High Performance Persistence

Donald Smith
Oracle Corporation
Goal

Show how you build high performance J2EE applications that scale using an object-relational persistence layer
Agenda

Persistence Architecture Overview

Performance Points:
- Models and Mapping
- Inheritance
- Locking
- Querying
  - Lazy Reading
  - N+1 Reads
- Projections
- Transactions
- Caching

Leveraging the Database and JDBC
Agenda

Persistence Architecture Overview

Performance Points:
- Models and Mapping
- Inheritance
- Locking
- Querying
  - Lazy Reading
  - N+1 Reads
  - Projections
- Transactions
- Caching

Leveraging the Database and JDBC
Java Access of Relational Data

- Direct JDBC
  - Direct SQL calls hard coded in Java
  - Use result sets ("rows") directly

- Persistence layer
  - Accessed as objects or components
  - Transparent that the data is stored in RDB
  - Persistence layer in middle tier handles object-relational mapping and infrastructure
  - Required if doing business logic in the middle tier!
JDBC—Java Database Connectivity

- Java standard for accessing databases
- JDBC is simply the database connection utilities Java developers need to build upon
### JDBC

- Simply – data source connection
- OK with:
  - “Window on data” applications
  - Business logic entrenched on database
  - Java nothing more than GUI tool
- Is it your only option for “high performance persistence”? – NO!
Persistence Layer

- Abstracts persistence details from the application layer

Java & Web Services

Persistence Layer

- Results are objects
- Results are returned as raw data
- Object creation and updates through object-level API
- API uses SQL or database specific calls

Object-level querying and creation
Persistence Layer

Object-level querying and creation results are objects. Results are returned as raw data.

J2EE & Web Services

Object creation and updates through object-level API

Persistence Layer

JDBC

API uses SQL or database specific calls

Objects

Rows

SQL
ORM Persistence Entity Types

- **EJB 2 Persistence: Entity Beans**
  - **BMP**: Developer must hand code persistence “life cycle” calls generate by J2EE Container
  - **CMP**: More Automatic Persistence

- **“Plain Old Java Objects” (POJO) Persistence**
  - Oracle TopLink
  - Open Source: JBoss Hibernate
  - DAO Pattern: Custom, Generators
  - JDO Implementations

- **EJB 3.0 Persistence: Entity Beans == POJOs**
iBATIS – Somewhere in between...

iBATIS is not meant to replace Object-Relational Mapping (ORM) tools. It is a low-level framework that allows you to directly hand-code your SQL statements and map them to Java object. You don’t need to lookup DataSource, get connections, create prepared statements, parse ResultSet or even cache the results – iBATIS does it all for you. Under the covers, iBATIS creates a PreparedStatement, sets the parameters (if any), executes the statements and builds a Map or JavaBean object from the ResultSet.

Summarized from Vinny Carpenter’s BLOG entry:

http://www.j2egeek.com/blog/2005/01/31/ibatis-where-have-you-been-all-my-life/
Agenda

Persistence Architecture Overview

Performance Points:

Models and Mapping
Inheritance
Locking
Querying
  Lazy Reading
  N+1 Reads
Projections
Transactions
Caching
Leveraging the Database and JDBC
**Impedance Mismatch**

- Difference in relational and object technology know as "**object-relational impedance mismatch**"

- Challenging problem
  - Requires relational and object expertise
# Impedance Mismatch

<table>
<thead>
<tr>
<th>Factor</th>
<th>J2EE</th>
<th>RDBMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Logical Data Format</td>
<td>Objects, methods, inheritance</td>
<td>Tables, SQL, stored procedures</td>
</tr>
<tr>
<td>Scale</td>
<td>Hundreds of megs</td>
<td>Gigabytes, terabytes</td>
</tr>
<tr>
<td>Relationship</td>
<td>Object references</td>
<td>Foreign keys</td>
</tr>
<tr>
<td>Uniqueness</td>
<td>Internal object identity</td>
<td>Primary keys</td>
</tr>
<tr>
<td>Key Skills</td>
<td>Java development, object modeling</td>
<td>SQL, Stored Procedures, data management, Data modeling</td>
</tr>
<tr>
<td>Tools</td>
<td>IDE, Source code management, Object Modeler</td>
<td>Schema designer, query manager, performance profilers, database config</td>
</tr>
<tr>
<td>Corporate Org. Structure</td>
<td>“Newer technology” often with weak organizational ties to database mgmt</td>
<td>Often mature infrastructure with significant legacy considerations</td>
</tr>
</tbody>
</table>
Mapping

