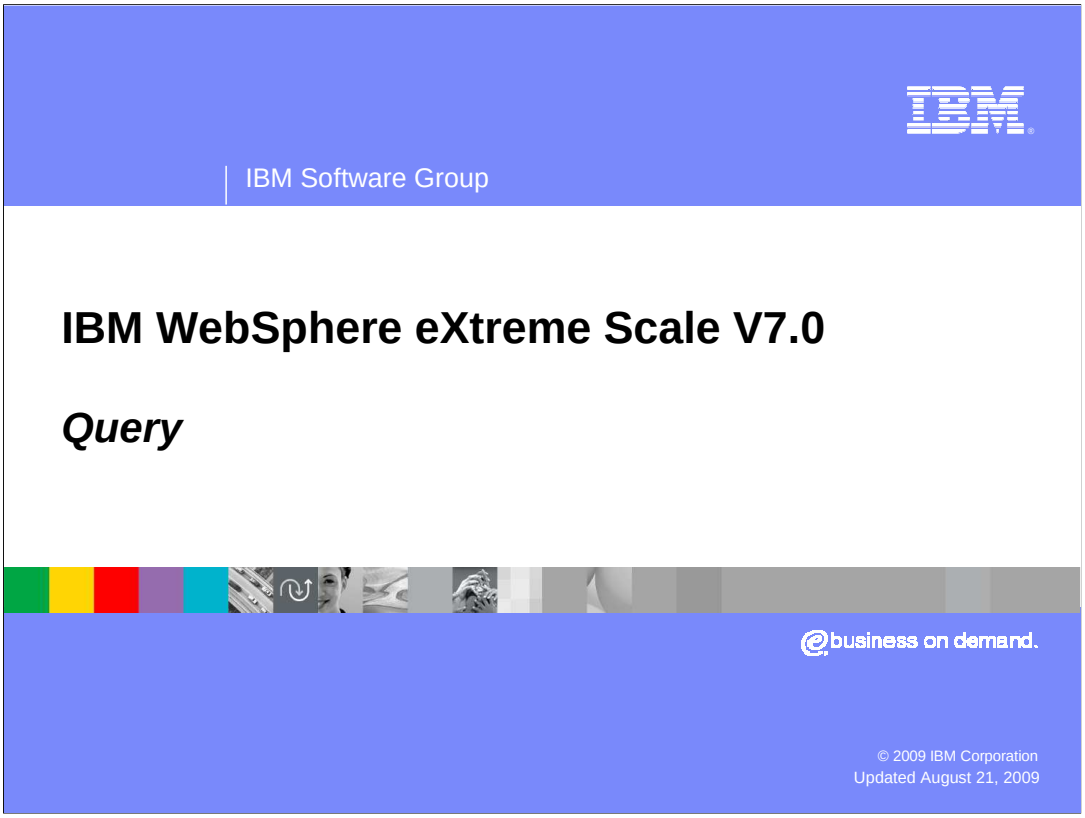


WebSphere® eXtreme Scale provides the ability to query data in the cache using traditional SQL queries. This presentation will provide an overview of the product's query capabilities.

Flexible query engine

- Single query engine
  - Traditional ObjectMap POJO objects
  - Entity objects using the EntityManager APIs
- Similar to JPA's Query API
  - Java™ Persistence Architecture Query Language
  - Tailored for ObjectMaps
- SELECT only

The eXtreme Scale core programming model allows you to retrieve individual Objects from the grid by primary key. It also provides mechanisms to retrieve a queue of primary keys for an ObjectMap which allows you to access all data in the map.

