSecurityConfiguration csConfig = ClientSecurityConfigurationFactory
        .getClientSecurityConfiguration("/properties/security.ogclient.props");

    UserPasswordCredentialGenerator gen= new
        UserPasswordCredentialGenerator("manager", "manager1");

    csConfig.setCredentialGenerator(gen);

    return objectGridManager.connect(csConfig, null);
}
```

When the connect is called, the WebSphere eXtreme Scale client calls the CredentialGenerator.getCredential method to get the client credential. This credential is sent along with the connect request to the server for authentication.

Using a different CredentialGenerator instance per session

In some cases, a WebSphere eXtreme Scale client represents just one client identity, but in others, it might represent multiple identities. Here is one scenario for the latter case: An WebSphere eXtreme Scale client is created and shared in a Web server. All servlets in this Web server use this one WebSphere eXtreme Scale client. Because every servlet represents a different Web client, use different credentials when sending requests to WebSphere eXtreme Scale servers.

WebSphere eXtreme Scale provides for changing the credential on the session level. Every session can uses a different CredentialGenerator object. Therefore, the previous scenarios can be implemented by letting the servlet get a session with a different CredentialGenerator object. The following example illustrates the ObjectGrid.getSession(CredentialGenerator) method in the ObjectGridManager interface.

```
/**
 * Get a session using a <code>CredentialGenerator</code>.
 * <p>
 * This method can only be called by the ObjectGrid client in an ObjectGrid
 * client server environment. If ObjectGrid is used in a local model, that is,
 * within the same JVM with no client or server existing, <code>getSession(Subject)</code>
 * or the <code>SubjectSource</code> plugin should be used to secure the ObjectGrid.
 * <p>If the <code>initialize()</code> method has not been invoked prior to
 * the first <code>getSession</code> invocation, an implicit initialization
 * will occur. This ensures that all of the configuration is complete
 * before any runtime usage is required.</p>
 * @param credGen A <code>CredentialGenerator</code> for generating a credential
 * for the session returned.
 * @return An instance of <code>Session</code>
 * @throws ObjectGridException if an error occurs during processing
 * @throws TransactionCallbackException if the <code>TransactionCallback</code>
```

```

*         throws an exception
* @throws IllegalStateException if this method is called after the
*         <code>destroy()</code> method is called.
*
* @see #destroy()
* @see #initialize()
* @see CredentialGenerator
* @see Session
* @since WAS XD 6.0.1
*/
Session getSession(CredentialGenerator credGen) throws
ObjectGridException, TransactionCallbackException;

```

The following is an example:

```

ObjectGridManager ogManager = ObjectGridManagerFactory.getObjectGridManager();

CredentialGenerator credGenManager = new UserPasswordCredentialGenerator("manager", "xxxxxx");
CredentialGenerator credGenEmployee = new UserPasswordCredentialGenerator("employee", "xxxxxx");

ObjectGrid og = ogManager.getObjectGrid(ctx, "accounting");

// Get a session with CredentialGenerator;
Session session = og.getSession(credGenManager );

// Get the employee map
ObjectMap om = session.getMap("employee");

// start a transaction.
session.begin();

Object rec1 = map.get("xxxxxx");

session.commit();

// Get another session with a different CredentialGenerator;
session = og.getSession(credGenEmployee );

// Get the employee map
om = session.getMap("employee");

// start a transaction.
session.begin();

Object rec2 = map.get("xxxxxx");

session.commit();

```

If you use the `ObjectGrid.getSession` method to get a `Session` object, the session uses the `CredentialGenerator` object set on the `ClientConfigurationSecurity` object. The `ObjectGrid.getSession(CredentialGenerator)` method overrides the `CredentialGenerator` set in the `ClientSecurityConfiguration` object.

If you can reuse the `Session` object, a performance gain results. However, calling the `ObjectGrid.getSession(CredentialGenerator)` method is not very expensive. The major overhead is the increased object garbage collection time. Make sure that you release the references after you are done with the `Session` objects. Generally, if your `Session` object can share the identity, try to reuse the `Session` object. If not, use the `ObjectGrid.getSession(CredentialGenerator)` method.

Client authorization programming

WebSphere eXtreme Scale supports Java Authentication and Authorization Service (JAAS) authorization that is ready to use and also supports custom authorization using the `ObjectGridAuthorization` interface.

The `ObjectGridAuthorization` plug-in is used to authorize `ObjectGrid`, `ObjectMap`, and `JavaMap` accesses to the Principals represented by a `Subject` object in a custom way. A typical implementation of this plug-in is to retrieve the Principals from the `Subject` object, and then check whether the specified permissions are granted to the Principals.

A permission passed to the `checkPermission(Subject, Permission)` method can be one of the following permissions:

- MapPermission
- ObjectGridPermission
- ServerMapPermission
- AgentPermission

Refer to ObjectGridAuthorization API documentation for more details.

MapPermission

The `com.ibm.websphere.objectgrid.security.MapPermission` public class represents permissions to the ObjectGrid resources, specifically the methods of `ObjectMap` or `JavaMap` interfaces. WebSphere eXtreme Scale defines the following permission strings to access the methods of `ObjectMap` and `JavaMap`:

- **read:** Permission to read the data from the map. The integer constant is defined as `MapPermission.READ`.
- **write:** Permission to update the data in the map. The integer constant is defined as `MapPermission.WRITE`.
- **insert:** Permission to insert the data into the map. The integer constant is defined as `MapPermission.INSERT`.
- **remove:** Permission to remove the data from the map. The integer constant is defined as `MapPermission.REMOVE`.
- **invalidate:** Permission to invalidate the data from the map. The integer constant is defined as `MapPermission.INVALIDATE`.
- **all:** All above permissions: read, write, insert, remote, and invalidate. The integer constant is defined as `MapPermission.ALL`.

Refer to MapPermission API documentation for more details.

You can construct a `MapPermission` object by passing the fully qualified ObjectGrid map name (in format `[ObjectGrid_name].[ObjectMap_name]`) and the permission string or integer value. A permission string can be a comma-delimited string of the previous permission strings such as `read, insert`, or it can be `all`. A permission integer value can be any previously mentioned permission integer constants or a mathematical value of several integer permission constants, such as `MapPermission.READ | MapPermission.WRITE`.

The authorization occurs when an `ObjectMap` or `JavaMap` method is called. The run time checks different permissions for different methods. If the required permissions are not granted to the client, an `AccessControlException` results.

Table 12. List of methods and the required MapPermission

Permission	ObjectMap/JavaMap
read	Boolean containsKey(Object)
	Boolean equals(Object)
	Object get(Object)
	Object get(Object, Serializable)
	List getAll(List)
	List getAll(List keyList, Serializable)
	List getAllForUpdate(List)
	List getAllForUpdate(List, Serializable)
	Object getForUpdate(Object)
	Object getForUpdate(Object, Serializable)
	public Object getNextKey(long)
write	Object put(Object key, Object value)
	void put(Object, Object, Serializable)
	void putAll(Map)
	void putAll(Map, Serializable)
	void update(Object, Object)
	void update(Object, Object, Serializable)
insert	public void insert (Object, Object)
	void insert(Object, Object, Serializable)
remove	Object remove (Object)
	void removeAll(Collection)
	void clear()
invalidate	public void invalidate (Object, Boolean)
	void invalidateAll(Collection, Boolean)
	void invalidateUsingKeyword(Serializable)
	int setTimeToLive(int)

Authorization is based solely on which method is used, rather than what the method really does. For example, a put method can insert or update a record based on whether the record exists. However, the insert or update cases are not distinguished.

An operation type can be achieved by combinations of other types. For example, an update can be achieved by a remove and then an insert. Consider these combinations when designing your authorization policies.

ObjectGridPermission

A `com.ibm.websphere.objectgrid.security.ObjectGridPermission` represents permissions to the ObjectGrid:

- Query: permission to create an object query or entity query. The integer constant is defined as `ObjectGridPermission.QUERY`.

- Dynamic map: permission to create a dynamic map based on the map template. The integer constant is defined as `ObjectGridPermission.DYNAMIC_MAP`.

Refer to `ObjectGridPermission` API documentation for more details.

The following table summarizes the methods and the required `ObjectGridPermission`:

Table 13. List of methods and the required `ObjectGridPermission`

Permission action	Methods
query	<code>com.ibm.websphere.objectgrid.Session.createObjectQuery(String)</code>
query	<code>com.ibm.websphere.objectgrid.em.EntityManager.createQuery(String)</code>
dynamicmap	<code>com.ibm.websphere.objectgrid.Session.getMap(String)</code>

ServerMapPermission

An `ServerMapPermission` represents permissions to an `ObjectMap` hosted in a server. The name of the permission is the full name of the `ObjectGrid` map name. It has the following actions:

- **replicate**: permission to replicate a server map to near cache
- **dynamicIndex**: permission for a client to create or remove a dynamic index on a server

Refer to `ServerMapPermission` API documentation for more details. The detailed methods, which require different `ServerMapPermission`, are listed in the following table:

Table 14. Permissions to a server-hosted `ObjectMap`

Permission action	Methods
replicate	<code>com.ibm.websphere.objectgrid.ClientReplicableMap.enableClientReplication(Mode, int[], ReplicationMapListener)</code>
dynamicIndex	<code>com.ibm.websphere.objectgrid.BackingMap.createDynamicIndex(String, Boolean, String, DynamicIndexCallback)</code>
dynamicIndex	<code>com.ibm.websphere.objectgrid.BackingMap.removeDynamicIndex(String)</code>

AgentPermission

An `AgentPermission` represents permissions to the datagrid agents. The name of the permission is the full name of the `ObjectGrid` map, and the action is a comma-delimited string of agent implementation class names or package names.

Refer to `AgentPermission` API documentation for more information.

The following methods in the class `com.ibm.websphere.objectgrid.datagrid.AgentManager` require `AgentPermission`.

```
com.ibm.websphere.objectgrid.datagrid.AgentManager#callMapAgent(MapGridAgent, Collection)
com.ibm.websphere.objectgrid.datagrid.AgentManager#callMapAgent(MapGridAgent)
com.ibm.websphere.objectgrid.datagrid.AgentManager#callReduceAgent(ReduceGridAgent, Collection)
com.ibm.websphere.objectgrid.datagrid.AgentManager#callReduceAgent(ReduceGridAgent, Collection)
```

Authorization mechanisms

WebSphere eXtreme Scale supports two kinds of authorization mechanisms: Java Authentication and Authorization Service (JAAS) authorization and custom authorization. These mechanisms apply to all authorizations. JAAS authorization augments the Java security policies with user-centric access controls. Permissions

can be granted based not just on what code is running, but also on who is running it. JAAS authorization is part of the SDK Version 5 and later.

Additionally, WebSphere eXtreme Scale also supports custom authorization with the following plug-in:

- ObjectGridAuthorization: custom way to authorize access to all artifacts.

You can implement your own authorization mechanism if you do not want to use JAAS authorization. By using a custom authorization mechanism, you can use the policy database, policy server, or Tivoli Access Manager to manage the authorizations.

You can configure the authorization mechanism in two ways:

- XML configuration

You can use the ObjectGrid XML file to define an ObjectGrid and set the authorization mechanism to either AUTHORIZATION_MECHANISM_JAAS or AUTHORIZATION_MECHANISM_CUSTOM. Here is the secure-objectgrid-definition.xml file that is used in the enterprise application ObjectGridSample:

```
<objectGrids>
  <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
    authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS">
    <bean id="TransactionCallback"
      classname="com.ibm.websphere.samples.objectgrid.HeapTransactionCallback" />
    ...
  </objectGrids>
```

- Programmatic configuration

If you want to create an ObjectGrid using method `ObjectGrid.setAuthorizationMechanism(int)`, you can call the following method to set the authorization mechanism. Calling this method applies only to the local WebSphere eXtreme Scale programming model when you directly instantiate the ObjectGrid instance:

```
/**
 * Set the authorization Mechanism. The default is
 * com.ibm.websphere.objectgrid.security.SecurityConstants.
 * AUTHORIZATION_MECHANISM_JAAS.
 * @param authMechanism the map authorization mechanism
 */
void setAuthorizationMechanism(int authMechanism);
```

JAAS authorization

A `javax.security.auth.Subject` object represents an authenticated user. A Subject consists of a set of principals, and each Principal represents an identity for that user. For example, a Subject can have a name principal, for example, Joe Smith, and a group principal, for example, manager.

Using the JAAS authorization policy, permissions can be granted to specific Principals. WebSphere eXtreme Scale associates the Subject with the current access control context. For each call to the `ObjectMap` or `Javamap` method, the Java runtime automatically determines if the policy grants the required permission only to a specific Principal and if so, the operation is allowed only if the Subject associated with the access control context contains the designated Principal.

You must be familiar with the policy syntax of the policy file. For detailed description of JAAS authorization, refer to the JAAS Reference Guide.

WebSphere eXtreme Scale has a special code base that is used for checking the JAAS authorization to the ObjectMap and JavaMap method calls. This special code base is `http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction`. Use this code base when granting ObjectMap or JavaMap permissions to principals. This special code was created because the Java archive (JAR) file for eXtreme Scale is granted with all permissions.

The template of the policy to grant the MapPermission permission is:

```
grant codeBase "http://www.ibm.com/com/ibm/ws/objectgrid/security/PrivilegedAction"
  <Principal field(s)>{
    permission com.ibm.websphere.objectgrid.security.MapPermission
      "[ObjectGrid_name].[ObjectMap_name]", "action";
    ....
    permission com.ibm.websphere.objectgrid.security.MapPermission
      "[ObjectGrid_name].[ObjectMap_name]", "action";
  };
```

A Principal field looks like the following example:

```
principal Principal_class "principal_name"
```

In this policy, only insert and read permissions are granted to these four maps to a certain principal. The other policy file, `fullAccessAuth.policy`, grants all permissions to these maps to a principal. Before running the application, change the `principal_name` and `principal class` to appropriate values. The value of the `principal_name` depends on the user registry. For example, if local OS is used as user registry, the machine name is `MACH1`, the user ID is `user1`, and the `principal_name` is `MACH1/user1`.

The JAAS authorization policy can be put directly into the Java policy file, or it can be put in a separate JAAS authorization file and then set in either of two ways:

- Use the following JVM argument:
`-Djava.security.auth.policy=file:[JAAS_AUTH_POLICY_FILE]`
- Use the following property in the `java.security` file:
`-Dauth.policy.url.x=file:[JAAS_AUTH_POLICY_FILE]`

Custom ObjectGrid authorization

ObjectGridAuthorization plug-in is used to authorize ObjectGrid, ObjectMap, and JavaMap accesses to the Principals represented by a Subject object in a custom way. A typical implementation of this plug-in is to retrieve the Principals from the Subject object, and then check whether or not the specified permissions are granted to the Principals.

A permission passed to the `checkPermission(Subject, Permission)` method could be one of the following:

- MapPermission
- ObjectGridPermission
- AgentPermission
- ServerMapPermission

Refer to ObjectGridAuthorization API documentation for more details.

The ObjectGridAuthorization plug-in can be configured in the following ways:

- XML configuration

You can use the ObjectGrid XML file to define an ObjectAuthorization plug-in. Here is an example:

```
<objectGrids>
  <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
    authorizationMechanism="AUTHORIZATION_MECHANISM_CUSTOM">
    ...
    <bean id="ObjectGridAuthorization"
      className="com.acme.ObjectGridAuthorizationImpl" />
  </objectGrids>
```

- Programmatic configuration

If you want to create an ObjectGrid using the API method `ObjectGrid.setObjectGridAuthorization(ObjectGridAuthorization)`, you can call the following method to set the authorization plug-in. This method only applies to the local eXtreme Scale programming model when you directly instantiate the ObjectGrid instance.

```
/**
 * Sets the <code>ObjectGridAuthorization</code> for this ObjectGrid instance.
 * <p>
 * Passing <code>null</code> to this method removes a previously set
 * <code>ObjectGridAuthorization</code> object from an earlier invocation of this method
 * and indicates that this <code>ObjectGrid</code> is not associated with a
 * <code>ObjectGridAuthorization</code> object.
 * <p>
 * This method should only be used when ObjectGrid security is enabled. If
 * the ObjectGrid security is disabled, the provided <code>ObjectGridAuthorization</code> object
 * will not be used.
 * <p>
 * A <code>ObjectGridAuthorization</code> plugin can be used to authorize
 * access to the ObjectGrid and maps. Please refer to <code>ObjectGridAuthorization</code> for more details.
 *
 * <p>
 * As of XD 6.1, the <code>setMapAuthorization</code> is deprecated and
 * <code>setObjectGridAuthorization</code> is recommended for use. However,
 * if both <code>MapAuthorization</code> plugin and <code>ObjectGridAuthorization</code> plugin
 * are used, ObjectGrid will use the provided <code>MapAuthorization</code> to authorize map accesses,
 * even though it is deprecated.
 * <p>
 * Note, to avoid an <code>IllegalStateException</code>, this method must be
 * called prior to the <code>initialize()</code> method. Also, keep in mind
 * that the <code>getSession</code> methods implicitly call the
 * <code>initialize()</code> method if it has yet to be called by the
 * application.
 *
 * @param ogAuthorization the <code>ObjectGridAuthorization</code> plugin
 *
 * @throws IllegalStateException if this method is called after the
 * <code>initialize()</code> method is called.
 *
 * @see #initialize()
 * @see ObjectGridAuthorization
 * @since WAS XD 6.1
 */
void setObjectGridAuthorization(ObjectGridAuthorization ogAuthorization);
```

Implementing ObjectGridAuthorization

The Boolean `checkPermission(Subject subject, Permission permission)` method of the `ObjectGridAuthorization` interface is called by the WebSphere eXtreme Scale run time to check whether the passed-in subject object has the passed-in permission. The implementation of the `ObjectGridAuthorization` interface returns true if the object has the permission, and false if not.

A typical implementation of this plug-in is to retrieve the principals from the Subject object and check whether the specified permissions are granted to the principals by consulting specific policies. These policies are defined by users. For example, the policies can be defined in a database, a plain file, or a Tivoli Access Manager policy server.

For example, we can use Tivoli Access Manager policy server to manage the authorization policy and use its API to authorize the access. For how to use Tivoli

Access Manager Authorization APIs, refer to the IBM Tivoli Access Manager Authorization Java Classes Developer Reference for more details.

This sample implementation has the following assumptions:

- Check authorization for MapPermission only. For other permissions, always return true.
- The Subject object contains a com.tivoli.mts.PDPrincipal principal.
- The Tivoli Access Manager policy server has defined the following permissions for the ObjectMap or JavaMap name object. The object that is defined in the policy server must have the same name as the ObjectMap or JavaMap name in the format of [ObjectGrid_name].[ObjectMap_name]. The permission is the first character of the permission strings that are defined in the MapPermission permission. For example, the permission "r" that is defined in the policy server represents the read permission to the ObjectMap map.

The following code snippet demonstrates how to implement the checkPermission method:

```
/**
 * @see com.ibm.websphere.objectgrid.security.plugins.
 * MapAuthorization#checkPermission
 * (javax.security.auth.Subject, com.ibm.websphere.objectgrid.security.
 * MapPermission)
 */
public boolean checkPermission(final Subject subject,
    Permission p) {

    // For non-MapPermission, we always authorize.
    if (!(p instanceof MapPermission)){
        return true;
    }

    MapPermission permission = (MapPermission) p;

    String[] str = permission.getParsedNames();

    StringBuffer pdPermissionStr = new StringBuffer(5);
    for (int i=0; i<str.length; i++) {
        pdPermissionStr.append(str[i].substring(0,1));
    }

    PDPermission pdPerm = new PDPermission(permission.getName(),
    pdPermissionStr.toString());

    Set principals = subject.getPrincipals();

    Iterator iter= principals.iterator();
    while(iter.hasNext()) {
        try {
            PDPrincipal principal = (PDPrincipal) iter.next();
            if (principal.implies(pdPerm)) {
                return true;
            }
        }
        catch (ClassCastException cce) {
            // Handle exception
        }
    }
    return false;
}
```

Data grid authentication

You can use the secure token manager plug-in to enable server-to-server authentication, which requires you to implement the SecureTokenManager interface.

The generateToken(Object) method takes an object protect, and then generates a token that cannot be understood by others. The verifyTokens(byte[]) method does the reverse process: it converts the token back to the original object.

A simple SecureTokenManager implementation uses a simple encoding algorithm, such as a XOR algorithm, to encode the object in serialized form and then use corresponding decoding algorithm to decode the token. This implementation is not secure and is easy to break.

WebSphere eXtreme Scale default implementation

WebSphere eXtreme Scale provides an immediately available implementation for this interface. This default implementation uses a key pair to sign and verify the signature, and uses a secret key to encrypt the content. Every server has a JCKES type keystore to store the key pair, a private key and public key, and a secret key. The keystore has to be the JCKES type to store secret keys. These keys are used to encrypt and sign or verify the secret string on the sending end. Also, the token is associated with an expiration time. On the receiving end, the data is verified, decrypted, and compared to the receiver secret string. Secure Sockets Layer (SSL) communication protocols are not required between a pair of servers for authentication because the private keys and public keys serve the same purpose. However, if server communication is not encrypted, the data can be stolen by looking at the communication. Because the token expires soon, the replay attack threat is minimized. This possibility is significantly decreased if all servers are deployed behind a firewall.

The disadvantage of this approach is that the WebSphere eXtreme Scale administrators have to generate keys and transport them to all servers, which can cause security breach during transportation.

Local security programming

WebSphere eXtreme Scale provides several security endpoints to allow you to integrate custom mechanisms. In the local programming model, the main security function is authorization, and has no authentication support . You must authenticate outside of WebSphere Application Server. However, there are provided plug-ins to obtain and validate Subject objects.

Authentication

In the local programming model, eXtreme Scale does not provide any authentication mechanism, but relies on the environment, either application servers or applications, for authentication. When eXtreme Scale is used in WebSphere Application Server or WebSphere Extended Deployment, applications can use the WebSphere Application Server security authentication mechanism. When eXtreme Scale is running in a Java 2 Platform, Standard Edition (J2SE) environment, the application has to manage authentications with Java Authentication and Authorization Service (JAAS) authentication or other authentication mechanisms. For more information about using JAAS authentication, see the JAAS reference guide. The contract between an application and an ObjectGrid instance is the javax.security.auth.Subject object. After the client is authenticated by the application

server or the application, the application can retrieve the authenticated `javax.security.auth.Subject` object and use this `Subject` object to get a session from the `ObjectGrid` instance by calling the `ObjectGrid.getSession(Subject)` method. This `Subject` object is used to authorize accesses to the map data. This contract is called a subject passing mechanism. The following example illustrates the `ObjectGrid.getSession(Subject)` API.

```
/**
 * This API allows the cache to use a specific subject rather than the one
 * configured on the ObjectGrid to get a session.
 * @param subject
 * @return An instance of Session
 * @throws ObjectGridException
 * @throws TransactionCallbackException
 * @throws InvalidSubjectException the subject passed in is not valid based
 * on the SubjectValidation mechanism.
 */
public Session getSession(Subject subject)
throws ObjectGridException, TransactionCallbackException, InvalidSubjectException;
```

The `ObjectGrid.getSession()` method in the `ObjectGrid` interface can also be used to get a `Session` object:

```
/**
 * This method returns a Session object that can be used by a single thread at a time.
 * You cannot share this Session object between threads without placing a
 * critical section around it. While the core framework allows the object to move
 * between threads, the TransactionCallback and Loader might prevent this usage,
 * especially in J2EE environments. When security is enabled, this method uses the
 * SubjectSource to get a Subject object.
 *
 * If the initialize method has not been invoked prior to the first
 * getSession invocation, then an implicit initialization occurs. This
 * initialization ensures that all of the configuration is complete before
 * any runtime usage is required.
 *
 * @see #initialize()
 * @return An instance of Session
 * @throws ObjectGridException
 * @throws TransactionCallbackException
 * @throws IllegalStateException if this method is called after the
 * destroy() method is called.
 */
public Session getSession()
throws ObjectGridException, TransactionCallbackException;
```

As the API documentation specifies, when security is enabled, this method uses the `SubjectSource` plug-in to get a `Subject` object. The `SubjectSource` plug-in is one of the security plug-ins defined in eXtreme Scale to support propagating `Subject` objects. See [Security-related plug-ins](#) for more information. The `getSession(Subject)` method can be called on the local `ObjectGrid` instance only. If you call the `getSession(Subject)` method on a client side in a distributed eXtreme Scale configuration, an `IllegalStateException` results.

Security plug-ins

WebSphere eXtreme Scale provides two security plug-ins that are related to the subject passing mechanism: the `SubjectSource` and `SubjectValidation` plug-ins.

SubjectSource plug-in

The `SubjectSource` plug-in, represented by the `com.ibm.websphere.objectgrid.security.plugins.SubjectSource` interface, is a plug-in that is used to get a `Subject` object from an eXtreme Scale running environment. This environment can be an application using the `ObjectGrid` or an application server that hosts the application. Consider the `SubjectSource` plug-in an alternative

to the subject passing mechanism. Using the subject passing mechanism, the application retrieves the Subject object and uses it to get the ObjectGrid session object. With the SubjectSource plug-in, the eXtreme Scale runtime retrieves the Subject object and uses it to get the session object. The subject passing mechanism gives the control of Subject objects to applications, while the SubjectSource plug-in mechanism frees applications from retrieving the Subject object. You can use the SubjectSource plug-in to get a Subject object that represents an eXtreme Scale client that is used for authorization. When the ObjectGrid.getSession method is called, the Subject getSubject throws an ObjectGridSecurityException if security is enabled. WebSphere eXtreme Scale provides a default implementation of this plug-in: com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectSourceImpl. This implementation can be used to retrieve a caller subject or a RunAs subject from the thread when an application is running in WebSphere Application Server. You can configure this class in your ObjectGrid descriptor XML file as the SubjectSource implementation class when using eXtreme Scale in WebSphere Application Server. The following code snippet shows the main flow of the WSSubjectSourceImpl.getSubject method.