- Object model and Schema must be mapped
  - True for any persistence approach
- Most contentious issue facing designers
  - Which classes map to which table(s)?
  - How are relationships mapped?
  - What data transformations are required?
Mapping Process

- Map Object Model to schema

Diagram:
- Mapping Tool
- Mapping Metadata (XML/Annotation)
- Schema
- Java source
- Java IDE

Java IDE
The underlying mapping flexibility is very important.
Modeling and Mapping

- Options:
  - Generate Object Model from schema
  - Generate schema from Object Model
  - Meet in the Middle
    - Existing Object Model
    - Existing Schema
Modeling and Mapping Generation

- Excellent productivity boost at start of project
  - Don’t take the generated code/schema as gospel!
  - Tools are smart not psychic
- Many legacy schemas will lead to classic problems discussed shortly...
Recommendations

- Ensure ORM layer allows customization of mappings
- Match the approach to current and long term project requirements
  - New/Flexible schema versus legacy static schema
  - Future enhancements
  - Maintenance
- If absolutely tied to a schema/model with no hope of change – reconsider strategy
Agenda

Persistence Architecture Overview

Performance Points:

- Models and Mapping
- Inheritance
- Locking
- Querying
  - Lazy Reading
  - N+1 Reads
  - Projections
- Transactions
- Caching
- Leveraging the Database and JDBC
Impedance Mismatch: Inheritance

- Java
  - Used to extend (share) common state & behavior

- Relational
  - Shared data can be normalized into common table
  - Adds additional unique constraint within common table
Inheritance

- Very straightforward at first, but can become challenging because of the number of possible solutions
- Compounded with relationships
- Can be source of contention with DBA – generally leads to lots of joining on queries!
“Leaf Table Mapping”

Map only concrete classes

<table>
<thead>
<tr>
<th>Root {a}</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: int</td>
</tr>
<tr>
<td>var: String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sub1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub1var: String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sub2 {a}</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub2var: int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUB1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SUB22</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>13</td>
</tr>
</tbody>
</table>
“Leaf Table” Issues

- How do you query for the Root class?
- Data is not normalized
- Performance Impact:
  - **PRO**: No joins or table contention between types
  - **CON**: No heterogeneous query results

Is this your fastest option? If you only do leaf queries, perhaps it is...
"All Table Mapping"

Root\{a\}
- id: int
- var: String

Sub1
- sub1var: String

Sub2\{a\}
- sub2var: int

Sub22
- sub22var: String

Map all classes

ROOT

<table>
<thead>
<tr>
<th>ID</th>
<th>VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Foo</td>
</tr>
<tr>
<td>13</td>
<td>Bar</td>
</tr>
</tbody>
</table>

Problem!
How to tell type?!

SUB1

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB1VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Pish</td>
</tr>
</tbody>
</table>

SUB2

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB2VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>677</td>
</tr>
</tbody>
</table>

SUB22

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB22VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Fim</td>
</tr>
</tbody>
</table>
Side Bar: Identifying Type

**Root {a}**
- id: int
- var: String

**Sub1**
- sub1var: String

**Sub2 {a}**
- sub2var: int

**Sub22**
- sub22var: String

---

**Map all classes**

**ROOT**

<table>
<thead>
<tr>
<th>ID</th>
<th>VAR</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Foo</td>
<td>S1</td>
</tr>
<tr>
<td>13</td>
<td>Bar</td>
<td>S22</td>
</tr>
</tbody>
</table>

**SUB1**

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB1VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Pish</td>
</tr>
</tbody>
</table>

