Java Persistence and the Open SQL Engine in SAP Web Application Server 6.40

Marc Chan
NetWeaver RIG US
SAP Labs
Agenda

Java Persistence @ SAP

Database Architecture

Runtime: Open SQL Engine

Developer View

Open SQL / SQLJ

Summary
Java Persistence Framework: Two Goals

Strict Java standard orientation

- Lower TCO …
- Customers like to be independent from any software manufacturer
- A huge amount of skilled developers worldwide

In doing so …

Enable the development of real life database centric business applications in Java that are portable between different DBMS.

- SAP transfers experiences learned with the proven and successful ABAP server to Java world!
A "normal" Java method executes within microseconds.

A database access normally needs a few milliseconds or even seconds to return a result.

General rule:
The performance of a business transaction is primarily determined by its database accesses.
Java Persistence Framework: Key Features

As many Java servers, SAP Web AS:

Complies with the established database programming standards for Java:

- basic platform: JDBC (as designated by J2EE)
- J2EE database programming model adjusted with encouraging standards: SQLJ and JDO APIs

The proprietary infrastructure of SAP Web AS:

Improves performance, reliability and supportability transparently to applications.

Enables SQL based database programming portable across supported RDBMS.
J2EE and database programming are complex technologies and it is indispensable to apply them properly.

Therefore, SAP releases recommendations restricting J2EE standard if in conflict with supportability, portability, reliability …

Restrict, but do not contradict
Agenda

Java Persistence @ SAP

Database Architecture

Runtime: Open SQL Engine

Developer View

Open SQL / SQLJ

Summary
Database programming starts with the database.

The J2EE cluster runs on a **single central database**.

This is your **Java database**.
Database is **relational** (RDBMS) and **accessible** with **JDBC**

**RDBMS** is supported by **SAP Web AS**

**Unicode** only

**Default database schema** contains

- Java Server data
- Application data
SAP Web AS 6.40 Java: Supported RDBMS

- Oracle
- MySQL MaxDB
- Microsoft SQL Server
- IBM DB2 UDB for UNIX and Windows
- IBM DB2 UDB for z/OS
- IBM DB2 UDB for iSeries

... complemented with a JDBC database driver approved by SAP.
SAP Web AS 6.40 is **two servers put into one.**

However

- Separate database storage
- Logically separate databases, separate schemas

**Preferred configuration**

- Single database, separate schemas
- Single database parameterisation, as of ABAP
Separate Storage and Connect Applications

Consequences of separation

- No database transactions spanning ABAP and Java stack
- No direct table access across ABAP / Java stacks
- No namespace coordination for database objects

Collaboration at the component level: JCo, Web Services, XI, ...

© SAP AG 2004, Java Persistence and Open SQL Engine in SAP Web AS 6.40 / Slide 12
Agenda

Java Persistence @ SAP

Database Architecture

Runtime: Open SQL Engine

Developer View

Open SQL / SQLJ

Summary
At runtime, there is an active persistence framework: **Open SQL Engine.**

Open SQL Engine wraps the proprietary JDBC driver transparently to your application.

Comparable to Open SQL for ABAP.

**JDBC code**

**Open SQL**

**JDBC Driver**
The Three Support Levels of Open SQL Framework

The support level is set for each connection pool.

The default setting is ‘Open SQL’
### Open SQL Framework: Adding Value Layer by Layer

#### Open SQL
- SQLJ
- Portability check SQL and JDBC
- Java Dictionary
- Table Buffer

**Generally recommended!**
**Obligatory for internal SAP projects!**

#### Native SQL
- SQL Trace
- SQL Statement Pooling

**Recommended if:**
- you cannot use the default database schema.
- database tables already exist.

#### Vendor SQL
- Database Connection Pool
- Full JDBC standard API
- Vendor-specific SQL statements

**Default database schema only**
SAP Web AS Java transparently caches open database connections in a server-side database connection pool.

Open and release database connections without delay!

Access the default database schema through the pre-configured default connection pool.