```
Subject s = null;
try {
    if (finalType == RUN_AS_SUBJECT) {
        // get the RunAs subject
        s = com.ibm.websphere.security.auth.WSSubject.getRunAsSubject();
    }
    else if (finalType == CALLER_SUBJECT) {
        // get the callersubject
        s = com.ibm.websphere.security.auth.WSSubject.getCallerSubject();
    }
}
catch (WSSecurityException wse) {
    throw new ObjectGridSecurityException(wse);
}

return s;
```

For other details, refer to the API documentation for the SubjectSource plug-in and the WSSubjectSourceImpl implementation.

SubjectValidation plug-in

The SubjectValidation plug-in, which is represented by the com.ibm.websphere.objectgrid.security.plugins.SubjectValidation interface, is another security plug-in. The SubjectValidation plug-in can be used to validate that a javax.security.auth.Subject, either passed to the ObjectGrid or retrieved by the SubjectSource plug-in, is a valid Subject that has not been tampered with.

The SubjectValidation.validateSubject(Subject) method in the SubjectValidation interface takes a Subject object and returns a Subject object. Whether a Subject object is considered valid and which Subject object is returned are all up to your implementations. If the Subject object is not valid, an InvalidSubjectException results.

You can use this plug-in if you do not trust the Subject object that is passed to this method. This case is rare considering that you trust the application developers who develop the code to retrieve the Subject object.

An implementation of this plug-in needs support from the Subject object creator because only the creator knows if the Subject object has been tampered with.

However, some subject creator might not know if the Subject has been tampered with. In this case, this plug-in is not useful.

WebSphere eXtreme Scale provides a default implementation of SubjectValidation: `com.ibm.websphere.objectgrid.security.plugins.builtins.WSSubjectValidationImpl`. You can use this implementation to validate the WebSphere Application Server-authenticated subject. You can configure this class as the SubjectValidation implementation class when using eXtreme Scale in WebSphere Application Server. The `WSSubjectValidationImpl` implementation considers a Subject object valid only if the credential token that is associated with this Subject has not been tampered with. You can change other parts of the Subject object. The `WSSubjectValidationImpl` implementation asks WebSphere Application Server for the original Subject corresponding to the credential token and returns the original Subject object as the validated Subject object. Therefore, the changes made to the Subject contents other than the credential token have no effects. The following code snippet shows the basic flow of the `WSSubjectValidationImpl.validateSubject(Subject)`.

```
// Create a LoginContext with scheme WSLogin and
// pass a Callback handler.
LoginContext lc = new LoginContext("WSLogin",
new WSCredTokenCallbackHandlerImpl(subject));

// When this method is called, the callback handler methods
// will be called to log the user in.
lc.login();

// Get the subject from the LoginContext
return lc.getSubject();
```

In the previous code snippet, a credential token callback handler object, `WSCredTokenCallbackHandlerImpl`, is created with the Subject object to validate. Then a `LoginContext` object is created with the login scheme `WSLogin`. When the `lc.login` method is called, WebSphere Application Server security retrieves the credential token from the Subject object and then returns the correspondent Subject as the validated Subject object.

For other details, refer to the Java APIs of SubjectValidation and `WSSubjectValidationImpl` implementation.

Plug-in configuration

You can configure the SubjectValidation plug-in and SubjectSource plug-in in two ways:

- **XML Configuration** You can use the ObjectGrid XML file to define an ObjectGrid and set these two plug-ins. Here is an example, in which the `WSSubjectSourceImpl` class is configured as the SubjectSource plug-in and the `WSSubjectValidation` class is configured as the SubjectValidation plug-in.

```
<objectGrids>
  <objectGrid name="secureClusterObjectGrid" securityEnabled="true"
    authorizationMechanism="AUTHORIZATION_MECHANISM_JAAS">
    <bean id="SubjectSource"
      className="com.ibm.websphere.objectgrid.security.plugins.builtins.
        WSSubjectSourceImpl" />
    <bean id="SubjectValidation"
      className="com.ibm.websphere.objectgrid.security.plugins.builtins.
        WSSubjectValidationImpl" />
    <bean id="TransactionCallback"
```

```

className="com.ibm.websphere.samples.objectgrid.
  HeapTransactionCallback" />
...
</objectGrids>

```

- **Programming** If you want to create an ObjectGrid through APIs, you can call the following methods to set the SubjectSource or SubjectValidation plug-ins.

```

**
* Set the SubjectValidation plug-in for this ObjectGrid instance. A
* SubjectValidation plug-in can be used to validate the Subject object
* passed in as a valid Subject. Refer to {@link SubjectValidation}
* for more details.
* @param subjectValidation the SubjectValidation plug-in
*/
void setSubjectValidation(SubjectValidation subjectValidation);

/**
* Set the SubjectSource plug-in. A SubjectSource plug-in can be used
* to get a Subject object from the environment to represent the
* ObjectGrid client.
*
* @param source the SubjectSource plug-in
*/
void setSubjectSource(SubjectSource source);

```

Write your own JAAS authentication code

You can write your own Java Authentication and Authorization Service (JAAS) authentication code to handle the authentication. You need to write your own login modules and then configure the login modules for your authentication module.

The login module receives information about a user and authenticates the user. This information can be anything that can identify the user. For example, the information can be a user ID and password, client certificate, and so on. After receiving the information, the login module verifies that the information represents a valid subject and then creates a Subject object. Currently, several implementations of login modules are available to the public.

After a login module is written, configure this login module for the run time to use. You must configure a JAAS login module. This login module contains the login module and its authentication scheme. For example:

```

FileLogin
{
    com.acme.auth.FileLoginModule required
};

```

The authentication scheme is FileLogin and the login module is com.acme.auth.FileLoginModule. The required token indicates that the FileLoginModule module must validate this login or the entire scheme fails.

Setting the JAAS login module configuration file can be done in one of the following ways:

- Set the JAAS login module configuration file in the login.config.url property in the java.security file, for example:


```
login.config.url.1=file:${java.home}/lib/security/file.login
```
- Set the JAAS login module configuration file from the command line by using the **-Djava.security.auth.login.config** Java virtual machine (JVM) arguments, for example, `-Djava.security.auth.login.config ==$JAVA_HOME/lib/security/file.login`

If your code is running in WebSphere Application Server, you must configure the JAAS login in the administrative console and store this login configuration in the application server configuration. See Login configuration for Java Authentication and Authorization Service for details.

Chapter 8. Troubleshooting



In addition to the logs and trace, messages, and release notes discussed in this section, you can use monitoring tools to figure out issues such as the location of data in the environment, the availability of servers in the data grid, and so on. If you are running in a WebSphere Application Server environment, you can use Performance Monitoring Infrastructure (PMI). If you are running in a stand-alone environment, you can use a vendor monitoring tool, such as CA Wily Introscope or Hyperic HQ. You can also use and customize the `xscmd` utility to display textual information about your environment.

Troubleshooting and support for WebSphere eXtreme Scale

To isolate and resolve problems with your IBM products, you can use the troubleshooting and support information. This information contains instructions for using the problem-determination resources that are provided with your IBM products, including WebSphere eXtreme Scale .

Techniques for troubleshooting problems

Troubleshooting is a systematic approach to solving a problem. The goal of troubleshooting is to determine why something does not work as expected and how to resolve the problem. Certain common techniques can help with the task of troubleshooting.

The first step in the troubleshooting process is to describe the problem completely. Problem descriptions help you and the IBM technical-support representative know where to start to find the cause of the problem. This step includes asking yourself basic questions:

- What are the symptoms of the problem?
- Where does the problem occur?
- When does the problem occur?
- Under which conditions does the problem occur?
- Can the problem be reproduced?

The answers to these questions typically lead to a good description of the problem, which can then lead you to a problem resolution.

What are the symptoms of the problem?

When starting to describe a problem, the most obvious question is “What is the problem?” This question might seem straightforward; however, you can break it down into several more-focused questions that create a more descriptive picture of the problem. These questions can include:

- Who, or what, is reporting the problem?
- What are the error codes and messages?
- How does the system fail? For example, is it a loop, hang, crash, performance degradation, or incorrect result?

Where does the problem occur?

Determining where the problem originates is not always easy, but it is one of the most important steps in resolving a problem. Many layers of technology can exist between the reporting and failing components. Networks, the data grid, and servers are only a few of the components to consider when you are investigating problems.

The following questions help you to focus on where the problem occurs to isolate the problem layer:

- Is the problem specific to one platform or operating system, or is it common across multiple platforms or operating systems?
- Is the current environment and configuration supported?
- Do all users have the problem?
- (For multi-site installations.) Do all sites have the problem?

If one layer reports the problem, the problem does not necessarily originate in that layer. Part of identifying where a problem originates is understanding the environment in which it exists. Take some time to completely describe the problem environment, including the operating system and version, all corresponding software and versions, and hardware information. Confirm that you are running within an environment that is a supported configuration; many problems can be traced back to incompatible levels of software that are not intended to run together or have not been fully tested together.

When does the problem occur?

Develop a detailed timeline of events leading up to a failure, especially for those cases that are one-time occurrences. You can most easily develop a timeline by working backward: Start at the time an error was reported (as precisely as possible, even down to the millisecond), and work backward through the available logs and information. Typically, you need to look only as far as the first suspicious event that you find in a diagnostic log.

To develop a detailed timeline of events, answer these questions:

- Does the problem happen only at a certain time of day or night?
- How often does the problem happen?
- What sequence of events leads up to the time that the problem is reported?
- Does the problem happen after an environment change, such as upgrading or installing software or hardware?

Responding to these types of questions can give you a frame of reference in which to investigate the problem.

Under which conditions does the problem occur?

Knowing which systems and applications are running at the time that a problem occurs is an important part of troubleshooting. These questions about your environment can help you to identify the root cause of the problem:

- Does the problem always occur when the same task is being performed?
- Does a certain sequence of events need to happen for the problem to occur?
- Do any other applications fail at the same time?

Answering these types of questions can help you explain the environment in which the problem occurs and correlate any dependencies. Remember that just because multiple problems might have occurred around the same time, the problems are not necessarily related.

Can the problem be reproduced?

From a troubleshooting standpoint, the ideal problem is one that can be reproduced. Typically, when a problem can be reproduced you have a larger set of tools or procedures at your disposal to help you investigate. Consequently, problems that you can reproduce are often easier to debug and solve.

However, problems that you can reproduce can have a disadvantage: If the problem is of significant business impact, you do not want it to recur. If possible, recreate the problem in a test or development environment, which typically offers you more flexibility and control during your investigation.

- Can the problem be recreated on a test system?
- Are multiple users or applications encountering the same type of problem?
- Can the problem be recreated by running a single command, a set of commands, or a particular application?

Searching knowledge bases

You can often find solutions to problems by searching IBM knowledge bases. You can optimize your results by using available resources, support tools, and search methods.

About this task

You can find useful information by searching the information center for WebSphere eXtreme Scale . However, sometimes you need to look beyond the information center to answer your questions or resolve problems.

Procedure

To search knowledge bases for information that you need, use one or more of the following approaches:

- Search for content by using the IBM Support Assistant (ISA).
ISA is a no-charge software serviceability workbench that helps you answer questions and resolve problems with IBM software products. You can find instructions for downloading and installing ISA on the ISA website.
- Find the content that you need by using the IBM Support Portal.
The IBM Support Portal is a unified, centralized view of all technical support tools and information for all IBM systems, software, and services. The IBM Support Portal lets you access the IBM electronic support portfolio from one place. You can tailor the pages to focus on the information and resources that you need for problem prevention and faster problem resolution. Familiarize yourself with the IBM Support Portal by viewing the demo videos (https://www.ibm.com/blogs/SPNA/entry/the_ibm_support_portal_videos) about this tool. These videos introduce you to the IBM Support Portal, explore troubleshooting and other resources, and demonstrate how you can tailor the page by moving, adding, and deleting portlets.
- Search for content about WebSphere eXtreme Scale by using one of the following additional technical resources:

- WebSphere eXtreme Scale release notes
- WebSphere eXtreme Scale Support website
- WebSphere eXtreme Scale forum
- Search for content by using the IBM masthead search. You can use the IBM masthead search by typing your search string into the Search field at the top of any ibm.com® page.
- Search for content by using any external search engine, such as Google, Yahoo, or Bing. If you use an external search engine, your results are more likely to include information that is outside the ibm.com domain. However, sometimes you can find useful problem-solving information about IBM products in newsgroups, forums, and blogs that are not on ibm.com.

Tip: Include “IBM” and the name of the product in your search if you are looking for information about an IBM product.

Getting fixes

A product fix might be available to resolve your problem.

Procedure

To find and install fixes:

1. Obtain the tools required to get the fix. Use the IBM Update Installer to install and apply various types of maintenance packages for WebSphere eXtreme Scale or WebSphere eXtreme Scale Client. Because the Update Installer undergoes regular maintenance, you must use the most current version of the tool.
2. Determine which fix you need. See the Recommended fixes for WebSphere eXtreme Scale to select the latest fix. When you select a fix, the download document for that fix opens.
3. Download the fix. In the download document, click the link for the latest fix in the “Download package” section.
4. Apply the fix. Follow the instructions in the “Installation Instructions” section of the download document.
5. Subscribe to receive weekly e-mail notifications about fixes and other IBM Support information.

Getting fixes from Fix Central

You can use Fix Central to find the fixes that are recommended by IBM Support for a variety of products, including WebSphere eXtreme Scale . With Fix Central, you can search, select, order, and download fixes for your system with a choice of delivery options. A WebSphere eXtreme Scale product fix might be available to resolve your problem.

Procedure

To find and install fixes:

1. Obtain the tools that are required to get the fix. If it is not installed, obtain your product update installer. You can download the installer from Fix Central. This site provides download, installation, and configuration instructions for the update installer.
2. Select as the product, and select one or more check boxes that are relevant to the problem that you want to resolve.
3. Identify and select the fix that is required.

4. Download the fix.
 - a. Open the download document and follow the link in the “Download Package” section.
 - b. When downloading the file, ensure that the name of the maintenance file is not changed. This change might be intentional, or it might be an inadvertent change that is caused by certain web browsers or download utilities.
5. Apply the fix.
 - a. Follow the instructions in the “Installation Instructions” section of the download document.
 - b. For more information, see the “Installing fixes with the Update Installer” topic in the product documentation.
6. Optional: Subscribe to receive weekly e-mail notifications about fixes and other IBM Support updates.

Contacting IBM Support

IBM Support provides assistance with product defects, answers FAQs, and helps users resolve problems with the product.

Before you begin

After trying to find your answer or solution by using other self-help options, such as release notes, you can contact IBM Support. Before contacting IBM Support, your company or organization must have an active IBM maintenance contract, and you must be authorized to submit problems to IBM. For information about the types of available support, see the Support portfolio topic in the *“Software Support Handbook”*.

Procedure

To contact IBM Support about a problem:

1. Define the problem, gather background information, and determine the severity of the problem. For more information, see the Getting IBM support topic in the *Software Support Handbook*.
2. Gather diagnostic information.
3. Submit the problem to IBM Support in one of the following ways:
 - Using IBM Support Assistant (ISA):
 - Online through the IBM Support Portal: You can open, update, and view all of your service requests from the Service Request portlet on the Service Request page.
 - By phone: For the phone number to call in your region, see the Directory of worldwide contacts web page.

Results

If the problem that you submit is for a software defect or for missing or inaccurate documentation, IBM Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the IBM Support website daily, so that other users who experience the same problem can benefit from the same resolution.

Exchanging information with IBM

To diagnose or identify a problem, you might need to provide IBM Support with data and information from your system. In other cases, IBM Support might provide you with tools or utilities to use for problem determination.

Sending information to IBM Support

To reduce the time that is required to resolve your problem, you can send trace and diagnostic information to IBM Support.

Procedure

To submit diagnostic information to IBM Support:

1. Open a problem management record (PMR).
2. Collect the diagnostic data that you need. Diagnostic data helps reduce the time that it takes to resolve your PMR. You can collect the diagnostic data manually or automatically:
 - Collect the data manually.
 - Collect the data automatically.
3. Compress the files by using the .zip or .tar file format.
4. Transfer the files to IBM. You can use one of the following methods to transfer the files to IBM:
 - IBM Support Assistant
 - The Service Request tool
 - Standard data upload methods: FTP, HTTP
 - Secure data upload methods: FTPS, SFTP, HTTPS
 - E-mail

If you are using a z/OS® product and you use ServiceLink / IBMLink to submit PMRs, you can send diagnostic data to IBM Support in an e-mail or by using FTP.

All of these data exchange methods are explained on the IBM Support website.

Receiving information from IBM Support

Occasionally an IBM technical-support representative might ask you to download diagnostic tools or other files. You can use FTP to download these files.

Before you begin

Ensure that your IBM technical-support representative provided you with the preferred server to use for downloading the files and the exact directory and file names to access.

Procedure

To download files from IBM Support:

1. Use FTP to connect to the site that your IBM technical-support representative provided and log in as anonymous. Use your e-mail address as the password.
2. Change to the appropriate directory:
 - a. Change to the /fromibm directory.