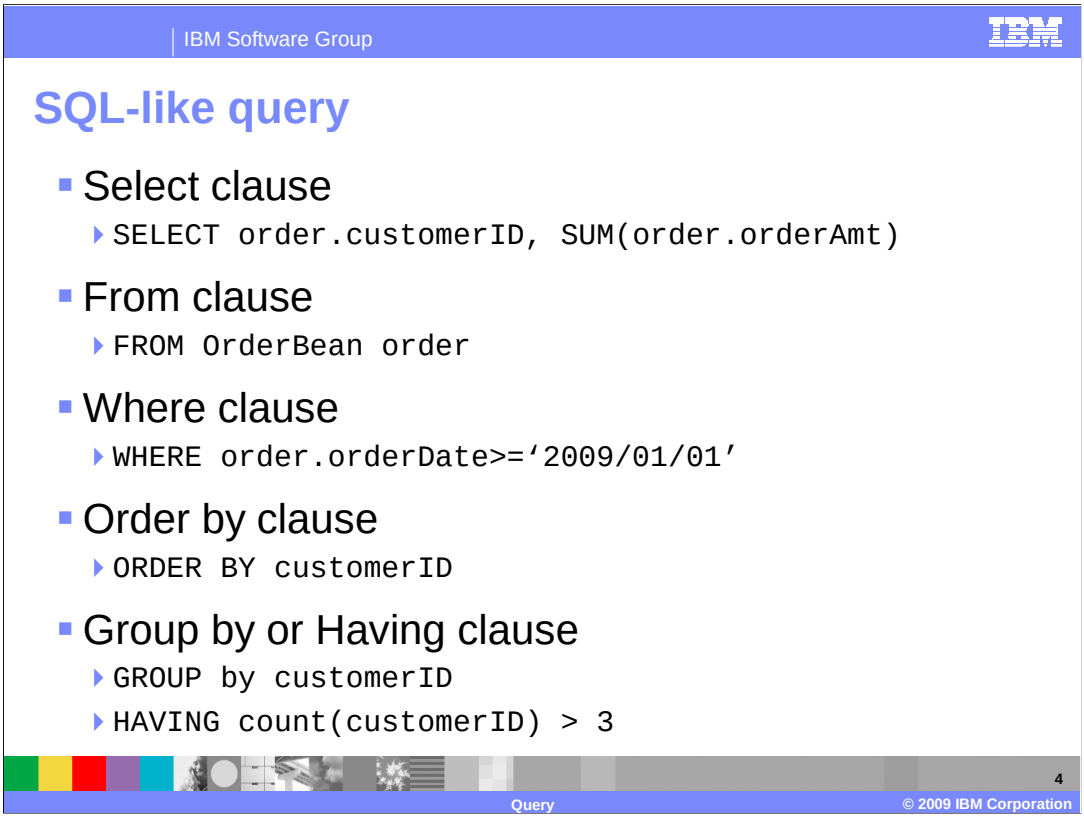
As more and more data is stored in the grid, it becomes desirable to use more sophisticated queries to retrieve a subset of the data in the grid. WebSphere eXtreme Scale V7.0 provides a flexible query engine for retrieving Entities or Java objects from the grid. The ObjectGrid query language uses semantics similar to the Java Persistence Architecture Query Language, but is tailored to more closely match the ObjectMap architecture.

The query engine allows SELECT type queries over both entities and Object-based schemas. The same query language is used to query both types of objects, but they are accessed using different API methods.

Query engine capabilities

- Aggregate functions
- Sorting and grouping
- Joins
- Conditional expressions with subqueries
- Named and positional parameters
- Path expression syntax for object navigation
- Indexing

The ObjectGrid query language provides advanced query capabilities such as data aggregation, sorting and grouping, and sub-expressions. These are described in detail in the WebSphere eXtreme Scale Programming Guide.



With the ObjectGrid query engine, you can use a single query language for searching the ObjectGrid cache. This query language can query Java objects that are stored in ObjectMaps, and it can query Entity objects.

An ObjectGrid query is a string that contains these elements:

A **SELECT clause** that specifies the objects or values to return. The query can return an entire Object or Entity, a list of attribute values, or results of aggregation functions such as SUM().

A **FROM clause** that 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.

An optional **WHERE clause** that specifies which objects to include in the result set. If the WHERE clause is omitted then all objects in the map are included.

An optional **ORDER BY clause** that specifies the ordering of the result collection.

The **GROUP BY clause** is used in conjunction with aggregate functions to group the result set by one or more columns. For example, you might want to find the total order value for each customer ID from an order map. To do this you select the customer ID and *sum* of order amounts, and group by customer ID. If you SELECT non-aggregate attributes along with aggregate functions you must include a GROUP BY clause. If you only include aggregate functions in the SELECT then the entire result set matching the WHERE is treated as a group.