You must not change the settings of the default connection pool!
Statement Pooling

Still active behind the scenes:

Statement Pooling:
avoids repeated (expensive) preparation of SQL statements in the database.

A server-side pool for both “Prepared” and “Callable” JDBC statement types.

Prepared statement pool for each physical database connection.
### SQL Trace Evaluation:

- **List for trace id 20030709114614481 from node 7564950**

<table>
<thead>
<tr>
<th>Time</th>
<th>Duration</th>
<th>Jdbc method Id</th>
<th>No.</th>
<th>Result</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>11:46:18,070</td>
<td>931</td>
<td>DirectPreparedStatement.executeQuery</td>
<td>931</td>
<td>SELECT &quot;PATH&quot;,&quot;PARENTCID&quot;,&quot;CACHEMODE&quot;,&quot;CTYPE&quot; FROM &quot;JZEE_CONFIG&quot; WHERE &quot;CID&quot; = ...</td>
<td></td>
</tr>
<tr>
<td>11:46:18,073</td>
<td>1267</td>
<td>DirectResultSet.next()</td>
<td>1</td>
<td>true</td>
<td>next()</td>
</tr>
<tr>
<td>11:46:18,076</td>
<td>6</td>
<td>DirectResultSet.close()</td>
<td>6</td>
<td>close</td>
<td></td>
</tr>
<tr>
<td>11:46:18,077</td>
<td>817</td>
<td>DirectPreparedStatement.executeQuery</td>
<td>817</td>
<td>SELECT &quot;PATH&quot;,&quot;CID&quot;,&quot;CACHEMODE&quot;,&quot;CTYPE&quot; FROM &quot;JZEE_CONFIG&quot; WHERE &quot;PARENTCID&quot; = ...</td>
<td></td>
</tr>
<tr>
<td>11:46:18,075</td>
<td>603</td>
<td>DirectResultSet.next()</td>
<td>603</td>
<td>true</td>
<td>next()</td>
</tr>
<tr>
<td>11:46:18,062</td>
<td>12</td>
<td>DirectResultSet.next()</td>
<td>1</td>
<td>false</td>
<td>next()</td>
</tr>
<tr>
<td>11:46:18,083</td>
<td>7</td>
<td>DirectResultSet.close()</td>
<td>7</td>
<td>close</td>
<td></td>
</tr>
<tr>
<td>11:46:18,084</td>
<td>638</td>
<td>DirectPreparedStatement.executeQuery</td>
<td>638</td>
<td>SELECT &quot;NAME&quot;,&quot;ISFILE&quot;,&quot;DTYPE&quot;,&quot;V8GINT&quot;,&quot;V8DOUBLE&quot;,&quot;VSTR&quot;,&quot;VBYTES&quot; FROM &quot;JZEE_C...</td>
<td></td>
</tr>
<tr>
<td>11:46:18,066</td>
<td>1019</td>
<td>DirectResultSet.next()</td>
<td>1019</td>
<td>true</td>
<td>next()</td>
</tr>
<tr>
<td>11:46:18,084</td>
<td>11</td>
<td>DirectResultSet.next()</td>
<td>11</td>
<td>false</td>
<td>next()</td>
</tr>
<tr>
<td>11:46:18,086</td>
<td>9</td>
<td>DirectResultSet.close()</td>
<td>9</td>
<td>close</td>
<td></td>
</tr>
</tbody>
</table>
Caches table data inside VM. One table buffer in each VM (Server node). Synchronization possibly delayed.
Enable Table Buffering: Developer Studio

Buffer small tables that are mostly read and rarely changed and certain inconsistencies are acceptable.

Is it a good idea to cache the table TMP_CUSTOMER?

Choose the granularity of buffering

Answer: No
Agenda

Java Persistence @ SAP

Database Architecture

Runtime: Open SQL Engine

Developer View

Open SQL / SQLJ

Summary
Portable Table Definition: Java Dictionary

Development: Portable definition, creation and modification of database tables.

Runtime: Code check against the Table Catalog.

