IBM Information >>> On Demand 2007

Future Technology Directions for Database Access from Java Applications

Curt Cotner, IBM Fellow, cotner@us.ibm.com

Session 1297A

Act.Right.Now.

IBM INFORMATION ON DEMAND 2007
October 14 – 19, 2007
Mandalay Bay
Las Vegas, Nevada

IBM Data Servers

Reduce cost of deployment and management of data

- *Innovation to reduce the cost of infrastructure*
- *Innovation to manage the lifecycle of data -
  from modeling and design through change management and sunsetting*

Enable rapid use of data throughout the enterprise

- *Innovation that accelerates SOA and XML initiatives*
- *Innovation that leverages Web 2.0 and situational applications*

Agenda

- Overview of Java data access challenges
- J-LinQ overview
- J-LinQ programming details
- Management and Monitoring

Java access to relational – no size fits all

I don't want to have any SQL in my program.

I need to run on J2SE.

I want a full persistence layer with transaction management.

I just need a light-weight interface that makes SQL coding easy.

I need a solution that locks in database access path for reliable performance.

I need OR mapping between relational and my object model.

I want business objects returned to my program – not individual columns.

I need an interface that allows efficient SQL access and detailed monitoring for problem diagnosis.

Java Data Access – many forms

POJO with inline SQL
JDBC, SQLJ

Spectrum of choices

POJO iBatis Hibernate EJB 3

EJB application

EJB query lang

OR mapping layer

Persistence Layer

config file for named queries

Pro's:
- Simplicity
- Easy to control SQL
- Good performance
- Good monitoring (SQLJ)

Con's:
- Not tied to object model
- More work for app pgmr

DB2

Pro's:
- Less work for app pgmr
- Access via OO business objects

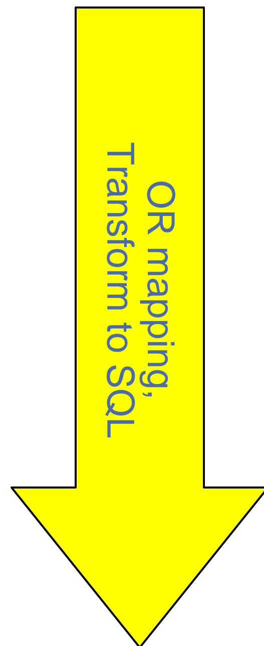
Con's:
- Complexity
- Less control over SQL issued
- Performance can suffer
- Very difficult to monitor or diagnose problems

What performance/diagnosis challenges?

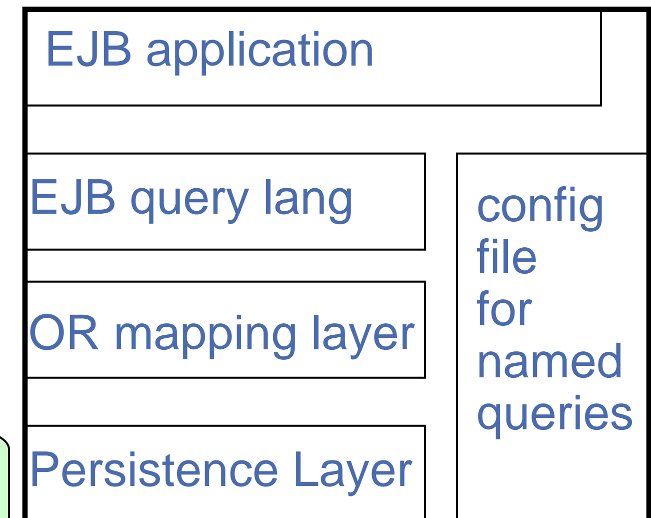
EJB Query Language:

```
SELECT object(e) FROM Employee e
  WHERE e.dept=?1 AND e.salary>=?2
```

Query language is a subset of SQL. Doesn't have all the SQL features you want.



App query syntax is different from SQL query. How do you track problem SQL queries back to the app that issued the original query???



In most cases, queries map to JDBC. No ability to lock in access path at program deployment. No ability to search catalog to see which queries are issued by a given program.