You can further filter these groups by including a **HAVING clause** that tests group properties before invoking selected aggregate functions or grouping members. This filtering is similar to how the WHERE clause filters objects from the FROM clause. However, while the WHERE selects input rows before groups and aggregates are computed, the HAVING selects groups of rows *after* groups and aggregates are computed. Unlike the WHERE clause, the HAVING clause can include aggregate functions. In the previous example, you can limit results to active customers by requesting total sales only for customers that have placed more than three orders.

Aggregation functions

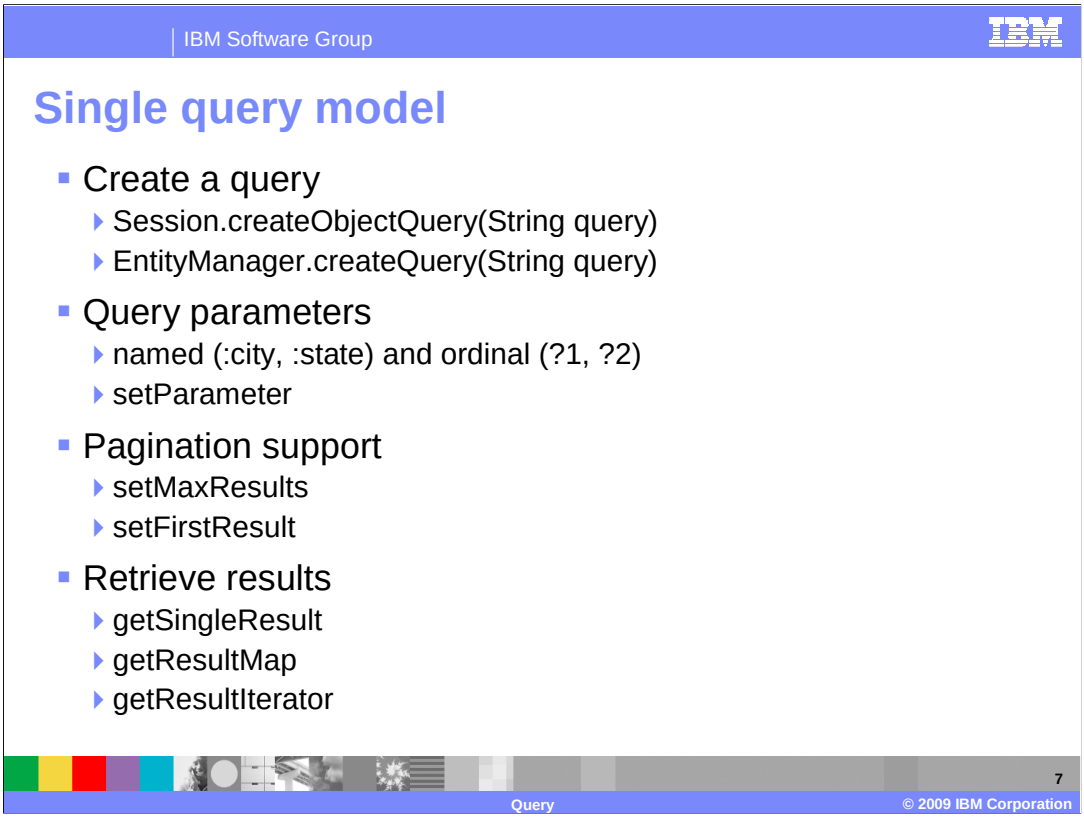
- AVG
- COUNT
- MIN
- MAX
- SUM

The aggregation functions include average of an attribute value for a group of objects, count of objects in the group, minimum value, maximum value, and sum of a given attribute value for a group of objects.

Section

Programming model

This section will provide an overview of the eXtreme Scale query programming model.



WebSphere eXtreme Scale provides similar query interfaces for both ObjectMap and Entities.

You create an ObjectQuery object through the session and an entity query through the EntityManager. The input to both is an SQL SELECT statement.

The select statement can contain replaceable parameters. These allow you to create a query once and run it multiple times with different values for the parameters. Replaceable parameters can be either positional or named. Positional input parameters are defined by a question mark followed by a positive number. They are numbered starting at one and correspond to the arguments of the query. Named input parameters are defined by a colon followed by a parameter name. Before executing the query you must provide actual values for these parameters using the Query object's setParameter method.

The query object also provides pagination support by allowing you to specify which entry in the result set to return first and the maximum number of results to return for this iteration.