True "write once, run anywhere" for data definition.
Developer Studio: Dictionary Perspective

Pre-defined JDBC like portable data types

here, not there
## Built-In Dictionary Database Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>string(N)</td>
<td>varying length character strings with a maximum length of N characters</td>
</tr>
<tr>
<td></td>
<td>Unicode only</td>
</tr>
<tr>
<td>binary(N)</td>
<td>byte strings with a fixed size of N bytes, if short (≤255 B)</td>
</tr>
<tr>
<td></td>
<td>varying length byte string with maximum length of N bytes, if longer</td>
</tr>
<tr>
<td>short</td>
<td>“2 Byte” signed integral numbers, exact numeric</td>
</tr>
<tr>
<td>integer</td>
<td>“4 Byte” signed integral numbers, exact numeric</td>
</tr>
<tr>
<td>long</td>
<td>“8 Byte” signed integral numbers, exact numeric</td>
</tr>
<tr>
<td>float</td>
<td>“4 Byte” floating point numbers, approximate numeric</td>
</tr>
<tr>
<td>double</td>
<td>“8 Byte” floating point numbers, approximate numeric</td>
</tr>
<tr>
<td>decimal(P, S)</td>
<td>exact numeric numbers with precision P &gt; 0 and scale S ≥ 0</td>
</tr>
<tr>
<td>date</td>
<td>date values expressed by year, month and day</td>
</tr>
<tr>
<td></td>
<td>no time zone attached</td>
</tr>
<tr>
<td>time</td>
<td>time values expressed by hour, minute and second</td>
</tr>
<tr>
<td></td>
<td>no time zone attached</td>
</tr>
<tr>
<td>timestamp</td>
<td>timestamps expressed by date, time and fractional seconds</td>
</tr>
<tr>
<td></td>
<td>time zone attached, SAP uses UTC</td>
</tr>
</tbody>
</table>

**Given Type Mapping:** Dictionary ↔ JDBC ↔ Java
Code Portability with Open SQL Engine

Java Program

Relational Persistence (SQL)
- Open SQL / SQLJ
- Open SQL / JDBC

Object Relational Persistence
- EJB (CMP)
- JDO

Open SQL Engine
- Open SQL Processor
- DB Access Layer
  - Table Buffer
  - Table Catalog
  - Statement Pool
  - SQL Trace

Relational Database System

- Native SQL / JDBC
- Vendor JDBC

© SAP AG 2004, Java Persistence and Open SQL Engine in SAP Web AS 6.40 / Slide 26
Developer View: Standardized Persistence APIs

Standardized Persistence APIs

- **JDBC, SQLJ**
  - relational, SQL-based coding: **expressive!**

- **JDO, EJB CMP entity beans**
  - object relational coding: **SQL-free! Portable!**

- **EJB BMP entity beans**
  - object relational, SQL based coding.
  - Alternatively: Can map to an **ABAP backend acting as the data store!**
Choose the adequate Java persistence API!
SQLJ: SQL Check at Development Time

**SQL statement check** against the **Open SQL Grammar** and against **Java Dictionary Table Definitions**:

At design time!

I know right away my code is correct and portable.

Use SQLJ for *static* SQL and JDBC for *dynamic* SQL
### J2EE Persistence APIs – Use of Open SQL Engine

<table>
<thead>
<tr>
<th></th>
<th>Open SQL</th>
<th>Native SQL</th>
<th>Vendor SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJB</td>
<td>yes</td>
<td>supported</td>
<td>tolerated</td>
</tr>
<tr>
<td>JDO</td>
<td>yes</td>
<td>supported</td>
<td>as required by JDO certification</td>
</tr>
<tr>
<td>JDBC</td>
<td>yes</td>
<td>supported</td>
<td>tolerated</td>
</tr>
<tr>
<td>SQLJ</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**Native vs. Open SQL**
- Open SQL language complies to SQL standard if at all possible
- Open SQL is NOT JDBC-compliant as it contains restrictions
- Valid Open SQL statements do not in general run on Native SQL
Agenda

Java Persistence @ SAP

Database Architecture

Runtime: Open SQL Engine

Developer View

Open SQL / SQLJ

Summary
SQLJ – What is it?