SQL issued to database:

```
SELECT * FROM PROD.EMP
  WHERE DEPT=? AND SALARY>?
```

Often, app query is intercepted by persistence layer, and the resulting SQL query looks nothing like the app query.
-- Resulting query might perform badly.
-- Changing app query might not result in a similar change in the SQL query…

JLINQ – Beyond Function

- Development of applications
  - Tools to assist SQL development in .java source file
  - Simple SQL APIs, easy to write to and extend
  - Multiple API "styles" to align with popular Java frameworks
- Query important data sources simply
  - Database, Cache, Collections, XML
- Problem Determination
  - When problems occur, find source quickly.
- Governance / Management
  - Track SQL back to individual apps, lock in access paths with static SQL packages, align with customer change control processes
- Provide high performance/scalability
  - Application: short path length, coding over metadata, optional code gen, JDBC and static SQL runtime optimizations
  - Database: static SQL, batching, pass app SQL directly to database

Java Data API Space

Complex

J2EE,
Managed Objects

EJB 2

JPA / EJB3

SQLJ

JDBC

Web 2.0,
Performance
XML, JSON

JLINQ, Spring, iBatis

Simple

J2EE Managed Objects Complex O/R	JPA With JLINQ
All other cases ▪Development speed ▪Performance ▪Web 2.0 / AJAX ▪XML / JSON ▪Dynamic & Static SQL	JLINQ API
Already using SQLJ	SQLJ

JLINQ API "Styles"

- Support several API styles to fit well into all of the popular Java programming models/frameworks
  - Inline style (familiar JDBC and SQLJ approach)
  - Method style (similar to JDBC 4 ease of use enhancements)
  - Named query style (similar to iBatis/JDO/Hibernate/JPA)

Java Persistence Technologies with JLINQ

JDBC
SQLJ

JPA API

JLINQ API

iBatis

Spring

JPA Runtime

JLINQ Runtime

High Speed API

JCC JDBC w/JLINQ

IBM
Database

JLINQ Metadata,
Manageability

Retrieve a single row from Database

JLINQ:

Automatically Optimizes for 1 row

```
addr = db.queryFirst("SELECT ADDRESS FROM EMP
            WHERE NAME=?name", String.class, name);
```

-or-

```
addr = getAddress(name);
```

XML file or Java annotation
SELECT ADDRESS FROM EMP
WHERE NAME=?1;

SQLJ:

```
#sql [con] { SELECT ADDRESS INTO :addr FROM EMP
            WHERE NAME=:name };
```

JDBC:

```
java.sql.PreparedStatement ps = con.prepareStatement(
      "SELECT ADDRESS FROM EMP WHERE NAME=?");
ps.setString(1, name);
java.sql.ResultSet names = ps.executeQuery();
names.next();
addr = names.getString(1);
names.close();
```

JLINQ Architecture

Popular Frameworks
Spring, iBatis, EJB 3, JDO, etc.