You can retrieve a single object, an ObjectMap containing just the results of the query, or an Iterator over the result set. A call to getSingleResult *must* return *exactly* one result. If there are no results or more than one in the result set it will produce an exception.

Programming model – ObjectQuery example

```
// A transaction is required when the results are in an iterator or ObjectMap.
// The results are transaction-scoped.
session.begin();

// Use Join syntax to join resolve a relationship.
ObjectQuery q = session.createObjectQuery(
    "SELECT e.name, e.salary, d
         FROM EmpMap e JOIN e.deptid as d
         WHERE e.salary > :eSalary");

System.out.println("Rich Employees:");
q.setParameter("eSalary", new Double(20000));
Iterator iEmployees = q.getResultIterator();

while(iEmployees.hasNext()) {
    // Results are in an object array
    Object[] emp = (Object[]) iEmployees.next();
    String name = (String)emp[0];
    double salary = ((Double)emp[1]).doubleValue();
    DeptBean dept = (DeptBean)emp[2];
    System.out.println(name + ", " + salary + ", " + dept.name);
}

session.commit();
```


This slide shows an example object query. The code creates a query object, sets replacement query parameters, and runs the query. The code then iterates over the result set. Note that the query results are only valid within the current transaction. EntityQuery is similar; the primary difference being that the query is created through the EntityManager rather than the session.

Programming model – ObjectQuery example

```
// A transaction is required when the results are in an iterator or ObjectMap.
// The results are transaction-scoped.
session.begin();

// Use Join syntax to join resolve a relationship.
ObjectQuery q = session.createObjectQuery(
    "SELECT emp
         FROM EmpMap emp
         WHERE emp.salary > :eSalary");

System.out.println("Rich Employees:");
q.setParameter("eSalary", new Double(20000));
Iterator iEmployees = q.getResultIterator();

while(iEmployees.hasNext()) {
    // Results are in an object array
    Employee emp = (Employee) iEmployees.next();
    String name = emp.getName();
    double salary = emp.getSalary().doubleValue();
    DeptBean dept = emp.getDepartment;
    System.out.println(name + ", " + salary + ", " + dept.name);
}

session.commit();
```


This is similar to the previous example except the query retrieves an entire employee object instead of just selected fields.

Query results

- Single value query
  - Attribute Object
  - POJO
  - Entity
- Multiple value query
  - Array of Attribute Objects
  - Entity

An individual query can return a single object, an ObjectMap containing just the results of the query, or an Iterator over the result set. Each returned value is either an Object for a single valued query, or an array of Objects for a multiple valued query. The values in an Object array result are stored in the order specified in the SELECT clause; that is, the zeroth element in the array corresponds to the first value specified in the SELECT. If you request a result map, the map key is the result number starting at one and the map value is an Object array.

Entity query example

```
EntityTransaction tran = em.getTransaction();
tran.begin();

Query q = em.createQuery(
    "SELECT e.name, e.salary, d FROM Employee e, Department d WHERE e.salary >
  :eSalary");

System.out.println("Rich Employees:");
q.setParameter("eSalary", new Double(20000));
Iterator iEmployees = q.getResultIterator();

while(iEmployees.hasNext()) {
    // Results are in an object array
    Object[] emp = (Object[]) iEmployees.next();

    String name = (String)emp[0];
    double salary = ((Double)emp[1]).doubleValue();
    DeptBean dept = (DeptBean)emp[2];
    System.out.println(name+", "+salary+", " +dept.name);
}
tran.commit();
```


This slide shows an example Entity query. Note that it is similar to the Object query. The code creates the transaction and query objects through the EntityManager rather than the session. The interface for setting replacement query parameters, running the query, and iterating over the result set is the same.