```
cd fromibm
```
 - b. Change to the directory that your IBM technical-support representative provided.

- ```
cd nameofdirectory
```
3. Enable binary mode for your session.  

```
binary
```
  4. Use the **get** command to download the file that your IBM technical-support representative specified.  

```
get filename.extension
```
  5. End your FTP session.  

```
quit
```

## Subscribing to Support updates

To stay informed of important information about the IBM products that you use, you can subscribe to updates.

### About this task

By subscribing to receive updates about the product, you can receive important technical information and updates for specific IBM Support tools and resources. You can subscribe to updates by using one of two approaches:

#### Social media subscriptions

The following RSS feed is available for the product:

- RSS feed for WebSphere eXtreme Scale forum

For general information about RSS, including steps for getting started and a list of RSS-enabled IBM web pages, visit the IBM Software Support RSS feeds site.

#### My Notifications

With My Notifications, you can subscribe to Support updates for any IBM product. My Notifications replaces My Support, which is a similar tool that you might have used in the past. With My Notifications, you can specify that you want to receive daily or weekly e-mail announcements. You can specify what type of information you want to receive, such as publications, hints and tips, product flashes (also known as alerts), downloads, and drivers. My Notifications enables you to customize and categorize the products about which you want to be informed and the delivery methods that best suit your needs.

### Procedure

To subscribe to Support updates:

1. Subscribe to the RSS feed for the WebSphere eXtreme Scale forum .
  - a. On the subscription page, click the RSS feed icon.
  - b. Select the option that you want to use to subscribe to the feed.
  - c. Click **Subscribe**.
2. Subscribe to My Notifications by going to the IBM Support Portal and click **My Notifications** in the **Notifications** portlet.
3. Sign in using your IBM ID and password, and click **Submit**.
4. Identify what and how you want to receive updates.
  - a. Click the **Subscribe** tab.
  - b. Select the appropriate software brand or type of hardware.
  - c. Select one or more products by name and click **Continue**.

- d. Select your preferences for how to receive updates, whether by e-mail, online in a designated folder, or as an RSS or Atom feed.
- e. Select the types of documentation updates that you want to receive, for example, new information about product downloads and discussion group comments.
- f. Click **Submit**.

## Results

Until you modify your RSS feeds and My Notifications preferences, you receive notifications of updates that you have requested. You can modify your preferences when needed; for example, if you stop using one product and begin using another product.

---

## Enabling logging

You can use logs to monitor and troubleshoot your environment.

### About this task

Logs are saved different locations and formats depending on your configuration.

### Procedure

- **Enable logs in a stand-alone environment.**

With stand-alone catalog servers, the logs are in the location where you run the **startOgServer** command. For container servers, you can use the default location or set a custom log location:

- **Default log location:** The logs are in the directory where the server command was run. If you start the servers in the *wxs\_home/bin* directory, the logs and trace files are in the *logs/<server\_name>* directories in the *bin* directory.
- **Custom log location:** To specify an alternate location for container server logs, create a properties file, such as *server.properties*, with the following contents:

```
workingDirectory=<directory>
traceSpec=
systemStreamToFileEnabled=true
```

The **workingDirectory** property is the root directory for the logs and optional trace file. WebSphere eXtreme Scale creates a directory with the name of the container server with a *SystemOut.log* file, a *SystemErr.log* file, and a trace file. To use a properties file during container startup, use the **-serverProps** option and provide the server properties file location.

- **Enable logs in WebSphere Application Server.**

See WebSphere Application Server: Enabling and disabling logging for more information.

- **Retrieve FFDC files.**

FFDC files are for IBM support to aid in debug. These files might be requested by IBM support when a problem occurs. These files are in a directory labeled, *ffdc*, and contain files that resemble the following:

```
server2_exception.log
server2_20802080_07.03.05_10.52.18_0.txt
```

## What to do next

View the log files in their specified locations. Common messages to look for in the `SystemOut.log` file are start confirmation messages, such as the following example:

```
CWOBJ1001I: ObjectGrid Server catalogServer01 is ready to process requests.
```

For more information about a specific message in the log files, see [Messages](#).

---

## Collecting trace

You can use trace to monitor and troubleshoot your environment. You must provide trace for a server when you work with IBM support.

### About this task

Collecting trace can help you monitor and fix problems in your deployment of WebSphere eXtreme Scale. How you collect trace depends on your configuration. See “Trace options” on page 470 for a list of the different trace specifications you can collect.

### Procedure

- **Collect trace within a WebSphere Application Server environment.**

If your catalog and container servers are in a WebSphere Application Server environment, see [WebSphere Application Server: Working with trace](#) for more information.

- **Collect trace with the stand-alone catalog or container server start command.**

You can set trace on a catalog service or container server by using the **-traceSpec** and **-traceFile** parameters with the **startOgServer** command. For example:

```
startOgServer.sh catalogServer -traceSpec ObjectGridPlacement=all=enabled -traceFile /home/user1/logs/trace.log
```

The **-traceFile** parameter is optional. If you do not set a **-traceFile** location, the trace file goes to the same location as the system out log files. For more information about these parameters, see the information about the `startOgServer` script in the *Administration Guide*.

- **Collect trace on the stand-alone catalog or container server with a properties file.**

To collect trace from a properties file, create a file, such as a `server.properties` file, with the following contents:

```
workingDirectory=<directory>
traceSpec=<trace_specification>
systemStreamToFileEnabled=true
```

The **workingDirectory** property is the root directory for the logs and optional trace file. If the **workingDirectory** value is not set, the default working directory is the location used to start the servers, such as `wxs_home/bin`. To use a properties file during server startup, use the **-serverProps** parameter with the **startOgServer** command and provide the server properties file location. For more information about the server properties file and how to use the file, see the information about the server properties file in the *Administration Guide*.

- **Collect trace on a stand-alone client.**

You can start trace collection on a stand-alone client by adding system properties to the startup script for the client application. In the following example, trace settings are specified for the `com.ibm.samples.MyClientProgram` application:

```
java -DtraceSettingsFile=MyTraceSettings.properties
-Djava.util.logging.manager=com.ibm.ws.bootstrap.WsLogManager
-Djava.util.logging.configureByServer=true com.ibm.samples.MyClientProgram
```

See WebSphere Application Server: Enabling trace on client and stand-alone applications for more information.

- **Collect trace with the ObjectGridManager interface.**

You can also set trace during run time on an ObjectGridManager interface. Setting trace on an ObjectGridManager interface can be used to get trace on an eXtreme Scale client while it connects to an eXtreme Scale and commits transactions. To set trace on an ObjectGridManager interface, supply a trace specification and a trace log.

```
ObjectGridManager manager = ObjectGridManagerFactory.getObjectGridManager();
...
manager.setTraceEnabled(true);
manager.setTraceFileName("logs/myClient.log");
manager.setTraceSpecification("ObjectGridReplication=all=enabled");
```

For more information about the ObjectGridManager interface, see the information about interacting with the ObjectGrid using the ObjectGridManager interface in the *Programming Guide*.

- **Collect trace on container servers with the xscmd utility.**

To collect trace with the **xscmd** utility, use the **-c setTraceSpec** command. Use the **xscmd** utility to collect trace on a stand-alone environment during run time instead of during startup. You can collect trace on all servers and catalog services or you can filter the servers based on the ObjectGrid name, and other properties. For example, to collect ObjectGridReplication trace with access to the catalog service server, run:

```
xscmd -c setTraceSpec "ObjectGridReplication=all=enabled"
```

You can also disable trace by setting the trace specification to **\*=all=disabled**.

## Results

Trace files are written to the specified location.

## Trace options

You can enable trace to provide information about your environment to IBM support.

### About trace

WebSphere eXtreme Scale trace is divided into several different components. You can specify the level of trace to use. Common levels of trace include: all, debug, entryExit, and event.

An example trace string follows:

```
ObjectGridComponent=level=enabled
```

You can concatenate trace strings. Use the \* (asterisk) symbol to specify a wildcard value, such as **ObjectGrid\*=all=enabled**. If you need to provide a trace to IBM support, a specific trace string is requested. For example, if a problem with replication occurs, the **ObjectGridReplication=debug=enabled** trace string might be requested.



## Trace specification

### **ObjectGrid**

General core cache engine.

### **ObjectGridCatalogServer**

General catalog service.

### **ObjectGridChannel**

Static deployment topology communications.

### **ObjectGridClientInfo**

DB2 client information.

### **ObjectGridClientInfoUser**

DB2 user information.

### **ObjectgridCORBA**

Dynamic deployment topology communications.

### **ObjectGridDataGrid**

The AgentManager API.

### **ObjectGridDynaCache**

The WebSphere eXtreme Scale dynamic cache provider.

### **ObjectGridEntityManager**

The EntityManager API. Use with the Projector option.

### **ObjectGridEvictors**

ObjectGrid built-in evictors.

### **ObjectGridJPA**

Java Persistence API (JPA) loaders.

### **ObjectGridJPACache**

JPA cache plug-ins.

### **ObjectGridLocking**

ObjectGrid cache entry lock manager.

### **ObjectGridMBean**

Management beans.

### **ObjectGridMonitor**

Historical monitoring infrastructure.

### **ObjectGridNative**

WebSphere eXtreme Scale native code trace, including eXtremeMemory native code.

### **ObjectGridOSGi**

The WebSphere eXtreme Scale OSGi integration components.

### **ObjectGridPlacement**

Catalog server shard placement service.

### **ObjectGridQuery**

ObjectGrid query.

### **ObjectGridReplication**

Replication service.

### **ObjectGridRouting**

Client/server routing details.

- ObjectGridSecurity**  
Security trace.
- ObjectGridSerializer**  
The DataSerializer plug-in infrastructure.
- ObjectGridStats**  
ObjectGrid statistics.
- ObjectGridTransactionManager**  
The WebSphere eXtreme Scale transaction manager.
- ObjectGridWriteBehind**  
ObjectGrid write behind.
- ObjectGridXM**  
General IBM eXtremeMemory trace.
- ObjectGridXMEviction**  
eXtremeMemory eviction trace.
- ObjectGridXMTransport**  
eXtremeMemory general transport trace.
- ObjectGridXMTransportInbound**  
eXtremeMemory inbound specific transport trace.
- ObjectGridXMTransportOutbound**  
eXtremeMemory outbound specific transport trace.
- Projector**  
The engine within the EntityManager API.
- QueryEngine**  
The query engine for the Object Query API and EntityManager Query API.
- QueryEnginePlan**  
Query plan trace.
- TCPChannel**  
The IBM eXtremeIO TCP/IP channel.
- XsByteBuffer**  
WebSphere eXtreme Scale byte buffer trace.

---

## Analyzing log and trace data

You can use the log analysis tools to analyze how your runtime is performing and solve problems that occur in the environment.

### About this task

You can generate reports from the existing log and trace files in the environment. These visual reports can be used for the following purposes:

- **To analyze runtime environment status and performance:**
  - Deployment environment consistency
  - Logging frequency
  - Running topology versus configured topology
  - Unplanned topology changes
  - Quorum status
  - Partition replication status

- Statistics of memory, throughput, processor usage, and so on
- **To troubleshoot problems in the environment:**
  - Topology views at specific points in time
  - Statistics of memory, throughput, processor usage during client failures
  - Current fix pack levels, tuning settings
  - Quorum status

## Log analysis overview

You can use the **xsLogAnalyzer** tool to help troubleshoot issues in the environment.

### All failover messages

Displays the total number of failover messages as a chart over time. Also displays a list of the failover messages, including the servers that have been affected

### All eXtreme Scale critical messages

Displays message IDs along with the associated explanations and user actions, which can save you the time from searching for messages.

### All exceptions

Displays the top five exceptions, including the messages and how many times they occurred, and what servers were affected by the exception.

### Topology summary

Displays a diagram of how your topology is configured according to the log files. You can use this summary to compare to your actual configuration, possibly identifying configuration errors.

### Topology consistency: Object Request Broker (ORB) comparison table

Displays ORB settings in the environment. You can use this table to help determine if the settings are consistent across your environment.

### Event timeline view

Displays a timeline diagram of different actions that have occurred on the data grid, including life cycle events, exceptions, critical messages, and first-failure data capture (FFDC) events.

## Running log analysis

You can run the **xsLogAnalyzer** tool on a set of log and trace files from any computer.

### Before you begin

- Enable logs and trace. See “Enabling logging” on page 468 and “Collecting trace” on page 469 for more information.
- Collect your log files. The log files can be in various locations depending on how you configured them. If you are using the default log settings, you can get the log files from the following locations:

- In a stand-alone installation: *wxs\_install\_root/bin/logs/<server\_name>*
- In an installation that is integrated with WebSphere Application Server: *was\_root/logs/<server\_name>*
- Collect your trace files. The trace files can be in various locations depending on how you configured them. If you are using the default trace settings, you can get the trace files from the following locations:
  - In a stand-alone installation: If no specific trace value is set, the trace files are written to the same location as the system out log files.
  - In an installation that is integrated with WebSphere Application Server: *was\_root/profiles/server\_name/logs*.

Copy the log and trace files to the computer from which you are planning to use the log analyzer tool.

- If you want to create custom scanners in your generated report, create a scanner specifications properties file and configuration file before you run the tool. For more information, see “Creating custom scanners for log analysis” on page 475.

## Procedure

1. Run the **xsLogAnalyzer** tool.

The script is in the following locations :

- In a stand-alone installation: *wxs\_install\_root/ObjectGrid/bin*
- In an installation that is integrated with WebSphere Application Server: *was\_root/bin*

**Tip:** If your log files are large, consider using the **-startTime**, **-endTime**, and **-maxRecords** parameters when you run the report to restrict the number of log entries that are scanned. Using these parameters when you run the report makes the reports easier to read and run more effectively. You can run multiple reports on the same set of log files.

```
xsLogAnalyzer.sh|bat -logsRoot c:\myxslogs -outDir c:\myxslogs\out
-startTime 11.09.27_15.10.56.089 -endTime 11.09.27_16.10.56.089 -maxRecords 100
```

### **-logsRoot**

Specifies the absolute path to the log directory that you want to evaluate (required).

### **-outDir**

Specifies an existing directory to write the report output. If you do not specify a value, the report is written to the root location of the **xsLogAnalyzer** tool.

### **-startTime**

Specifies the start time to evaluate in the logs. The date is in the following format: *year.month.day\_hour.minute.second.millisecond*

### **-endTime**

Specifies the end time to evaluate in the logs. The date is in the following format: *year.month.day\_hour.minute.second.millisecond*

**-trace** Specifies a trace string, such as *ObjectGrid\*=all=enabled*.

### **-maxRecords**

Specifies the maximum number of records to generate in the report. The default is 100. If you specify the value as 50, the first 50 records are generated for the specified time period.

2. Open the generated files. If you did not define an output directory, the reports are generated in a folder called `report_date_time`. To open the main page of the reports, open the `index.html` file.
3. Use the reports to analyze the log data. Use the following tips to maximize the performance of the report displays:
  - To maximize the performance of queries on the log data, use as specific information as possible. For example, a query for `server` takes much longer to run and returns more results than `server_host_name`.
  - Some views have a limited number of data points that are displayed at one time. You can adjust the segment of time that is being viewed by changing the current data, such as start and end time, in the view.

## What to do next

For more information about troubleshooting the **xsLogAnalyzer** tool and the generated reports, see “Troubleshooting log analysis” on page 476.

## Creating custom scanners for log analysis

You can create custom scanners for log analysis. After you configure the scanner, the results are generated in the reports when you run the **xsLogAnalyzer** tool. The custom scanner scans the logs for event records based on the regular expressions that you specified.

### Procedure

1. Create a scanner specifications properties file that specifies the general expression to run for the custom scanner.
  - a. Create and save a properties file. The file must be in the `loganalyzer_root/config/custom` directory. You can name the file as: you like. The file is used by the new scanner, so naming the scanner in the properties file is useful, for example:  
`my_new_server_scanner_spec.properties`.
  - b. Include the following properties in the `my_new_server_scanner_spec.properties` file:  
`include.regular_expression = REGULAR_EXPRESSION_TO_SCAN`

The `REGULAR_EXPRESSION_TO_SCAN` variable is a regular expression on which to filter the log files.

Example: To scan for instances of lines that contain both the "xception" and "rerror" strings regardless of the order, set the **include.regular\_expression** property to the following value:

```
include.regular_expression = (xception.+rerror)|(rerror.+xception)
```

This regular expression causes events to be recorded if the string "rerror" comes before or after the "xception" string.

Example: To scan through each line in the logs for instances of lines that contain either the phrase "xception" or the phrase "rerror" strings regardless of the order, set the **include.regular\_expression** property to the following value:

```
include.regular_expression = (xception)|(rerror)
```

This regular expression causes events to be recorded if the either the "rerror" string or the "xception" string exist.

2. Create a configuration file that the **xsLogAnalyzer** tool uses to create the scanner.

- a. Create and save a configuration file. The file must be in the *logalyzer\_root/config/custom* directory. You can name the file as *scanner\_nameScanner.config*, where *scanner\_name* is a unique name for the new scanner. For example, you might name the file *serverScanner.config*
- b. Include the following properties in the *scanner\_nameScanner.config* file:  
`scannerSpecificationFiles = LOCATION_OF_SCANNER_SPECIFICATION_FILE`

The *LOCATION\_OF\_SCANNER\_SPECIFICATION\_FILE* variable is the path and location of the specification file that you created in the previous step. For example: *logalyzer\_root/config/custom/my\_new\_scanner\_spec.properties*. You can also specify multiple scanner specification files by using a semi-colon separated list:

```
scannerSpecificationFiles = LOCATION_OF_SCANNER_SPECIFICATION_FILE1;LOCATION_OF_SCANNER_SPECIFICATION_FILE2
```

3. Run the **xsLogAnalyzer** tool. For more information, see “Running log analysis” on page 473.

## Results

After you run the **xsLogAnalyzer** tool, the report contains new tabs in the report for the custom scanners that you configured. Each tab contains the following views:

**Charts** A plotted graph that illustrates recorded events. The events are displayed in the order in which the events were found.

**Tables** A tabular representation of the recorded events.

**Summary reports**

## Troubleshooting log analysis

Use the following troubleshooting information to diagnose and fix problems with the **xsLogAnalyzer** tool and its generated reports.

### Procedure

- **Problem:** Out of memory conditions occur when you are using the **xsLogAnalyzer** tool to generate reports. An example of an error that might occur follows: `java.lang.OutOfMemoryError: GC overhead limit exceeded`.

**Solution:** The **xsLogAnalyzer** tool runs within a Java virtual machine (JVM). You can configure the JVM to increase the heap size before you run the **xsLogAnalyzer** tool by specifying some settings when you run the tool. Increasing the heap size enables more event records to be stored in JVM memory. Start with a setting of 2048M, assuming the operating system has enough main memory. On the same command-line instance in which you are planning to run the **xsLogAnalyzer** tool, set the maximum JVM heap size:

```
java -XmxHEAP_SIZEm
```

The *HEAP\_SIZE* value can be any integer and represents the number of megabytes that are allocated to JVM heap. For example, you might run `java -Xmx2048m`. If the out of memory messages continue, or you do not have the resources to allocate 2048m or more of memory, limit the number of events that are being held in the heap. You can limit the number of events in the heap up by passing the **-maxRecords** parameter to the **xsLogAnalyzer** command

- **Problem:** When you open a generated report from the **xsLogAnalyzer** tool, the browser hangs or does not load the page.

**Cause:** The generated HTML files are too large and cannot be loaded by the browser. These files are large because the scope of the log files that you are analyzing is too broad.

**Solution:** Consider using the `-startTime`, `-endTime`, and `-maxRecords` parameters when you run the `xsLogAnalyzer` tool to restrict the number of log entries that are scanned. Using these parameters when you run the report makes the reports easier to read and run more effectively. You can run multiple reports on the same set of log files.

---

## Troubleshooting client connectivity

There are several common problems specific to clients and client connectivity that you can solve as described in the following sections.

### Procedure

- **Problem:** If you are using the EntityManager API or byte array maps with the COPY\_TO\_BYTES copy mode, client data access methods result in various serialization-related exceptions or a NullPointerException exception.
  - The following error occurs when you are using the COPY\_TO\_BYTES copy mode:

```
java.lang.NullPointerException
 at com.ibm.ws.objectgrid.map.BaseMap$BaseMapObjectTransformer2.inflateObject(BaseMap.java:5278)
 at com.ibm.ws.objectgrid.map.BaseMap$BaseMapObjectTransformer.inflateValue(BaseMap.java:5155)
```

- The following error occurs when you are using the EntityManager API:

```
java.lang.NullPointerException
 at com.ibm.ws.objectgrid.em.GraphTraversalHelper.fluffFetchMD(GraphTraversalHelper.java:323)
 at com.ibm.ws.objectgrid.em.GraphTraversalHelper.fluffFetchMD(GraphTraversalHelper.java:343)
 at com.ibm.ws.objectgrid.em.GraphTraversalHelper.getObjectGraph(GraphTraversalHelper.java:102)
 at com.ibm.ws.objectgrid.ServerCoreEventProcessor.getFromMap(ServerCoreEventProcessor.java:709)
 at com.ibm.ws.objectgrid.ServerCoreEventProcessor.processGetRequest(ServerCoreEventProcessor.java:323)
```

**Cause:** The EntityManager API and COPY\_TO\_BYTES copy mode use a metadata repository that is embedded in the data grid. When clients connect, the data grid stores the repository identifiers in the client and caches the identifiers for the duration of the client connection. If you restart the data grid, you lose all metadata and the regenerated identifiers do not match the cached identifiers on the client.

**Solution:** If you are using the EntityManager API or the COPY\_TO\_BYTES copy mode, disconnect and reconnect all of the clients if the ObjectGrid is stopped and restarted. Disconnecting and reconnecting the clients refreshes the metadata identifier cache. You can disconnect clients by using the `ObjectGridManager.disconnect` method or the `ObjectGrid.destroy` method.

- **Problem:** The client hangs during a getObjectGrid method call.

A client might seem to hang when calling the getObjectGrid method on the ObjectGridManager or throw an exception: `com.ibm.websphere.projector.MetadataException`. The EntityMetadata repository is not available and the timeout threshold is reached.

**Cause:** The reason is the client is waiting for the entity metadata on the ObjectGrid server to become available.

**Solution:** This error can occur when a container server has been started, but placement has not yet started. Take the following actions:

- Examine the deployment policy for the ObjectGrid and verify that the number of active containers is greater than or equal to both the `numInitialContainers` and `minSyncReplicas` attributes in the deployment policy descriptor file.

- Examine the setting for the **placementDeferralInterval** property in the container server server properties file to see how much time needs to pass before placement operations occur.
- If you used the **xscmd -c suspendBalancing** command to stop the balancing of shards for a specific data grid and map set, use the **xscmd -c resumeBalancing** to start balancing again.

---

## Troubleshooting cache integration

Use this information to troubleshoot issues with your cache integration configuration, including HTTP session and dynamic cache configurations.

### Procedure

- **Problem:** HTTP session IDs are not being reused.

**Cause:** You can reuse session IDs. If you create a data grid for session persistence in Version 7.1.1 or later, session ID reuse is automatically enabled. However, if you created prior configurations, this setting might already be set with the wrong value.

**Solution:** Check the following settings to verify that you have HTTP session ID reuse enabled:

- The `reuseSessionId` property in the `splicer.properties` file must be set to `true`.
- The `HttpSessionIdReuse` custom property value must be set to `true`. This custom property might be set on one of the following paths in the WebSphere Application Server administrative console:
  - **Servers > *server\_name* > Session management > Custom properties**
  - **Dynamic clusters > *dynamic\_cluster\_name* > Server template > Session management > Custom properties**
  - **Servers > Server Types > WebSphere application servers > *server\_name*, and then, under Server Infrastructure, click **Java and process management > Process definition > Java virtual machine > Custom properties****
  - **Servers > Server Types > WebSphere application servers > *server\_name* > Web container settings > Web container**

If you update any custom property values, reconfigure eXtreme Scale session management so the `splicer.properties` file becomes aware of the change.

- **Problem:** When you are using a data grid to store HTTP sessions and the transaction load is high, a `CWOBJ0006W` message displays in the `SystemOut.log` file.