JLINQ
API
Styles

POJO apps

JDBC
SQLJ

Optimized
static
SQL
generated
Code

Dynamic
SQL

Data
Server

App
SQL
Meta
Data

Custom

Entities

Objects

Cache

XML

Atom

SQL

XQuery

Java editor w/ Integrated Query

Java compiler w/ Integrated Query

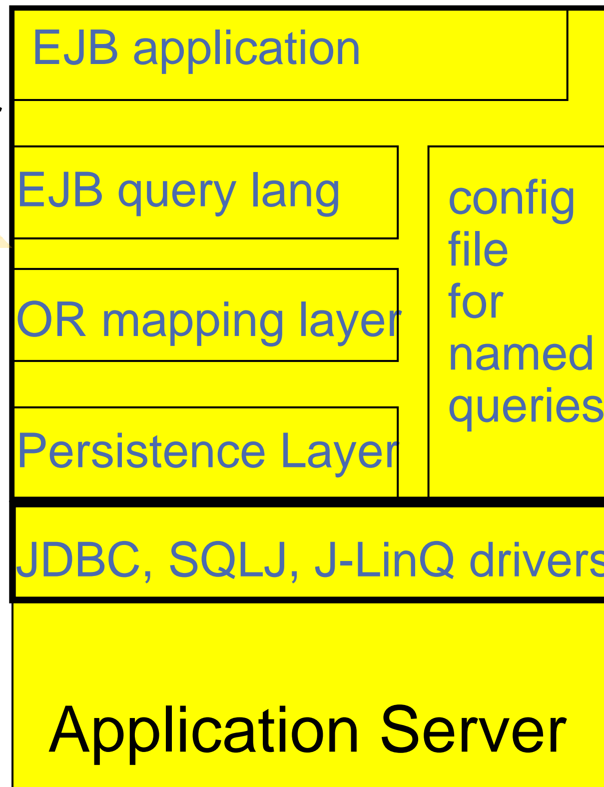
Problem Determination and Monitoring

Original query issued by app
Java class and line number
XML filename and line number
Last compile date/time for app
Full history of all SQL issued
by each application

Generated static SQL
-- SQL after O/R mapping and persistence
-- plan lockdown
-- package versioning

Set Client Information APIs
-- end user's ID
-- end user's IP address
-- application name
-- accounting chargeback data

J-LinQ



App server IP address
App server connection pool userid
JDBC driver package name

JDBC Static SQL Profiling
-- SQL after O/R mapping and persistence
-- Java stack trace for SQL call
-- plan lockdown
-- package versioning

Set Client Information APIs
-- end user's ID
-- end user's IP address
-- application name
-- accounting chargeback data

JDBC

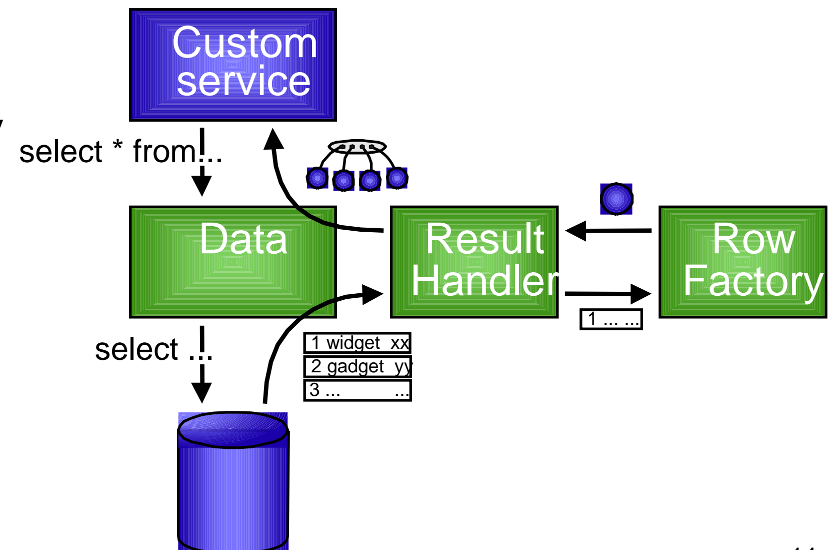
JLINQ Technology SQL Query API

- Simple, straightforward programming model for data access
  - A fairly thin layer on top of JDBC that simplifies the most common tasks
  - Supports DB2, IDS, Oracle, SQL Server, etc.   (any JDBC database)
  - Out-of-the-box support for storing/retrieving Beans and Maps to/from the database
- Extensible framework
  - Pluggable custom result processing patterns
    - Use Java to implement the mapping behavior instead of a "mapping language"
    - Instantiate result types other than Beans and Maps
  - Framework itself uses the same extension points to provide the out-of-the-box behavior
    - Library of the most common patterns
- Full expressiveness of SQL available
  - In practice, even simple applications do "sophisticated" SQL
- SQL inlined in data access methods
  - Everything that is needed to understand a data access method is in the method

JLINQ – Data API

- Data
  - API for accessing databases or in-memory collections
    - A pluggable "callback" mechanism for customizing the data access patterns
    - A set of convenience methods that wrap the default plugin use for most common patterns
  - Encapsulates connection caching, DB2 performance metrics and problem determination etc. functions