Entity query example

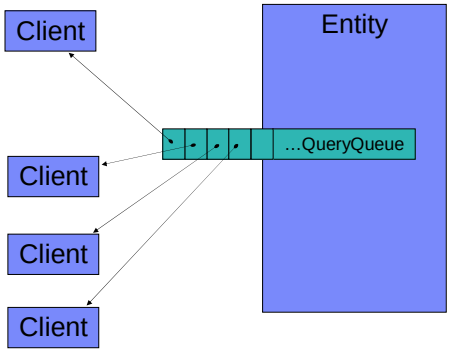
```
EntityTransaction tran = em.getTransaction();
tran.begin();

Query q = em.createQuery(
    "SELECT e.name, e.salary, d FROM Employee e, Department d WHERE e.salary > :eSalary");

System.out.println("Rich Employees:");
q.setParameter("eSalary", new Double(20000));
Iterator iEmployees = q.getResultIterator(QueryResult.class);

while(iEmployees.hasNext()) {
    // Results are injected into a QueryResult temporary entity
    QueryResult emp = (QueryResult) iEmployees.next();
    System.out.println(emp.name + ", " + emp.salary + ", " + emp.dept.getName());
}
tran.commit();

 …

// Create an entity that is compatible with the queried fields.
// Since query retrieves name, salary and department, the result entity only needs these.
@Entity
Public static class QueryResult {
    @Basic String name;
    @Basic double salary;
    @OneToOne Deptartment dept;
}
```


An Entity Query can also project the results of a multiple valued query into a separate, temporary, Entity designed to hold the query results. The temporary entity is valid for the life of the transaction.

This slide shows an example Entity query which projects the results into a different class. The QueryResult object defined in this example contains attributes that correspond to the values in the SELECT clause of the query. The entity query getResultIterator method can take this entity class as a parameter, instructing the EntityManager to convert the query results to Entities rather than Object arrays.

Entity QueryQueue

- Useful for retrieving select entities in an iterative manner
- Like ObjectMap.getNextKey()
  - But uses query to set match criteria
- Interface similar to EntityManager Query
  - `EntityManager.createQueryQueue`
  - `setParameter(…)`
  - `getNextEntity`
  - `getNextEntities(<num>,…)`
  - `setPartition`



Entity QueryQueues operate like the ObjectMap getNextKey method, which retrieves keys from a map in the order they were inserted. Where the ObjectMap getNextKey method returns all entries in an ObjectMap, a QueryQueue uses the query to filter returned Entities. Entities from the query result are retrieved from a queue rather than through an Iterator. Unlike the Iterator returned by a normal Entity query, a QueryQueue is shared by multiple transactions/clients.

You can retrieve entries from a QueryQueue one at a time, or in batches. When an entry is retrieved from a QueryQueue it is locked for update, so no other client can retrieve that Entity.

When a query queue is created and the getNextEntity is called, the query associated with the queue is run and the results are placed into the queue. Every time getNextEntity is called, an entity is taken off the queue until the queue is empty. When getNextEntity is called against the empty queue, the entity query will automatically re-run and re-populate the queue. Since the QueryQueue automatically recycles when it is empty, you must be aware of the possibility of an infinite loop over the data. To avoid this, modify returned entities so they no longer match the query.

For example, a customer support application includes a Call entity. A support agent client gets the next unassigned Call for the agent's specialty and assigns it to the agent. After the Call is assigned to an agent it will no longer match the query, so it will no longer show up in the QueryQueue.

When operating in a distributed environment you can optionally specify which partition the query must be routed to. If the partition ID is not specified, the QueryQueue is routed to all partitions, starting from a random partition.

A query queue is uniquely identified within an eXtreme Scale grid by the query string and parameters. The grid will contain only one result set for each unique query queue. If 1000 grid clients all create QueryQueue instances using the same string, the grid will create one "global" queue instance within the grid rather than evaluating the query separately for each client. The client QueryQueue instances will attach to this "global" queue and pull entities from it.

See the EntityManager API documentation for additional information.

Query and partitions

- Query.setPartitionId()
  - Requires multiple transactions
  - Loop over all partitions
- Use MapGridAgent or ReduceGridAgent to spray queries over all partitions simultaneously
  - Agent will serialize results (including entities) to client
    - Entities must support serialization and are therefore detached
  - Limited to joins over data collocated in single partition
  - No aggregation over partitions
    - For example, count() will retrieve a count per partition and the total result has to be computed by applying the function to the results from the agent

Normally, queries only search within a single partition. If you have multiple partitions your code must specify which partition to run the query against using the setPartitionId method. If you need to gather data from multiple partitions you must loop through the available partitions, running the query against each partition in turn.

As an alternative, you can use the data grid APIs to run the same query in parallel on all partitions or a subset of partitions in order to query large amounts of data efficiently.