```
CWOBJ0006W: An exception occurred:
com.ibm.websphere.objectgrid.ObjectGridRuntimeException:
java.util.ConcurrentModificationException
```

This message occurs only when the **replicationInterval** parameter in the `splicer.properties` file is set to a value greater than zero and the Web application modifies a List object that was set as an attribute on the `HTTPSession`.

**Solution:** Clone the attribute that contains the modified List object and put the cloned attribute into the session object.



---

## Troubleshooting the JPA cache plug-in

Use this information to troubleshoot issues with your JPA cache plug-in configuration. These problems can occur in both Hibernate and OpenJPA configurations.

### Procedure

- **Problem:** The following exception displays: `CacheException: Failed to get ObjectGrid server.`

With either an `EMBEDDED` or `EMBEDDED_PARTITION` **ObjectGridType** attribute value, the eXtreme Scale cache tries to obtain a server instance from the run time. In a Java Platform, Standard Edition environment, an eXtreme Scale server with embedded catalog service is started. The embedded catalog service tries to listen to port 2809. If that port is being used by another process, the error occurs.

**Solution:** If external catalog service endpoints are specified, for example, with the `objectGridServer.properties` file, this error occurs if the host name or port is specified incorrectly. Correct the port conflict.

- **Problem:** The following exception displays: `CacheException: Failed to get REMOTE ObjectGrid for configured REMOTE ObjectGrid. objectGridName = [ObjectGridName], PU name = [persistenceUnitName]`

This error occurs because the cache cannot get the ObjectGrid instance from the provided catalog service end points.

**Solution:** This problem typically occurs because of an incorrect host name or port.

- **Problem:** The following exception displays: `CacheException: Cannot have two PUs [persistenceUnitName_1, persistenceUnitName_2] configured with same ObjectGridName [ObjectGridName] of EMBEDDED ObjectGridType`

This exception results if you have many persistence units configured and the eXtreme Scale caches of these units are configured with the same ObjectGrid name and `EMBEDDED ObjectGridType` attribute value. These persistence unit configurations could be in the same or different `persistence.xml` files.

**Solution:** You must verify that the ObjectGrid name is unique for each persistence unit when the **ObjectGridType** attribute value is `EMBEDDED`.

- **Problem:** The following exception displays: `CacheException: REMOTE ObjectGrid [ObjectGridName] does not include required BackingMaps [mapName_1, mapName_2,...]`

With a `REMOTE ObjectGrid` type, if the obtained client-side ObjectGrid does not have complete entity backing maps to support the persistence unit cache, this exception occurs. For example, five entity classes are listed in the persistence unit configuration, but the obtained ObjectGrid only has two BackingMaps. Even though the obtained ObjectGrid might have 10 BackingMaps, if any one of the five required entity BackingMaps are not found in the 10 backing maps, this exception still occurs.

**Solution:** Make sure that your backing map configuration supports the persistence unit cache.

---

## Troubleshooting IBM eXtremeMemory and IBM eXtremeIO

Use the following information to troubleshoot eXtremeMemory and eXtremeIO.

## Procedure

**Problem:** If the shared resource, `libstdc++.so.5`, is not installed, then when you start the container server, IBM eXtremeMemory native libraries do not load.

**Linux Symptom:** On a Linux 64-bit operating system, if you try to start a container server with the `enableXM` server property set to `true`, and the `libstdc++.so.5` shared resource is not installed, you get an error similar to the following example:

```
00000000 Initialization W CW0BJ0006W: An exception occurred: java.lang.reflect.InvocationTargetException
at sun.reflect.NativeConstructorAccessorImpl.newInstance0(Native Method)
at sun.reflect.NativeConstructorAccessorImpl.newInstance(NativeConstructorAccessorImpl.java:56)
at sun.reflect.DelegatingConstructorAccessorImpl.newInstance(DelegatingConstructorAccessorImpl.java:39)
at java.lang.reflect.Constructor.newInstance(Constructor.java:527)
at com.ibm.websphere.objectgrid.server.ServerFactory.initialize(ServerFactory.java:350)
at com.ibm.websphere.objectgrid.server.ServerFactory$2.run(ServerFactory.java:303)
at java.security.AccessController.doPrivileged(AccessController.java:202)
at com.ibm.websphere.objectgrid.server.ServerFactory.getInstance(ServerFactory.java:301)
at com.ibm.ws.objectgrid.InitializationService.main(InitializationService.java:302)

Caused by: com.ibm.websphere.objectgrid.ObjectGridRuntimeException: java.lang.UnsatisfiedLinkError:
OffheapMapdbg (Not found in java.library.path)
at com.ibm.ws.objectgrid.ServerImpl.<init>(ServerImpl.java:1033)
... 9 more Caused by: java.lang.UnsatisfiedLinkError: OffheapMapdbg (Not found in java.library.path)
at java.lang.ClassLoader.loadLibraryWithPath(ClassLoader.java:1011)
at java.lang.ClassLoader.loadLibraryWithClassLoader(ClassLoader.java:975)
at java.lang.System.loadLibrary(System.java:469)
at com.ibm.ws.objectgrid.io.offheap.ObjectGridHashTableOH.initializeNative(ObjectGridHashTableOH.java:112)
at com.ibm.ws.objectgrid.io.offheap.ObjectGridHashTableOH.<clinit>(ObjectGridHashTableOH.java:87)
at java.lang.J9VMInternals.initializeImpl(Native Method)
at java.lang.J9VMInternals.initialize(J9VMInternals.java:200)
at com.ibm.ws.objectgrid.ServerImpl.<init>(ServerImpl.java:1028)
... 9 more
```

**Cause:** The shared resource `libstdc++.so.5` has not been installed.

**Diagnosing the problem:** To verify that the resource `libstdc++.so.5` is installed, issue the following command from the `ObjectGrid/native` directory of your installation:

```
ldd lib0ffheapMap.so
```

If you do not have the shared library installed, you get the following error:

```
ldd lib0ffheapMap.so
libstdc++.so.5 => not found
```

**Resolving the problem:** Use the package installer of your 64-bit Linux distribution to install the required resource file. The package might be listed as `compat-libstdc++-33.x86_64` or `libstdc++5`. After installing the required resource, verify that the `libstdc++5` package is installed by issuing the following command from the `ObjectGrid` directory of your installation:

```
ldd lib0ffheapMap.so
```

---

## Troubleshooting administration

Use the following information to troubleshoot administration, including starting and stopping servers, using the `xscmd` utility, and so on.

### Procedure

- **Problem:** Administration scripts are missing from the `profile_root/bin` directory of a WebSphere Application Server installation.

**Cause:** When you update the installation, new script files do not automatically get installed in the profiles.

**Solution:** If you want to run a script from your `profile_root/bin` directory, unaugment and reaugment the profile with the latest release. For more

information, see Unaugmenting a profile using the command prompt and Creating and augmenting profiles for WebSphere eXtreme Scale.

- **Problem:** When you are running a `xscmd` command, the following message is printed to the screen:

```
java.lang.IllegalStateException: Placement service MBean not available.
[]
 at
com.ibm.websphere.samples.objectgrid.admin.OGAdmin.main(OGAdmin.java:1449)
 at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
 at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:60)
 at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:37)
 at java.lang.reflect.Method.invoke(Method.java:611)
 at com.ibm.ws.bootstrap.WSLauncher.main(WSLauncher.java:267)
Ending at: 2011-11-10 18:13:00.000000484
```

**Cause:** A connection problem has occurred with the catalog server.

**Solution:** Verify that your catalog servers are running and are available through the network. This message can also occur when you have a catalog service domain defined, but less than two catalog servers are running. The environment is not available until two catalog servers are started.

- **8.5+ Problem:** When you are running a `xscmd` command, the following message is printed to the screen:

```
CWXSIO066E: Unmatched argument .. was detected
```

**Cause:** You entered a command format that the `xscmd` utility did not recognize.

**Solution:** Check the format of the command. You might encounter this issue when running regular expressions with the `-c findbyKey` command. See Querying and invalidating data for more information.

---

## Troubleshooting multiple data center configurations

Use this information to troubleshoot multiple data center configurations, including linking between catalog service domains.

### Procedure

**Problem:** Data is missing in one or more catalog service domains. For example, you might run the `xscmd -c establishLink` command. When you look at the data for each linked catalog service domain, the data looks different, for example from the `xscmd -c showMapSizes` command.

**Solution:** You can troubleshoot this problem with the `xscmd -c showLinkedPrimaries` command. This command prints out each primary shard, and including which foreign primaries are linked.

In the described scenario, you might discover from running the `xscmd -c showLinkedPrimaries` command that the first catalog service domain primary shards are linked to the second catalog service domain primary shards, but the second catalog service domain does not have links to the first catalog service domain. You might consider rerunning the `xscmd -c establishLink` command from the second catalog service domain to the first catalog service domain.

---

## Troubleshooting loaders

Use this information to troubleshoot issues with your database loaders.

### Procedure

- **Problem:** The loader is unable to communicate with the database. A `LoaderNotAvailableException` exception occurs.

**Explanation:** The loader plug-in can fail when it is unable to communicate to the database back end. This failure can happen if the database server or the network connection is down. The write-behind loader queues the updates and tries to push the data changes to the loader periodically. The loader must notify the ObjectGrid run time that there is a database connectivity problem by throwing a `LoaderNotAvailableException` exception.

**Solution:** The Loader implementation must be able to distinguish a data failure or a physical loader failure. Data failure should be thrown or rethrown as a `LoaderException` or an `OptimisticCollisionException`, but a physical loader failure must be thrown or rethrown as a `LoaderNotAvailableException`. ObjectGrid handles these two exceptions differently:

- If a `LoaderException` is caught by the write-behind loader, the write-behind loader considers the exception a failure, such as duplicate key failure. The write-behind loader unbatches the update, and tries the update one record at one time to isolate the data failure. If a `LoaderException` is caught again during the one record update, a failed update record is created and logged in the failed update map.
- If a `LoaderNotAvailableException` is caught by the write-behind loader, the write-behind loader considers it failed because it cannot connect to the database end, for example, the database back-end is down, a database connection is not available, or the network is down. The write-behind loader waits for 15 seconds and then try the batch update to the database again.

The common mistake is to throw a `LoaderException` while a `LoaderNotAvailableException` must be thrown. All the records queued in the write-behind loader become failed update records, which defeats the purpose of back-end failure isolation.

- **Problem:** When you are using an OpenJPA loader with DB2 in WebSphere Application Server, a closed cursor exception occurs.

The following exception is from DB2 in the `org.apache.openjpa.persistence.PersistenceException` log file:  
[jcc][t4][10120][10898][3.57.82] Invalid operation: result set is closed.

**Solution:** By default, the application server configures the `resultSetHoldability` custom property with a value of 2 (`CLOSE_CURSORS_AT_COMMIT`). This property causes DB2 to close its `resultSet/cursor` at transaction boundaries. To remove the exception, change the value of the custom property to 1 (`HOLD_CURSORS_OVER_COMMIT`). Set the `resultSetHoldability` custom property on the following path in the WebSphere Application Server cell:

**Resources > JDBC provider > DB2 Universal JDBC Driver Provider > DataSources > *data\_source\_name* > Custom properties > New.**

- **Problem** DB2 displays an exception: The current transaction has been rolled back because of a deadlock or timeout. Reason code "2".. `SQLCODE=-911, SQLSTATE=40001, DRIVER=3.50.152`

This exception occurs because of a lock contention problem when you are running with OpenJPA with DB2 in WebSphere Application Server. The default isolation level for WebSphere Application Server is Repeatable Read (RR), which obtains long-lived locks with DB2.**Solution:**

Set the isolation level to Read Committed to reduce the lock contention. Set the `webSphereDefaultIsolationLevel` data source custom property to set the isolation level to 2(`TRANSACTION_READ_COMMITTED`) on the following path in the WebSphere Application Server cell: **Resources > JDBC provider > JDBC\_provider > Data sources > *data\_source\_name* > Custom properties >**

**New.** For more information about the `webSphereDefaultIsolationLevel` custom property and transaction isolation levels, see Requirements for setting data access isolation levels.

- **Problem:** When you are using the preload function of the `JPALoader` or `JPAEntityLoader`, the following `CWOBJ1511I` message does not display for the partition in a container server: `CWOBJ1511I:`

`GRID_NAME:MAPSET_NAME:PARTITION_ID (primary) is open for business.`

Instead, a `TargetNotAvailableException` exception occurs in the container server, which activates the partition that is specified by the `preloadPartition` property.

**Solution:** Set the `preloadMode` attribute to `true` if you use a `JPALoader` or `JPAEntityLoader` to preload data into the map. If the `preloadPartition` property of the `JPALoader` and `JPAEntityLoader` is set to a value between `0` and `total_number_of_partitions - 1`, then the `JPALoader` and `JPAEntityLoader` try to preload the data from backend database into the map. The following snippet of code illustrates how the `preloadMode` attribute is set to enable asynchronous preload:

```
BackingMap bm = og.defineMap("map1");
bm.setPreloadMode(true);
```

You can also set the `preloadMode` attribute by using an XML file as illustrated in the following example:

```
<backingMap name="map1" preloadMode="true" pluginCollectionRef="map1"
lockStrategy="OPTIMISTIC" />
```

---

## Troubleshooting deadlocks

The following sections describe some of the most common deadlock scenarios and suggestions on how to avoid them.

### Before you begin

Implement exception handling in your application. See “Implementing exception handling in locking scenarios” on page 238 for more information.

The following exception displays as a result:

```
com.ibm.websphere.objectgrid.plugins.LockDeadlockException: Message
```

This message represents the string that is passed as a parameter when the exception is created and thrown.

### Procedure

- **Problem:** `LockTimeoutException` exception.

**Description:** When a transaction or client asks for a lock to be granted for a specific map entry, the request often waits for the current client to release the lock before the request is submitted. If the lock request remains idle for an extended period of time, and a lock is never granted, `LockTimeoutException` exception is created to prevent a deadlock, which is described in more detail in the following section. You are more likely to see this exception when using a pessimistic locking strategy, because the lock never releases until the transaction commits.

#### Retrieve more details:

The `LockTimeoutException` exception contains the `getLockRequestQueueDetails` method, which returns a string. You can use this method to see a detailed

description of the situation that triggers the exception. The following is an example of code that catches the exception, and displays an error message.

```
try {
 ...
}
catch (LockTimeoutException lte) {
 System.out.println(lte.getLockRequestQueueDetails());
}
```

The output result is:

```
lock request queue
->[TX:163C269E-0105-4000-E0D7-5B3B090A571D, state =
 Granted 5348 milli-seconds ago, mode = U]
->[TX:163C2734-0105-4000-E024-5B3B090A571D, state =
 Waiting for 5348 milli-seconds, mode = U]
->[TX:163C328C-0105-4000-E114-5B3B090A571D, state =
 Waiting for 1402 milli-seconds, mode = U]
```

If you receive the exception in an ObjectGridException exception catch block, the following code determines the exception and displays the queue details. It also uses the findRootCause utility method.

```
try {
 ...
}
catch (ObjectGridException oe) {
 Throwable root = findRootCause(oe);
 if (root instanceof LockTimeoutException) {
 LockTimeoutException lte = (LockTimeoutException)root;
 System.out.println(lte.getLockRequestQueueDetails());
 }
}
```

**Solution:** A LockTimeoutException exception prevents possible deadlocks in your application. An exception of this type results when the exception waits a set amount of time. You can set the amount of time that the exception waits by using the setLockTimeout(int) method, which is available for the BackingMap. If a deadlock does not actually exist in your application, adjust the lock timeout to avoid the LockTimeoutException.

The following code shows how to create an ObjectGrid object, define a map, and set its LockTimeout value to 30 seconds:

```
ObjectGrid objGrid = new ObjectGrid();
BackingMap bMap = objGrid.defineMap("MapName");
bMap.setLockTimeout(30);
```

Use the previous hardcoded example to set ObjectGrid and map properties. If you create ObjectGrid from an XML file, set the **LockTimeout** attribute within the backingMap element. The following is an example of a backingMap element that sets a map LockTimeout value to 30 seconds.