**SQLJ**

- Standard for using **Java** and **SQL** together
  
  SQLJ = Java + SQL

**SQLJ**

- Standard way for embedding static SQL into Java programs

Java source text

```java
ResultSet rs = stmt.executeQuery("SELECT cs, ci FROM dbtab");
```

- As opposed to **JDBC**
  
  Where SQL statement is a String argument of a JDBC method
SQLJ – How Does It Work?

Special Syntax for identifying SQL

---

The actual SQL statement

Java source text

#sql ... { SQL } ;

---

SQLJ source files with own extension .sqlj

Pre-processing by the SQLJ translator
SQLJ – Translating SQLJ Sources

MyClass.sqlj

MyClass.java

SQLJ source file

SQLJ Translator

MyClass.java

Java source file

javac

MyClass.class

Java class file

β replaces SQLJ clauses by calls to the SQLJ runtime

β transparently integrated into the IDE
SQLJ – Syntax

Java text is case sensitive.

```sql
#sql [ctx] { Select col into :var FROM tab };
```

Host variables are prefixed with " : "

SQL text is enclosed by "{" and "}".

SQL text is case insensitive.
Creating SQLJ sources

- **File**
  - New
  - Other
  - Persistence

- **Context menu**
  - New
  - Other
  - Persistence
Logical Catalog

- Offline description of database tables
- Represented by a Dictionary Project
- Used by the SQLJ translator
  for checking SQL statements

Usage of a Logical Catalog

- By referring to the corresponding Dictionary Project
Working with SQLJ

JDBC

Data source (jdbc/myDB)

Get a connection from the data source

Work with the connection

- Send SQL statements
- Iterate over result sets
- Release resources

Close the connection

SQLJ

Connection Context Class

Connection Context

Named Iterator
SQLJ – Connection Context Class

Connection Context Class
- Represents a data source
- Represents a logical catalog at design-time

Declaration

```sql
#sql context MyCtx
with (dataSource = "java:comp/env/jdbc/myDB");
```

Connection Context Class = Ordinary Java Class
- Created by the SQLJ translator
- for each "#sql context" clause
SQLJ – Connection Contexts

Connection Context (Object)

- Represents a database connection
- Instance of a connection context class

Example

```java
MyCtx ctx = null;

try {
    ctx = new MyCtx();
    #sql [ctx] { DELETE FROM dbtab WHERE key = 123 };
}
finally {
    ctx.close();
}
```

Get a connection

Always close the connection in the finally block!
SQLJ – Insert, Update, Delete

Insert Statement

```sql
#sql [ctx] { INSERT INTO dbtab
  (Id, Lastname, FirstName)
  VALUES (1, 'Smith', 'John');
}
```

Update Statement

```sql
#sql [ctx] { UPDATE dbtab SET name = :newName
  WHERE name = :oldName }
```

Delete Statement

```sql
#sql [ctx] { DELETE FROM dbtab WHERE key = 123 }
```
Fetch a **single** row of a Result Set into host variables

The Result Set must contain only one row

- specify the full key
- use aggregate functions

```sql
String s;
int i, keyVal = 1234;

#sql [ctx] { SELECT cs, ci INTO :s, :i
  FROM dbtab
  WHERE key = :keyVal };
```

The selected columns are fetched into host variables.

Ensure that the result set contains only one row
**SQLJ – Single Row Query (Exceptions)**

Empty result set: `com.sap.sql.NoDataException`

More than one row: `com.sap.sql.CardinalityViolationException`

```java
String s;

try {
    #sql [ctx] { SELECT cs INTO :s FROM tab };

} catch (NoDataException ex) {
    // no data
}

} catch (CardinalityViolationException ex) {
    // too many rows
}
```
SQLJ – Queries Returning More Than One Row

How to access the rows of the result of a SQL query?