- ResultHandler – optional control point
  - Implements the result set iteration strategy
  - Returns sets of objects (e.g. List)
- RowFactory – optional control point
  - Implements the result object creation & hydration strategy
  - Returns single objects (e.g. Bean)



Data Queries

- Queries are inlined in the application code
  - Standard SQL with no limitations

```
Person person = data.queryFirst("SELECT * FROM person
          WHERE person.name=?", Person.class, person);
```

- Query parameters alternatives
  - Standard parameter markers (e.g. “?”)
  - Numbered parameter markers (e.g. “?1”)
  - Named parameters (e.g. “?my_var”)
    - Parameters passed in either as a map or as a Bean

```
Map parms = (new HashMap()).put("name","Brian");
Person person = data.queryFirst("SELECT * FROM person
     WHERE person.name=?name", Person.class, person);
```

Data API: query Beans

- The class of the return Bean type is passed in as a parameter

```
Person person = data.queryFirst("SELECT * FROM person
                        WHERE person.name=?", Person.class, "Brian");

List<Person> people = data.queryList("SELECT * FROM person",
                        Person.class);

Person[] people = data.queryArray("SELECT * FROM person",
                        Person.class);

Iterator<Person> people = data.queryIterator("SELECT * FROM person",
                        Person.class);
```

- Beans, Maps, Arrays, Collections, Iterators, or your own data

Data API: query Maps

- Query result can be returned as a Map

```
Map<String,Object> person = data.queryFirst("SELECT * FROM person
                                    WHERE person.name=?", "Brian");

List<Map<String,Object>> people = data.queryList("SELECT * FROM
                                    person WHERE person.name LIKE ?", "Br%");
```

- In the result Map
  - the column names become String keys
  - the column values become Object values

Data API: updates

- Write operations are performed via “update” method:
  - Insert

```
int rowsAffected = data.update("INSERT INTO person (id, name,
                        address)    VALUES (?id, ?name, ?address)", person);
```

  - Many inserts and updates – ***automatically batches***

```
rowsAffected = data.updateMany("INSERT INTO person (id, name,
                        address) VALUES (?id, ?name, ?address)", people);
```

  - Update

```
int rowsAffected = data.update(“UPDATE person set name = ?name,
                        address = ?address WHERE id = ?id", person);
```

  - Delete

```
int rowsAffected = data.update(“DELETE FROM person where id=?”, id);
```

Result Handlers

All Handlers are first class - equal from the runtime's point of view

1. Generic based on reflection
   - Ex: TwoWayJoin, ThreeWayJoin, Bean and Map Factories
2. Handcrafted custom handlers – XML and JSON
   - Ex: JSONResultHandler,
   - Ex: XMLResultHandler
3. Generated from tools
   - Ex: EmployeeDepartmentJoin, generated from Employee, Department, Select statement

Example – create XML from a query

- Create XML from a query:

```
String xml = d.query("select * from Department", new
XMLResultHandler());
```

- XMLResultHandler has one method: `String handle(ResultSet rs)`

```
sb.append("\t<"+ m.getTableName(x) +">");
for (int x=1; x<=cols; x++) {
  sb.append("<"+ m.getColumnName(x) +">");
  sb.append(rs.getString(x));
  sb.append("</"+ m.getColumnName(x) +">");
  }
sb.append("\t</"+ m.getTableName(x) +">");
```

- Output:

```
<result>
  <DEPARTMENT><DEPTNO>A00</DEPTNO><DEPTNAME>SPIFFY
    COMPUTER SERVICE DIV.</DEPTNAME> <MGRNO>000010</MGRNO> ...
  </DEPARTMENT>
</result>
```

Example – create JSON from a query

- Create JSON from a query:

```
String json = d.query("select * from Department", new
JSONResultHandler());
```

- JSONResultHandler has one method:

```
String handle(ResultSet rs)

sb.append("{");
for (int x=1; x<=cols; x++) {
  sb.append("\""+ m.getColumnName(x) +"\"=\"");
  sb.append(rs.getString(x) +"\"");
  if (x<cols) sb.append(",");
  }
sb.append("}");
```