**SUB2**

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB2VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>677</td>
</tr>
</tbody>
</table>

**SUB22**

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB22VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Flim</td>
</tr>
</tbody>
</table>

---

**Need Type Identifier!**
Lots of strategies exist with various performance impacts...

<table>
<thead>
<tr>
<th>ID</th>
<th>VAR</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Foo</td>
<td>S1</td>
</tr>
<tr>
<td>13</td>
<td>Bar</td>
<td>S22</td>
</tr>
</tbody>
</table>

```java
if id < 10
    type = Sub1
else
    type = Sub22
```
Side Bar: Identifying Type

- Another option to "type identifier fields" exists → Just read through ALL Leaf classes and figure it out in memory... Performance?

```
<table>
<thead>
<tr>
<th>Root {a}</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: int</td>
</tr>
<tr>
<td>var: String</td>
</tr>
</tbody>
</table>

Sub1

<table>
<thead>
<tr>
<th>Sub2 {a}</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub2var: int</td>
</tr>
</tbody>
</table>

Sub22

ROOT

<table>
<thead>
<tr>
<th>ID</th>
<th>VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Foo</td>
</tr>
<tr>
<td>13</td>
<td>Bar</td>
</tr>
</tbody>
</table>

SUB1

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB1VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Pish</td>
</tr>
</tbody>
</table>

SUB2

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB2VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>677</td>
</tr>
</tbody>
</table>

SUB22

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB22VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Flim</td>
</tr>
</tbody>
</table>
"All Table" Issues

- Lots (and lots) of joining
- Type identification can be inefficient
- Performance Impact:
  - **PRO:** Root Level/Abstract Class query results possible
  - **CON:** Additional queries or joins required
  - **CON:** Type Identification
“Single Table Mapping”

Map to one table

```
<table>
<thead>
<tr>
<th>ID</th>
<th>VAR</th>
<th>SUB1VAR</th>
<th>SUB2VAR</th>
<th>SUB22VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Foo</td>
<td>Pish</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>13</td>
<td>Bar</td>
<td>null</td>
<td>677</td>
<td>Flim</td>
</tr>
</tbody>
</table>
```
Identifying Type

<table>
<thead>
<tr>
<th>Root {a}</th>
</tr>
</thead>
<tbody>
<tr>
<td>id: int</td>
</tr>
<tr>
<td>var: String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sub1</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub1var: String</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sub2 {a}</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub2var: int</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sub22</th>
</tr>
</thead>
<tbody>
<tr>
<td>sub22var: String</td>
</tr>
</tbody>
</table>

### ROOT

<table>
<thead>
<tr>
<th>ID</th>
<th>VAR</th>
<th>SUB1VAR</th>
<th>SUB2VAR</th>
<th>SUB22VAR</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Foo</td>
<td>Pish</td>
<td>null</td>
<td>null</td>
<td>S1</td>
</tr>
<tr>
<td>13</td>
<td>Bar</td>
<td>null</td>
<td>677</td>
<td>Flim</td>
<td>SUB22</td>
</tr>
</tbody>
</table>

Two options for type detection:

```python
if SUB1VAR != null
    type = Sub1
else
    type = Sub22
```
“Single Table” Issues

- DBA will freak out – de-normalized database
- Type identification can be inefficient
- Performance Impact:
  ➢ PRO:
    • Heterogeneous query results possible
    • No extra queries or joins for subclasses
  ➢ CON: Additional table size, unused columns
"Combination Table Mapping"

**Root {a}**
- id: int
- var: String

**Sub1**
- sub1var: String

**Sub2 {a}**
- sub2var: int

**ROOT**

<table>
<thead>
<tr>
<th>ID</th>
<th>VAR</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Foo</td>
<td>S1</td>
</tr>
<tr>
<td>13</td>
<td>Bar</td>
<td>S22</td>
</tr>
</tbody>
</table>

**SUB1**

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB1VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Pish</td>
</tr>
</tbody>
</table>

**SUB22**

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB22VAR</th>
<th>SUB2VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Flim</td>
<td>677</td>
</tr>
</tbody>
</table>
Which Strategy to Use?

- **Leaf Table**
  - When never querying at abstract class level

- **All Table**
  - When not doing much querying
    - Too many joins

- **Single Table**
  - When one subclass is much more prevalent, and your DBA doesn’t care about normalization

- **Combination**
  - Use to optimize above situations
Wouldn’t It Be Cool?

Map to view for reads...

ROOT (View)

<table>
<thead>
<tr>
<th>ID</th>
<th>VAR</th>
<th>SUB1VAR</th>
<th>SUB2VAR</th>
<th>SUB22VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Foo</td>
<td>Pish</td>
<td>null</td>
<td>null</td>
</tr>
<tr>
<td>13</td>
<td>Bar</td>
<td>null</td>
<td>677</td>
<td>Flim</td>
</tr>
</tbody>
</table>

Map to tables for writes...

ROOT

<table>
<thead>
<tr>
<th>ID</th>
<th>VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Foo</td>
</tr>
<tr>
<td>13</td>
<td>Bar</td>
</tr>
</tbody>
</table>

Sub1

<table>
<thead>
<tr>
<th>sub1var: int</th>
</tr>
</thead>
</table>

Sub2

<table>
<thead>
<tr>
<th>sub2var: int</th>
</tr>
</thead>
</table>

Sub22

<table>
<thead>
<tr>
<th>sub22var: String</th>
</tr>
</thead>
</table>

SUB2

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB2VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>677</td>
</tr>
</tbody>
</table>

SUB22

<table>
<thead>
<tr>
<th>ID</th>
<th>SUB22VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Flim</td>
</tr>
</tbody>
</table>
Agenda

Persistence Architecture Overview

Performance Points:
- Models and Mapping
- Inheritance
- **Locking**
- Querying
  - Lazy Reading
  - N+1 Reads
  - Projections
- Transactions
- Caching

Leveraging the Database and JDBC
Locking

- J2EE Developers want to think of locking at the object level
- Databases may need to manage locking across many applications
- Persistence layer or application server must be able to respect and participate in locks at database level
Optimistic Locking

- DBA may wish to use version, timestamp and/or last update field to represent optimistic lock
  - Java developer may not want this in their business model
  - Persistence layer must be able to abstract this
- Must be able to support using any fields including business domain
Optimistic Locking

SQL from Transaction A
UPDATE EMP SET SAL=50,000, VER=4 WHERE ID=7 AND VER=3

EMP

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>SAL</th>
<th>VER</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Don</td>
<td>40,000</td>
<td>3</td>
</tr>
<tr>
<td>13</td>
<td>Trisha</td>
<td>55,000</td>
<td>8</td>
</tr>
</tbody>
</table>

SQL from Transaction B
UPDATE EMP SET NAME='DONALD', VER=4 WHERE ID=7 AND VER=3

If A "Wins":
B gets "Opt Lock Exception"

<table>
<thead>
<tr>
<th>ID</th>
<th>NAME</th>
<th>SAL</th>
<th>VER</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Donald</td>
<td>40,000</td>
<td>4</td>
</tr>
<tr>
<td>13</td>
<td>Trisha</td>
<td>55,000</td>
<td>8</td>
</tr>
</tbody>
</table>

If B "Wins":
A gets "Opt Lock Exception"
Pessimistic Locking

- Requires careful attention as a JDBC connection is required for duration of pessimistic lock
- Should support SELECT FOR UPDATE [NOWAIT] semantics
Concurrent Protection: Locking

- Proper locking semantics is a must
- Recommendations:
  - Use optimistic locking
    - Reads can avoid unnecessary refresh by comparing lock values
  - Minimize pessimistic locking
    - Reduce database resource contention with longer transactions
    - Use when retry is more costly than locking
    - Typically used in data processing or non-user interactive modules
Agenda

Persistence Architecture Overview

Performance Points:
- Models and Mapping
- Inheritance
- Locking
- Querying
  - Lazy Reading
  - N+1 Reads
  - Projections
- Transactions
- Caching
- Leveraging the Database and JDBC
Querying

- Critical element of ORM tuning
  - Most applications have more reading than writing

- GOALS
  - Reducing the amount of SELECT statements
  - Optimize the speed of each SELECT statement

- RECOMMENDED SOLUTIONS
  - Just in time reading
  - Result caching – object or result container
  - Optimized SQL generation
  - Data projections versus full object loading
Agenda

Persistence Architecture Overview

Performance Points:
- Models and Mapping
- Inheritance
- Locking
- Querying
  - Lazy Reading
  - N+1 Reads
- Projections
- Transactions
- Caching
- Leveraging the Database and JDBC
Object Traversal – Lazy Reads

- J2EE Applications work on the scale of a few hundreds of megabytes
- Databases routinely manage gigabytes and terabytes of data
- Persistence layer must be able to transparently fetch data “just in time”
Classic Support Case

Customer – “Uh, Hi. I am trying to read 1 object and get 3,000 SQL statements.”

Support – “Have you turned on indirection?”

Customer – *silence*

Support – “Turn on Proxies and let us know what happens”

Customer – “Oh, 3 SQL statements now. Thanks 😊”
1. Accessing relationship for first time

2. Get related object based on FK

3a. Check Cache

3b. SQL if not cached

4. Plug result into Proxy
Lazy Loading – Just in Time Reading

- Use of proxy to defer reading until required
- Very valuable performance feature
- Several Implementation Options
  - Explicit proxy
  - Dynamic proxy (java.lang.reflect.Proxy)
  - Development time class enhancement (source or byte codes)
  - Dynamic weaving (AOP)
Agenda

Persistence Architecture Overview

Performance Points:

- Models and Mapping
- Inheritance
- Locking
- Querying
  - Lazy Reading
  - N+1 Reads
- Projections
- Transactions
- Caching
- Leveraging the Database and JDBC
N+1 Reads Problem

- Many persistence layers and application servers have an N+1 reads problem
- Causes N subsequent queries to fetch related data when a collection is queried for
- A side effect of the impedance mismatch and poor mapping and querying support in persistence layers
**N+1 Reads Problem**

1. `findByCity()`
2. Pool of Created Objects or Beans
3. Returns collection
4. For each Customer
   - Fetch their Address
4. If Address had related objects, they too may be fetched 2n+1 Reads!
5. Container returns results
6. Address

Diagram:
- findByCity() in Persistence Layer or EJB Container
- Pool of Created Objects or Beans
- Returns collection
- For each Customer
  - Fetch their Address
  - If Address had related objects, they too may be fetched 2n+1 Reads!
- Container returns results
Classic Support Case #2

Customer – “Uh, Hi. I am trying to read 1 object and get 3,000 SQL statements.”
Support – “Have you turned on indirection?”
Customer – “Yes, and it’s still at 3,000 SQLs.”
Support – “Turn on join/batch reading”
Customer - * SILENCE *
Customer – “Oh, 3 SQL statements now. Thanks 😊”
Query Example

Find all pending PO’s with a customer in San Francisco and display customer details

**EJB QL**