Named Result Set Iterators
- Java class declared by `#sql iterator`
- Like a Result Set / Cursor
- Strongly typed
  - Name-Type pair for columns
  - Type compatibility checked at compile-time

```java
#sql iterator NamedIter ( String CS, int CI );

NamedIter nIter;

#sql [ctx] nIter = { SELECT cs, ci FROM dbtab };
```

The result of the query is assigned to the iterator `nIter`. 
SQLJ – Using Named Result Set Iterators

```java
#sql iterator NamedIter ( String CS, int CI );
int i;
String s;
NamedIter nIter;

#sql [ctx] nIter = { SELECT cs, ci FROM dbtab };

while ( nIter.next() ) {
    System.out.println( nIter.CS() );
    ...
    System.out.println( nIter.CI() );
}

nIter.close();
```

- **Forward only iterator**
- **Move to the next row with** `next()`
- **Close iterator with** `close()`
SQLJ – Host Variables

Prefixed with ":"  

Static type checking of host variables

Available for IN and OUT parameters

```java
int keyValue = 5;
String value = null;

#sql [ctx] { SELECT col INTO :value
  FROM dbtab
  WHERE key = :keyValue };
```
Enclosed between "(:" and ")"

May have side effects

**IN** and **OUT** expressions evaluated before execution of the statement in lexical order

```java
String[] values = new String[5];
MyClass ref = new MyClass();
int i = 3;

#sql [ctx] { SELECT col INTO :(values[++i])
FROM dbtab
WHERE key = :(ref.getKey(i)) };
```

Evaluated first before execution

Evaluated second before execution
SQLJ – What About Dynamic SQL Statements?

Dynamic SQL Statement
- not known statically (compile-time)
- constructed at run-time

SQLJ not applicable!

Resort to JDBC
- Dynamic call-level API
- SQL statement = String argument of JDBC method
SQLJ – Interoperability with JDBC

Usage of SQLJ + JDBC
- in the same transaction
- on the same connection
- on the same result set

Easy to accomplish
- since SQLJ is built on top of JDBC
Connections – SQLJ → JDBC

Example:

```java
#sql context MyCtx
    with (dataSource = "java:comp/env/jdbc/myDB");

MyCtx myCtx = new myCtx();

java.sql.Connection myCon = myCtx.getConnection();
```
Connections – JDBC → SQLJ

Example:

```java
#sql context MyCtx;

database.Connection myCon = myDS.getConnection();

MyCtx myCtx = new MyCtx( myCon );
```
Result Sets – SQLJ → JDBC

Example:

```java
#sql iterator MyIter ( String CS, int CI );
#sql [ctx] myIter = { SELECT cs, ci FROM dbtab };
java.sql.ResultSet rs = myIter.getResultSet();
```
Result Sets – JDBC → SQLJ

Example:

```java
#sql iterator MyIter ( String CS, int CI );

String query = "SELECT cs, ci FROM dbtab";
java.sql.ResultSet rs = stmt.executeQuery(query);

MyIter myIter;
#sql myIter = { CAST :rs };
```
SQLJ – Summary

Benefits

- Simpler, more compact and more robust programs
- Compile-time checks (syntax / types)
- Integrated into NetWeaver Developer Studio
  - SQLJ translator transparently invoked
  - Syntax high-lighting
- Integrated into the Java Development Infrastructure

Always use SQLJ for static SQL

Use JDBC only for dynamic SQL
Agenda

Java Persistence @ SAP
Database Architecture
Runtime: Open SQL Engine
Developer View
Open SQL / SQLJ
Summary
Java Persistence – Summarizing the Benefits

Database Portability
- Oracle, DB2, MSSQL, SAPDB

Standard Conformant APIs
- JDBC, SQLJ, JDO, EJB entity beans

Performance Enhancements
- Table Buffer
- Statement Pool
- Connection Pool

Supportability
- SQL Trace

Seamlessly integrated into the J2EE server
Comes with the J2EE server
Runs out of the box
SAP Library in SAP NetWeaver Developer Studio

- Architecture Manual
  - “Java Persistence”
- Development Manual
  - “Java Dictionary”
  - Developing Business Logic
    - “Java Persistence”
- Reference Manual
  - “Java Persistence Reference”
Q&A