- Output:

```
[{"DEPTNO"="A00","DEPTNAME"="SPIFFY COMPUTER SERVICE
 DIV.","MGRNO"="000010","ADMRDEPT"="A00","LOCATION"="null"},
{"DEPTNO"="B01","DEPTNAME"="PLANNING","MGRNO"="000020",
 "ADMRDEPT"="A00","LOCATION"="null"} ... ]
```

Named Query Style – XML SQL declaration and OR mapping

```
Iterator<EObjAddress> getAddress(long address_id);

<entity-mappings xmlns="http://java.sun.com/xml/ns/persistence/orm">
  <named-native-query name="junit.addressData#getAddress(long)">
        <query>
            select * from ADDRESS where ADDRESS_ID = ?
        </query>
  </named-native-query>

  <entity class="junit.addressData.EObjAddress">
    <table name="ADDRESS"/>
        <attributes>
            <basic name="addressIdPK">
              <column name="ADDRESS_ID" />
            </basic>
        </attributes>
    </table>
  </entity>
</entity-mappings>
```

SQL - method

SQL – Property name

XML document is JPA compatible

Method Style – declaring the method

- SQL can be in Annotation, or
- SQL can be in XML file
  - Source does not have dependencies on JLINQ API
  - Manage SQL separately, uses JPA XML format

```
@Select(sql="select ADDRESS_ID, COUNTRY_TP_CD,
  RESIDENCE_TP_CD, PROV_STATE_TP_CD, ADDR_LINE_ONE,
  P_ADDR_LINE_ONE , ADDR_LINE_TWO, P_ADDR_LINE_TWO,
  ADDR_LINE_THREE, P_ADDR_LINE_THREE, CITY_NAME,
  POSTAL_CODE, ADDR_STANDARD_IND, OVERRIDE_IND,
  RESIDENCE_NUM, COUNTY_CODE, LATITUDE_DEGREES,
  LONGITUDE_DEGREES, LAST_UPDATE_DT, LAST_UPDATE_USER,
  LAST_UPDATE_TX_ID, POSTAL_BARCODE from ADDRESS where
  ADDRESS_ID= ?")
Iterator<Address> getAddress(long address_id);
```

Bean mapping annotation

- Bean can be Annotated, or
- SQL can be in XML file
  - Source does not have dependencies on JLINQ Annotations
  - Manage mapping separately, uses JPA XML format

```
@Table(name="ADDR", schema="ADMIN")
public class Address {
  @Column(name="ADDRESS_ID") protected long addressIdPK;
  @Column(name="COUNTRY_TP_CD") protected long countryTpCd;
  @Column(name="RESIDENCE_TP_CD") protected long residenceTpCd;
  @Column(name="PROV_STATE_TP_CD") protected long provStateTpCd;
  protected String addrLineOne;
  protected String pAddrLineOne;
  protected String addrLineTwo;
  protected String pAddrLineTwo;
  @Column(name="ADDR_LINE_THREE") public String addrLineThree;
  @Column(name="P_ADDR_LINE_THREE") public String pAddrLineThree;
  @Column(name="CITY_NAME") public String cityName;
}
```

Optional table, schema name

Optional column name

Control type, visibility

SQL collection query example – join in memory

- Query in-memory **unmanaged** objects
- Query is over the **original** objects on the heap: **no copies, no extra storage**
- Join with results from data server query
- **Full standard SQL**
- Ex: In memory query over `Customers` collection from the data server