```
SELECT po FROM PurchaseOrder po
WHERE po.status = 'ACTIVE' AND
po.customer.address.city = 'SFO'
```
Query Example: Initial SQL

SELECT PO.* FROM PO, CUST, ADDR WHERE
  PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND
  ADDR.CITY = 'SFO'

  {Returns N Purchase Orders}

SELECT * FROM CUST WHERE CUST.ID = 1 ...  {N}
SELECT * FROM ADDR WHERE ADDR.ID = 100 ...  {N}
SELECT * FROM PHONE WHERE PHONE.CUST_ID = 1 ...  {N}

RESULT: 3N+1 queries (100 PO’s = 301)

“N+1 Query – explosion of SQL”
N+1 Reads

- Must have solution to minimize queries
- Need flexibility to reduce to 1 query, 1+1 query or N+1 query where appropriate
  - 1 Query when displaying list of customers and addresses – known as a “Join Read”
  - 1+1 Query when displaying list of customers and user may click button to see addresses – known as a “Batch Read”
  - N+1 Query when displaying list of customers but only want to see address for selected customer
Where Do I Define This?

- Many options exist, depends on persistence layer and spec
  - In the query language syntax
  - On the mapping
  - Development time/runtime
Join Reading 1:1’s

- Join Reading
  - Join rows together in a single SELECT
- Example of generated SQL