Section

Configuration

This section will describe how to configure an eXtreme Scale grid for Query.

Query schema

- Type/class of object stored in the ObjectMap
- Primary key attribute name
- The method for which each query should access the data attributes in the Objects
  - Directly or getter/setter
- Relationships between ObjectMaps

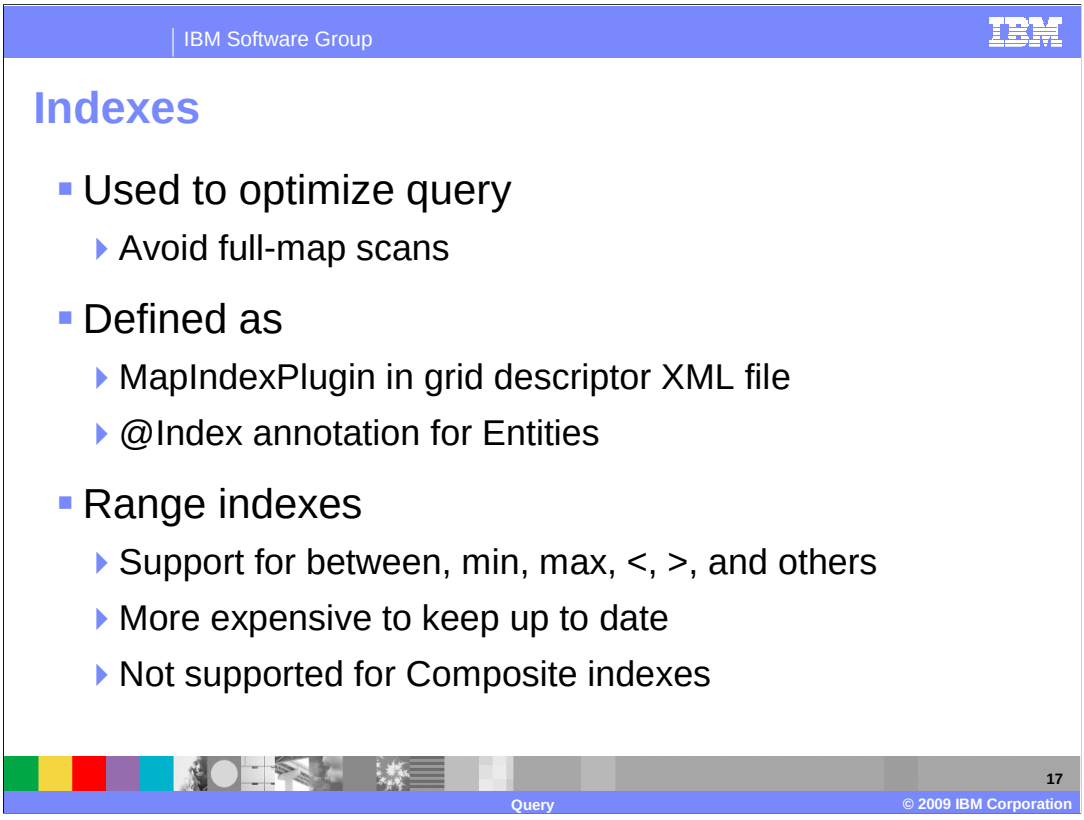
Before you can query an object you must define a query schema in the grid descriptor XML file. The query schema tells the ObjectGrid what type of data you plan on inserting into the maps and how to relate them. The query schema includes a map schema entry for each queryable map and an entry for each relationship between maps. The map schema specifies the name and the type of object that is stored in the queryable BackingMap. It also identifies which field within the stored object represents the primary key attribute. The primary key must also be stored in the key portion of the BackingMap.

The query engine uses Java introspection to access persistent data in the Objects stored in the map. You can specify whether the query engine access these fields directly or through a getter method.

Specify relationships between objects in the maps if you need to follow the relationships in a query. For instance, if you specify that an Employee is a member of a department, your query can include a reference to employee.department.number. Relationships are also used implicitly to optimize joins.

Each relationship must be explicitly defined in the schema configuration. The cardinality of the relationship is automatically determined by the type of the attribute. For instance, if the attribute implements the java.util.Collection interface, then the relationship is either a one-to-many or many-to-many relationship.

Entities accessed through the EntityManager do not require a separate query schema as the required information is explicitly defined in the Entity metadata.