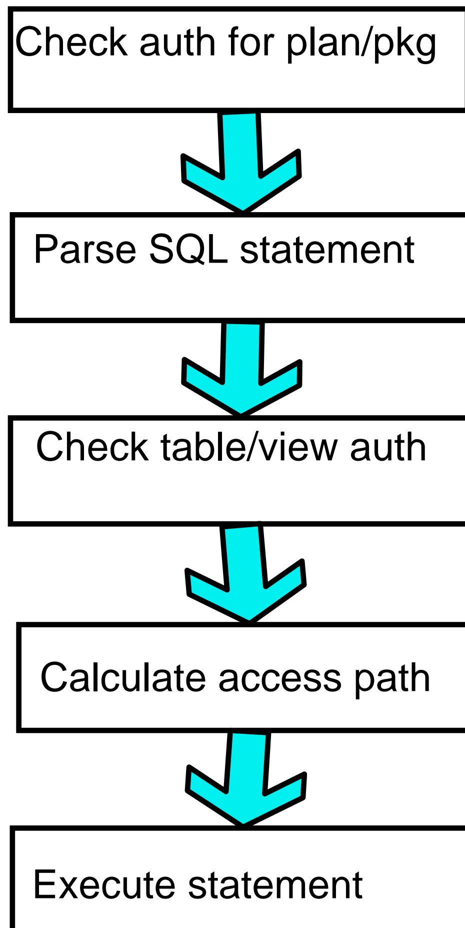
```
Customer[] customers = ...;
Purchase[] purchases = ...;
int zip = 54321;
List<Address> addresses = data.queryList(
   "select c.street, c.city from ? c, ? p where c.zip=? and
  c.id=p.cid", Address.class, customers, purchases, zip);
for(Address a : addresses)
{
  System.out.println(a.street+", "+a.city);
}
```

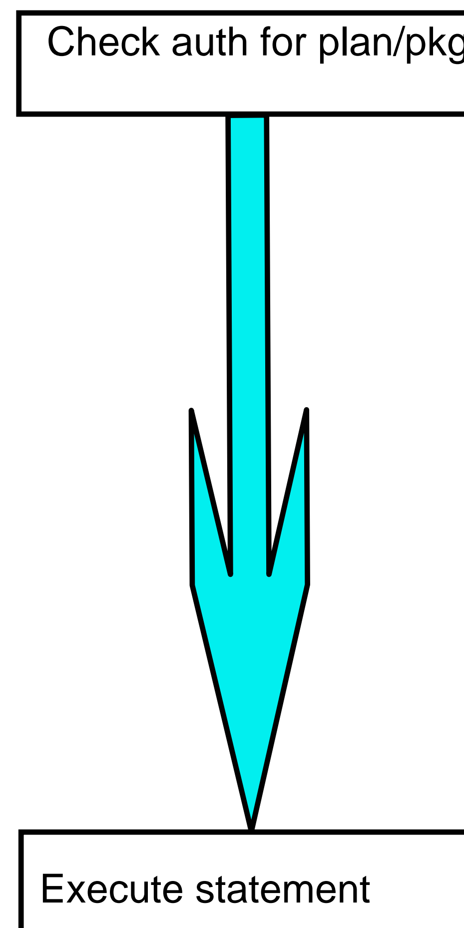
JLINQ Technology Value Proposition

- Benefits to all database vendors:
  - Single API for queries to relational, persistence layer, cache, and in-memory objects for both Relational and XML
  - Language Integration with Java
  - Tooling to greatly simplify tasks associated with coding SQL in Java
  - Apps can easily issue complex queries (multi-table joins, nested subselects, etc.)
  - Simple API syntax that eliminates the need for “get” and “set” methods
  - API returns objects, reducing the object/relational impedance mismatch
  - By default, runtime will be existing JDBC or CLI interface, so the API will be portable across all databases on day one

- Benefits to IBM products:
  - Single API for joins in-memory across cache, relational, and in-memory objects. Joins in-memory across XML documents.
  - Static SQL for better performance
  - Access path locked-in at deployment – reliable production runtime behavior
  - Multiple versions of the access path (fallback to prior version easily)
  - Candle Omegamon and DB2 Performance Expert for deep performance metrics including historical trends (app-level or statement-level)
  - All SQL statements and access paths recorded in the DB2 server, which helps DBA with problem determination and capacity planning
  - Application origin captured for all SQL statements for rapid problem source identification

Static SQL is FASTER!!!