```sql
SELECT PO.*, CUST.*, ADDR.* FROM PO, CUST, ADDR WHERE PO.STATUS = ‘ACTIVE’ AND PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = ‘SFO’
{Returns N Purchase Orders with Customers & Addresses}

SELECT * FROM PHONE WHERE PHONE.CUST_ID = 1
...{N calls}

RESULT: N+1 queries (100 Phone #’s = 101)
Join Reading All Relationships

- **Join Reading**
  - Read in rows that can be joined together in a single SELECT

- **Example**

```sql
SELECT PO.*, CUST.*, ADDR.*, PHONE.*
FROM PO, CUST, ADDR, PHONE
WHERE PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO' AND PHONE.CUST_ID = CUST.ID

{Returns N Purchase Orders with Customers, Addresses, and PhoneNumbers}
```

RESULT: 1 query (100 PO’s)
Batch and Join Reading

- Use SELECT per child with join to original clause
- Example

```
SELECT PO.*, CUST.*, ADDR.* FROM PO, CUST, ADDR WHERE
  PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND
  CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO'
{Returns N Purchase Orders with Customers & Addresses}

SELECT PHONE.* FROM PHONE, PO, CUST, ADDR WHERE
  PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND
  CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO' AND
  PHONE.CUST_ID = CUST.ID
```

RESULT: 2 queries (100 PO’s)
Joining + Batch Reading

- Use SELECT per child with join to original clause

Example

```sql
SELECT PO.* FROM PO, CUST, ADDR WHERE PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO'
```

```sql
SELECT CUST.* FROM PO, CUST, ADDR WHERE PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO'
```

```sql
SELECT ADDR.* FROM PO, CUST, ADDR WHERE PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO'
```

```sql
SELECT PHONE.* FROM PHONE, PO, CUST, ADDR WHERE PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO' AND PHONE.CUST_ID = CUST.ID
```

RESULT: 100 PO’s = 4 SQL
Join & Batch Reading

- Reducing the quantity of SQL calls will produce a significant performance gain
- Joining and batching can be combined
- Be careful not to extend the depth too deep
  - The cost of the join could become too expensive
# Query Optimization Options

## Reading 100 PO’s with relationships

<table>
<thead>
<tr>
<th>Option</th>
<th># SQL</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>301</td>
<td>200 ms</td>
</tr>
<tr>
<td>Join 1:1’s</td>
<td>101</td>
<td>150 ms</td>
</tr>
<tr>
<td>Join All</td>
<td>1</td>
<td>60 ms</td>
</tr>
<tr>
<td>Batch All</td>
<td>4</td>
<td>40 ms</td>
</tr>
<tr>
<td>Batch &amp; Join</td>
<td>2</td>
<td>20 ms</td>
</tr>
</tbody>
</table>

Source: Simple test environment

Fewer SQL ! Always = Faster Execution
Agenda

Persistence Architecture Overview

Performance Points:
- Models and Mapping
- Inheritance
- Locking
- Querying
  - Lazy Reading
  - N+1 Reads
  - Projections
- Transactions
- Caching

Leveraging the Database and JDBC
Queries: Minimize Data Read

- Only read the entire object when required
  - When modifying the object
  - When caching will assist concurrent clients
  - All fields of the object are required

- Consider projections versus object queries
Query: Projection Example

**EJB QL:**
```
SELECT new PODetails(po.id, po.num, po.amount, c.lastName, a.city)
FROM PurchaseOrder po JOIN po.customer c,
    JOIN po.customer.address a
WHERE po.status = 'ACTIVE' AND a.city = 'SFO'
```