ObjectGrid uses backing map plug-ins to manage indexes on maps. A backing map can have multiple indexes. The query engine automatically incorporates any indexes that are defined on a schema map element. eXtreme Scale includes a default index implementation: the HashIndex plug-in . You can provide your own index implementation by implementing the MapIndexPlugin interface.

In addition to indexes statically defined in the grid descriptor XML file, you can dynamically create indexes after the backing map has been initialized using the createDynamicIndex method.

Entities can use the @Index annotation to implicitly define indexes in the map schema.

A RangeIndex is a specialization of the MapIndex that adds flexibility in the type comparisons supported. RangeIndexes can greatly improve the performance of some queries, but they are relatively expensive to maintain.

In general, the more indexes that are available, the faster the query will run. However, each index consumes system resources. In addition to the memory required for indexes, additional processing is required for index maintenance as grid entries are added, removed, and updated. This can have an adverse affect on performance as the number of the indexes on the BackingMap increases.

Problem determination

- The following trace specifications are available
  - Query Engine: QueryEngine (or com.ibm.queryengine.*)
    - Object Query: com.ibm.ws.objectgrid.query.*
    - Entity Query: com.ibm.ws.objectgrid.em.query.*
  - QueryEnginePlan
    - Query plan diagnostics
    - Must be set on server in distributed environment
- Query.getPlan()
  - In distributed environment use agent to run on server
- Informational messages, warnings and errors can be logged in the ObjectGrid logs

This slide lists trace strings you can set to get more details on query engine operations. See the extreme Scale information center for more details on these traces.

You can also call the query object's getPlan method from your code. getPlan returns a String which describes the how the query engine will run the query. This string can be displayed to standard output or written to a log. In a distributed environment, the getPlan method does not run against the server and will not reflect any defined indexes. To view the plan, use a data grid agent to view the plan on the server.

Query plan

- Sample query

```
SELECT emp from employees AS emp WHERE
  emp.salary<=?1
```

- No index

```
for q2 in employees ObjectMap using INDEX SCAN
 filter ( q2.getSalary() <=  ?1 )
returning new Tuple( q2  )
```

- With index

```
for q2 in employees ObjectMap
  using RANGE INDEX on SalaryIdx with range(, ?1]
 filter ( q2.getSalary() <=  ?1 )
returning new Tuple( q2  )
```


This slide shows two plans for a simple query as returned by the getPlan method. The first query does not use an index. This plan indicates it will perform a full scan. The second sample is the same query after creating a range index named SalaryIDX on the salary attribute. The query plan will also show how the query engine uses loops for JOINs and complex WHERE clauses.

For complex queries the query engine might not always choose the best index for a query. For those cases, the Query interface provides a setHint method that allows you to specify the preferred index. The WebSphere eXtreme Scale Programming Guide contains more examples of query plans along with advice on how queries can be re-factored for better performance.

Summary

- Single query engine
  - POJOs
  - Entities
- Similar to JPA's Query API
  - Java Persistence Architecture Query Language
  - Tailored for ObjectMaps
- SELECT only

WebSphere eXtreme Scale V7.0 provides a flexible query engine for retrieving Java objects using the ObjectQuery API and entities using the EntityManager API. The ObjectGrid query language uses SQL semantics similar to the Java Persistence Architecture Query Language. The query engine allows SELECT type queries over both entity or Object-based schema using a common query language.

Feedback

Your feedback is valuable

You can help improve the quality of IBM Education Assistant content to better meet your needs by providing feedback.

- Did you find this module useful?
- Did it help you solve a problem or answer a question?
- Do you have suggestions for improvements?

Click to send e-mail feedback:

mailto:iea@us.ibm.com?subject=Feedback_about_WXS70_Query.ppt

This module is also available in PDF format at: ../WXS70_Query.pdf

You can help improve the quality of IBM Education Assistant content by providing feedback.

Trademarks, copyrights, and disclaimers

IBM, the IBM logo, ibm.com, and the following terms are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both:

WebSphere

If these and other IBM trademarked terms are marked on their first occurrence in this information with a trademark symbol (® or ™), these symbols indicate U.S. registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of other IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

Java, and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.

Note to U.S. Government Users - Documentation related to restricted rights-Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract and IBM Corp.