Toughest issue for Web applications – Problem diagnosis and resolution

Web Browser Users

Web Server

Application Server

Business Logic

EJB Query Language

Data Access Logic

Persistence Layer

DB2 Java Driver

DB2 Server

JDBC Package

Simplifying Problem Determination Scenario

Java Profiling

Application Developer

- Available for each db access
  - –SQL text generated
  - –Access path
  - –Cost estimates
  - –Estimated response time
  - –Elapsed & CPU time
  - –Data transfer (getpages)
  - –Tuning advice

J-LinQ

Database Administrator

- Available for each SQL
  - –Application name
  - –Java class name
  - –Java method name
  - –Java object name
  - –Source code line number
  - –Source code context
  - –J-LinQ transaction name
  - –Last compile timestamp

DRDA Extentions

J-LinQ with IBM Runtime/Tooling

Application Server

EJB Query Language

Data Access Logic

Persistence Layer

DB2 Java Driver

A1 A4 A3 A2 A5 A6

DB2 or IDS

A1 A4 A5

Application Meta data

A1 A2 A3

A4 A5 A6

Catalog data for SQL statements

A1 A2 A3 A4 A5 A6

Compile-time application details:
- Java class/line number for SQL
- Original query syntax
- Final SQL query syntax

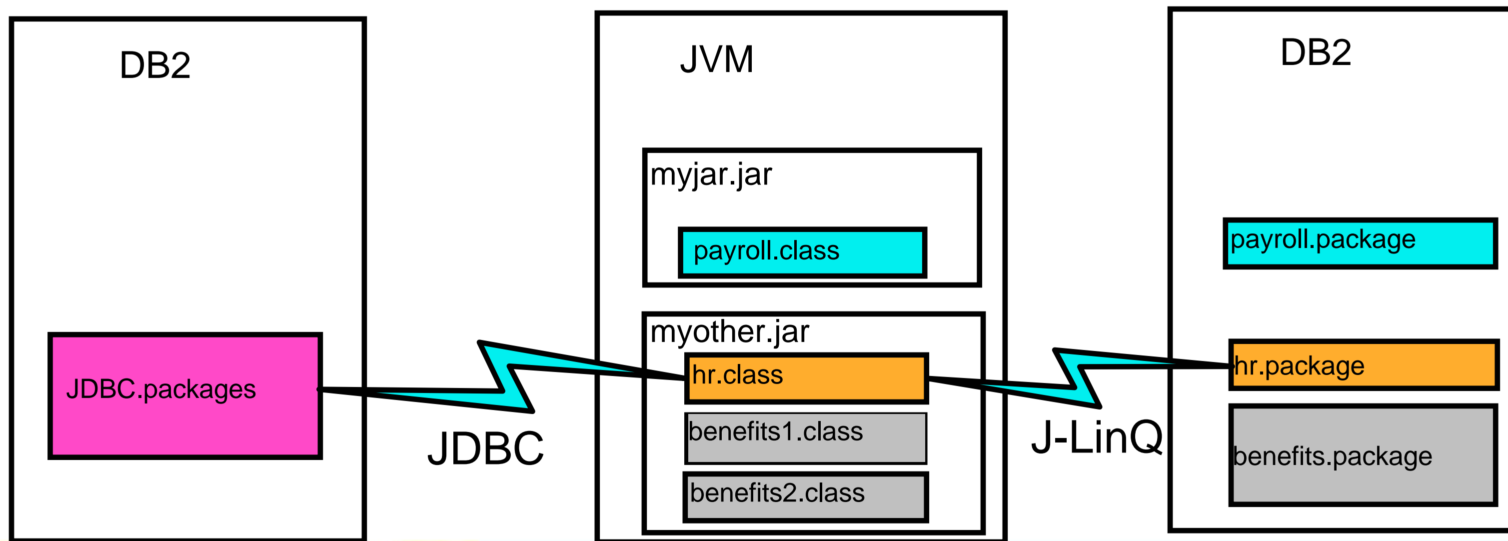
Performance data by application:
- CPU and elapsed time
- Getpages, locks, etc.

App	CPU
A1	2.1
A4	8.3
A5	22.0

SQL Tuning

What can we do to improve HR performance?

- JDBC
  - SQL statements are not stored in DB2
  - run an SQL performance trace and crawl through the details...
- J-LinQ APIs
  - all static SQL statements in recorded in DB2 catalog
  - package level accounting (CPU time, SQL counts, getpages, etc.)



Viper II Deliverables for Java

- JDBC 4.0
- Simplified SOA runtime support for stored procedure and SQL query applications
- J-LinQ Technology Preview