```
<backingMap name="MapName" lockStrategy="PESSIMISTIC" lockTimeout="30">
```

- **Problem:** Single key deadlocks.

**Description:** The following scenarios describe how deadlocks can occur when a single key is accessed using a S lock and later updated. When this happens from two transactions simultaneously, it results in a deadlock.

Table 15. Single key deadlocks scenario

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.get(key1)	map.get(key1)	S lock granted to both transactions for key1.

Table 15. Single key deadlocks scenario (continued)

	Thread 1	Thread 2	
3	map.update(Key1,v)		No U lock. Update performed in transactional cache.
4		map.update(key1,v)	No U lock. Update performed in the transactional cache
5	session.commit()		Blocked: The S lock for key1 cannot be upgraded to an X lock because Thread 2 has an S lock.
6		session.commit()	Deadlock: The S lock for key1 cannot be upgraded to an X lock because T1 has an S lock.

Table 16. Single key deadlocks, continued

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.get(key1)		S lock granted for key1
3	map.getForUpdate(key1,v)		S lock is upgraded to a U lock for key1.
4		map.get(key1)	S lock granted for key1.
5		map.getForUpdate(key1,v)	Blocked: T1 already has U lock.
6	session.commit()		Deadlock: The U lock for key1 cannot be upgraded.
7		session.commit()	Deadlock: The S lock for key1 cannot be upgraded.

Table 17. Single key deadlocks, continued

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establish an independent transaction
2	map.get(key1)		S lock granted for key1.
3	map.getForUpdate(key1,v)		S lock is upgraded to a U lock for key1
4		map.get(key1)	S lock is granted for key1.
5		map.getForUpdate(key1,v)	Blocked: Thread 1 already has a U lock.
6	session.commit()		Deadlock: The U lock for key1 cannot be upgraded to an X lock because Thread 2 has an S lock.

If the ObjectMap.getForUpdate is used to avoid the S lock, then the deadlock is avoided:

Table 18. Single key deadlocks, continued

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.getForUpdate(key1)		U lock granted to thread 1 for key1.
3		map.getForUpdate(key1)	U lock request is blocked.
4	map.update(key1,v)	<blocked>	

Table 18. Single key deadlocks, continued (continued)

	Thread 1	Thread 2	
5	session.commit()	<blocked>	The U lock for key1 can be successfully upgraded to an X lock.
6		<released>	The U lock is finally granted to key1 for thread 2.
7		map.update(key2,v)	U lock granted to thread 2 for key2.
8		session.commit()	The U lock for key1 can successfully be upgraded to an X lock.

**Solutions:**

1. Use the getForUpdate method instead of get to acquire a U lock instead of an S lock.
2. Use a transaction isolation level of read committed to avoid holding S locks. Reducing the transaction isolation level increases the possibility of non-repeatable reads. However, non-repeatable reads from one client are only possible if the transaction cache is explicitly invalidated by the same client.
3. Use the optimistic lock strategy. Using the optimistic lock strategy requires handling optimistic collision exceptions.

- **Problem:** Ordered multiple key deadlock

**Description:** This scenario describes what happens if two transactions attempt to update the same entry directly and hold S locks to other entries.

Table 19. Ordered multiple key deadlock scenario

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.get(key1)	map.get(key1)	S lock granted to both transactions for key1.
3	map.get(key2)	map.get(key2)	S lock granted to both transactions for key2.
4	map.update(key1,v)		No U lock. Update performed in transactional cache.
5		map.update(key2,v)	No U lock. Update performed in transactional cache.
6.	session.commit()		Blocked: The S lock for key 1 cannot be upgraded to an X lock because thread 2 has an S lock.
7		session.commit()	Deadlock: The S lock for key 2 cannot be upgraded because thread 1 has an S lock.

You can use the `ObjectMap.getForUpdate` method to avoid the S lock, then you can avoid the deadlock:

Table 20. Ordered multiple key deadlock scenario, continued

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.getForUpdate(key1)		U lock granted to transaction T1 for key1.
3		map.getForUpdate(key1)	U lock request is blocked.



Table 20. Ordered multiple key deadlock scenario, continued (continued)

	Thread 1	Thread 2	
4	map.get(key2)	<blocked>	S lock granted for T1 for key2.
5	map.update(key1,v)	<blocked>	
6	session.commit()	<blocked>	The U lock for key1 can be successfully upgraded to an X lock.
7		<released>	The U lock is finally granted to key1 for T2
8		map.get(key2)	S lock granted to T2 for key2.
9		map.update(key2,v)	U lock granted to T2 for key2.
10		session.commit()	The U lock for key1 can be successfully upgraded to an X lock.

**Solutions:**

1. Use the getForUpdate method instead of the get method to acquire a U lock directly for the first key. This strategy works only if the method order is deterministic.
  2. Use a transaction isolation level of read committed to avoid holding S locks. This solution is the easiest to implement if the method order is not deterministic. Reducing the transaction isolation level increases the possibility of non-repeatable reads. However, non-repeatable reads are only possible if the transaction cache is explicitly invalidated.
  3. Use the optimistic lock strategy. Using the optimistic lock strategy requires handling optimistic collision exceptions.
- **Problem:** Out of order with U lock

**Description:** If the order in which keys are requested cannot be guaranteed, then a deadlock can still occur.

Table 21. Out of order with U lock scenario

	Thread 1	Thread 2	
1	session.begin()	session.begin()	Each thread establishes an independent transaction.
2	map.getforUpdate(key1)	map.getForUpdate(key2)	U locks successfully granted for key1 and key2.
3	map.get(key2)	map.get(key1)	S lock granted for key1 and key2.
4	map.update(key1,v)	map.update(key2,v)	
5	session.commit()		The U lock cannot be upgraded to an X lock because T2 has an S lock.
6		session.commit()	The U lock cannot be upgraded to an X lock because T1 has an S lock.

**Solutions:**

1. Wrap all work with a single global U lock (mutex). This method reduces concurrency, but handles all scenarios when access and order is non-deterministic.
2. Use a transaction isolation level of read committed to avoid holding S locks. This solution is the easiest to implement if the method order is not deterministic and provides the greatest amount of concurrency. Reducing the

transaction isolation level increases the possibility of non-repeatable reads. However, non-repeatable reads are only possible if the transaction cache is explicitly invalidated.

3. Use the optimistic lock strategy. Using the optimistic lock strategy requires handling optimistic collision exceptions.

---

## IBM Support Assistant for WebSphere eXtreme Scale

You can use the IBM Support Assistant to collect data, analyze symptoms, and access product information.

### IBM Support Assistant Lite

IBM Support Assistant Lite for WebSphere eXtreme Scale provides automatic data collection and symptom analysis support for problem determination scenarios.

IBM Support Assistant Lite reduces the amount of time it takes to reproduce a problem with the proper Reliability, Availability, and Serviceability tracing levels set (trace levels are set automatically by the tool) to streamline problem determination. If you need further assistance, IBM Support Assistant Lite also reduces the effort required to send the appropriate log information to IBM Support.

IBM Support Assistant Lite is included in each installation of WebSphere eXtreme Scale Version 7.1.0

### IBM Support Assistant

IBM® Support Assistant (ISA) provides quick access to product, education, and support resources that can help you answer questions and resolve problems with IBM software products on your own, without needing to contact IBM Support. Different product-specific plug-ins let you customize IBM Support Assistant for the particular products you have installed. IBM Support Assistant can also collect system data, log files, and other information to help IBM Support determine the cause of a particular problem.

IBM Support Assistant is a utility to be installed on your workstation, not directly onto the WebSphere eXtreme Scale server system itself. The memory and resource requirements for the Assistant could negatively affect the performance of the WebSphere eXtreme Scale server system. The included portable diagnostic components are designed for minimal impact to the normal operation of a server.

You can use IBM Support Assistant to help you in the following ways:

- To search through IBM and non-IBM knowledge and information sources across multiple IBM products to answer a question or solve a problem
- To find additional information through product-specific Web resources; including product and support home pages, customer news groups and forums, skills and training resources and information about troubleshooting and commonly asked questions
- To extend your ability to diagnose product-specific problems with targeted diagnostic tools available in the Support Assistant
- To simplify collection of diagnostic data to help you and IBM resolve your problems (collecting either general or product/symptom-specific data)

- To help in reporting of problem incidents to IBM Support through a customized online interface, including the ability to attach the diagnostic data referenced above or any other information to new or existing incidents

Finally, you can use the built-in Updater facility to obtain support for additional software products and capabilities as they become available. To set up IBM Support Assistant for use with WebSphere eXtreme Scale, first install IBM Support Assistant using the files provided in the downloaded image from the IBM Support Overview Web page at: [http://www-947.ibm.com/support/entry/portal/Overview/Software/Other\\_Software/IBM\\_Support\\_Assistant](http://www-947.ibm.com/support/entry/portal/Overview/Software/Other_Software/IBM_Support_Assistant). Next, use IBM Support Assistant to locate and install any product updates. You can also choose to install plug-ins available for other IBM software in your environment. More information and the latest version of the IBM Support Assistant are available from the IBM Support Assistant Web page at: <http://www.ibm.com/software/support/isa/>.



---

## Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, is the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing  
IBM Corporation  
500 Columbus Avenue  
Thornwood, New York 10594 USA

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation  
Mail Station P300  
522 South Road  
Poughkeepsie, NY 12601-5400  
USA  
Attention: Information Requests

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.



---

## Trademarks

IBM, the IBM logo, and [ibm.com](http://ibm.com) are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at [www.ibm.com/legal/copytrade.shtml](http://www.ibm.com/legal/copytrade.shtml).

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

Linux is a trademark of Linus Torvalds in the U.S., other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.





---

# Index

## A

- administration
  - troubleshooting 480
- AP 105
- APIs
  - ClientLoader 370
  - DataGrid 248
  - DynamicIndexCallBack 145
  - EntityAgentMixin 248
  - EntityManager 164, 173
  - EntityTransaction 190
  - Index 142
  - JavaMap 160
  - ObjectMap 160
  - statistics 380
  - system 279
- application development
  - overview 125
  - planning 116
- architecture
  - topologies 83
- authorization 446
- availability
  - replication
    - client side 325
- availability partition (AP) 105

## B

- back-end 333
- backing maps
  - lock strategy 220
- batchUpdate method 342
- benefits
  - write-behind caching 95, 330
- best practices
  - tuning evictors 404
- byte array maps
  - performance improvement 400

## C

- cache
  - distributed 88
  - embedded 87
  - local 84
- cache integration
  - troubleshooting 478
- caching
  - configuring loader support 329
- class loaders
  - planning for 122
- classpaths
  - planning for 122
- client connections
  - administering
    - using JCA 71
- clients
  - programmatic configuration 253
  - troubleshooting 477

- coherent cache 90
- complete cache 92
- connecting
  - to a distributed data grid 130
- connection factories
  - configuring 63
  - creating resource references 65
- CopyMode
  - best practices 395
- create ObjectGrid 135

## D

- data access
  - indexes 142
  - ObjectGrid shard 141
  - overview 215
  - partitions 215
  - queries 215
  - REST data service 255
  - sessions 146
  - stored data 215
  - transactions 215
  - with applications 130
- database
  - data preloading 98
  - data preparation 98
  - database synchronization
    - techniques 100
  - read-through cache 93
  - side cache 92
  - sparse and complete cache 92
  - synchronization 100
  - write-behind cache 95, 330
  - write-through cache 93
- DataGrid agent
  - overview 248
- DataGrid API
  - example 249
  - overview 248
  - partitioning with 248
- deadlock
  - troubleshooting 483
- deadlocks
  - scenarios for 234
- development environment 125
- distributed cache 88
- distributing changes
  - using Java message service 223
- dynamic maps
  - maps 157

## E

- Eclipse Equinox
  - environment setup 37
- embedded cache 87
- entities
  - relationships 122, 164

- entity
  - life cycles of 177
  - listener 182
  - schema 166
- entity manager 9, 10
  - creating an entity class 9
  - entity relationship 10
  - fetch plan 183
  - querying 17
  - tutorial 9, 10
  - updating entries 15, 17
  - using an index to update and remove entries 16
- entity managerEntityManager
  - creating an order entity schema 12
- entity maps
  - creating 342
- entity schema
  - entity 166
- EntityManager API
  - distributed 173
  - fetch plan 183
  - for caching objects 164
  - performance 421
  - simple queries for 203
- EntityTransaction interface 190
- event listeners 296
- event-based validation 101
- evictors
  - configuring
    - with a stand-alone server 125
    - with Apache Tomcat 127
    - with WebSphere Application Server 129
  - map update 132
  - exception handling
    - collision exception 246
    - implementation with locking 238
- exclusive lock 234
- external transaction manager 358

## F

- failed updates 333
- FetchPlan 183
- FIFO queues
  - maps 161
- fixes
  - getting 464

## G

- get method
  - loaders
    - entity maps and tuples 342
- get ObjectGrid instance 139
- getting started
  - overview 73
  - with development 81
- grid authorization 454

## H

- heaps 404
- Hibernate
  - preload data
  - example 375

## I

- IBM Support Assistant 488
- in-line cache 92
- indexes
  - configuration 305
  - data quality 103
  - DynamicIndexCallBack 145
  - HashIndex 305
  - performance 103
- indexing
  - composite index 311
  - hash index 311
- instrumentation agent 422
- isolation
  - for transactions 244
  - pessimistic locking 244
  - repeatable read 244

## J

- Java Persistence API (JPA)
  - client-based loader
    - development 367
    - development with DataGrid
      - agent 371
      - example 370
      - example for custom 371
  - JPAEntityLoader plug-in
    - introduction 340
  - preload utility
    - example 369
    - overview 368
  - reload
    - example 370
  - time-based data updater
    - overview 378
  - time-based updater
    - starting 375
  - using with eXtreme Scale
    - overview 365
- JavaMap interface 160
- JCA
  - administering
    - client connections 71
- JPA cache plug-in
  - troubleshooting 479

## L

- listeners
  - callback methods 179
  - for backing map objects 296
  - introduction 296
  - MapEventListener plug-in 297
  - ObjectGridEventListener 298
  - ObjectGridEventListener plug-in 298
  - plug-ins 296
- load balancing 325

- loader
  - replica preload 346
- loaders
  - database 97
  - Java Persistence API (JPA)
    - overview 365
  - JPA programming considerations 338
  - overview 313
  - preloading 315
  - troubleshooting 481
  - update failures 333
  - update tracking 132
  - using with entity maps and tuples 342
  - writing 321
- local cache
  - peer replication 85
- local security
  - programming 454
- locking
  - configuration with XML 239
  - configuring programmatically 239
  - no 239
  - optimistic 221, 239
  - performance 405
  - pessimistic 221, 239
  - strategies for 221
- locks
  - compatibility 234
  - life cycle 234
  - time out 234
  - timeout 241
  - usage overview 234
- log analysis
  - custom 475
  - overview 473
  - running 473
  - troubleshooting 476
- log data 472
- log element 132
- log sequence 132
- LogElement 132
- logs 468
- LogSequence 132

## M

- map entry locks
  - indexes 242
  - query 242
- map pre-loading 325
- multi-master data grid replication
  - planning 105
- multi-master replication
  - configuration planning 109
  - custom arbiters 283
  - design planning 111
  - planning 105
  - planning for loaders 109
- multimaster replication
  - topologies 105
- multiple data center configurations 481

## O

- object query
  - index 3
  - map schema 1
  - primary key 1
  - tutorial 1, 3, 5
- ObjectGridManager interface
  - controlling life cycle with 140
  - createObjectGrid methods 135
  - getObjectGrid methods 139
  - removeObjectGrid methods 139
  - using to interact with an ObjectGrid 135
- ObjectMap API
  - caching objects with 152
  - overview 154
- ObjectTransformer
  - best practices for 403, 408
- OSGi
  - administering applications 41
  - administering servers 41
  - administering services 54
  - building dynamic plug-ins 43, 362
  - building plug-ins 42
  - configuring plug-ins 50
  - configuring servers 57
  - developing plug-ins 35
  - Eclipse Equinox environment 37
  - installing bundles 38
  - installing plug-ins 48
  - overview 35
  - programming 361
  - running containers 40
    - with non-dynamic plug-ins 50
  - running plug-ins 35
  - starting servers 51
  - tutorials
    - configuration files 21
    - configuring containers 26
    - configuring servers 25
    - finding service rankings 33
    - installing bundles 24
    - installing protocol buffers 27
    - overview 18
    - preparing to install bundles 19
    - querying bundles 31
    - querying service rankings 31
    - running bundles 18
    - running clients 29
    - sample bundles 20
    - setting up Eclipse to run clients 29
    - starting bundles 24, 28
    - starting clients 30
    - updating service rankings 34
    - upgrading bundles 31
- OSGi container
  - Apache Aries Blueprint
    - configuration 46

## P

- partitions
  - transactions 225
  - using non-keys to find objects in 153

- performance
  - best practices
    - locking 405
  - database 325
  - EntityManager 421
  - evictors 404
  - locking 405
  - tuning
    - application development 395
- Performance Monitoring Infrastructure (PMI) 380
- performance tuning 391
- planning 83
  - application development 116
  - cache keys 124
  - class loaders 122
  - classpaths 122
- plug-ins
  - BackingMapLifecycleListener 300
  - BackingMapPlugin 281
  - HashIndex 305, 306
  - index 308
  - introduction 117
  - lifecycle management 279
  - multi-master replication 283
  - ObjectGridLifecycleListener 302
  - ObjectGridPlugin 280
  - ObjectTransformer 292
  - OptimisticCallback 284
  - plug-in slots 356
  - TransactionCallback 351
  - WebSphereTransactionCallback 360
- Programming eXtreme Scale 116

## Q

- query
  - Backus Naur 212
  - BNF 212
  - clauses 204
  - client failure 186
  - composite index 311
  - entity 200
  - example 203
  - functions 204
  - get plan 411
  - index 203, 413
  - key collision 186
  - methods 190
  - object map 196
  - ObjectQuery schema 198
  - optimization with indexes 413
  - pagination 203
  - parameters 203
  - predicates 204
  - query plan 411
  - queue 186
  - schema 198
  - search elements 190
  - tuning 409
  - valid attributes 198
- queues 404

## R

- replication
  - enabling client-side 254
  - preload 346
- request
  - per-container 150
  - routing 150
  - Session 150
- resource adapters
  - installing 61
- REST data service
  - delete requests 277
  - insert requests 270
  - non-entity retrieval 265
  - operations 255
  - optimistic concurrency 257
  - overview 119
  - planning 119
  - request protocols 258
  - retrieve request 258
  - update requests 273
- scenarios 35
- security
  - client authentication 429
  - J2C client connections 65, 426
  - local 454
  - overview 425
  - plug-ins 454
  - programming 427
- security profile 425
- securityAPI 427
- serialization
  - locking 407
  - performance 407
- serializer
  - APIs 290
  - developing 290
  - overview 288
  - plug-ins 288
- SessionHandle
  - routing 149
- sessions
  - access data 146
  - collision 246
  - transaction 246
- shared lock 234
- side cache
  - database integration 92
- sizing 392
- sparse cache 92
- Spring
  - clients 388
  - container servers 386
  - extension beans 121, 380, 383
  - framework 121, 380
  - namespace 383
  - namespace support 121, 380
  - native transactions 121, 380
  - packaging 121, 380
  - shard scope 121, 380
  - transactions 381
  - webflow 121, 380

- starting
  - container servers
    - Spring 386
- statistics API 380
- Support 488
- system API 279

## T

- time zones
  - inserting data 124, 195
  - querying data in 194
- topologies
  - plan 83
- trace
  - options for configuring 470
  - troubleshooting 469
- trace data 472
- transactions
  - callback 315
  - connecting applications 58
  - copyMode 219
  - cross-grid 225
  - data access 215
  - developing client components 67, 230
  - external managers 358
  - ID 315
  - overview 218
  - processing 59
  - processing overview 214, 355
  - programming for 214
  - single-partition 225
  - Spring 381
- troubleshoot
  - cache integration 478
  - HTTP session 478
- troubleshooting 461
  - administration 480
  - identifying problems, techniques for 461
  - trace 469
- troubleshooting and support
  - getting fixes 464
    - Fix Central 464
  - IBM Support 465
  - overview 461
  - search known problems 463
  - subscribing to IBM Support 467
  - troubleshooting techniques 461
- tuple objects
  - creating 342
- tutorials 1
  - configuration files 21
  - configuring eXtreme Scale
    - containers 26
  - configuring eXtreme Scale servers 25
  - creating entity classes 9
  - finding service rankings 33
  - forming entity manager
    - relationships 10
  - installing bundles 24
  - installing eXtreme Scale bundles 24
  - installing Google Protocol Buffers 27
  - object query 1, 3, 5
  - ordering entity schemas 12

## tutorials (continued)

### OSGi

- configuration files 21
  - configuring containers 26
  - configuring servers 25
  - finding service rankings 33
  - installing bundles 24
  - installing protocol buffers 27
  - overview 18
  - preparing to install bundles 19
  - querying bundles 31
  - querying service rankings 31
  - running clients 29
  - sample bundles 20
  - setting up Eclipse to run clients 29
  - starting bundles 18, 24, 28
  - starting clients 30
  - updating service rankings 34
  - upgrading bundles 31
- overview
- starting servers and containers 18
- preparing to install eXtreme Scale bundles 19
- querying bundles 31
  - querying local data grids 1
  - querying service rankings 31
- running sample clients in OSGi 29
- sample OSGi bundles 20
  - setting up Eclipse for OSGi 29
  - start OSGi bundles 28
  - starting bundles 18
  - starting client applications in the OSGi framework 30
- storing information in entities 9
- updating and removing entities using queries 17
  - updating and removing entries using an index 16
  - updating bundles 31
  - updating entries 15
  - updating service rankings 34

## U

- upgradeable lock 234

## W

### write-behind

- configuring loader support 329
- database integration 95, 330
- example 335
- failed updates 333

## X

### xscmd

- security profile 425
- xsloganalyzer 473, 475





Printed in USA