**SQL:**
```
SELECT PO.ID, PO.NUM, PO.AMOUNT, CUST.L_NAME, ADDR.CITY FROM PO, CUST, ADDR WHERE PO.STATUS = 'ACTIVE' AND PO.CUST_ID = CUST.ID AND CUST.ADDR_ID = ADDR.ID AND ADDR.CITY = 'SFO'
```

<table>
<thead>
<tr>
<th>ID</th>
<th>PO #</th>
<th>$</th>
<th>Last</th>
<th>City</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AB123</td>
<td>100.00</td>
<td>Smith</td>
<td>Ottawa</td>
</tr>
<tr>
<td>2</td>
<td>CD456</td>
<td>500.00</td>
<td>Jones</td>
<td>New York</td>
</tr>
</tbody>
</table>
Object Traversals

- How would a developer using a typical persistence layer find “The average order amount for customers in city X”?
- In other words, how do Java developers write “report” style queries?

```
Customer
  name
  city
  orders

Order
  orderDate
  cost
  orderItems

OrderItem
  quantity
  discount
  product

Product
  productName
  cost
  weight
```
Example Code

```java
public double findAverageOrderSizeFor(String city) {
    int count=0; double value=0.0;
    Customer c = new Customer();
    c.setCity(city);
    Collection custs = pLayer.executeQueryByExample(c);
    iterate i over custs {
        Collection ords = custs[i].getOrders();
        iterate j over ords {
            value += ords[j].getValue(); count++;
        }
    }
    return value/count;
}
```

This approach causes unnecessary interactions with the database and brings too much data into the appserver…
A Better Solution

**EJB QL:**

\[
\text{SELECT } \text{avg(ord.amount)} \\
\text{FROM Customer cu JOIN cu.order o,} \\
\text{WHERE cu.city = :city}
\]

Generates

\[
\text{SELECT AVG(t0.VALUE) FROM ORDERS t0, ADDRESS t2, CUSTOMER t1} \\
\text{WHERE ((t2.NAME = city) AND ((t0.CUST_ID = t1.ID) AND (t2.ID = t1.ADD_ID)))}
\]

Projections allow Java developers to leverage the database instead of trying to do all the computation in JVM.
Query: Projection Optimizations

- Specify the required fields in terms of the object model
- Use relationships to indicate joins
- Use aggregate functions
  - SUM, MIN, MAX, COUNT, AVERAGE
- Produce easy to render results
Agenda

Persistence Architecture Overview

Performance Points:
- Models and Mapping
- Inheritance
- Locking
- Querying
  - Lazy Reading
  - N+1 Reads
  - Projections
- Transactions
- Caching

Leveraging the Database and JDBC
Transactions: Short & Sweet

- J2EE apps typically support many clients sharing small number of db connections
- Ideally would like to minimize length of transaction on database
Transactions: Minimal Writes

- Persistence layer tracks/calculates changes
- Only UPDATE modified columns
- Order UPDATE, INSERT, & DELETEs using referential integrity rules and consistent order

Benefits
- Simplified development
- Reduced TX costs
- Minimize deadlock occurrences
Transactions: Minimize Context

- Each object in a transaction context has a cost
- Only include objects which may be changed in the transaction context
- Symptoms
  - Long commit cycles
  - Increased garbage creation
Minimize Context

- **Bad:**

  ```
  begin UOW
  read 10,000 objects
  pick 1
  change 1
  commit UOW
  ```

- **Better:**

  ```
  read 10,000 objects
  pick 1
  begin UOW
  change 1
  commit UOW
  ```
Transactions: Other Optimizations

- Data Access Optimizations
  - Parameter Binding
  - ResultSet streaming
  - Streaming of large objects (BLOB/CLOB)

- Cache Interaction
  - Transaction updates to shared object cache
  - Distribution of changes to clustered caches
Agenda

Persistence Architecture Overview

Performance Points:
- Models and Mapping
- Inheritance
- Locking
- Querying
  - Lazy Reading
  - N+1 Reads
  - Projections
- Transactions
- Caching

Leveraging the Database and JDBC
Why Have a Cache?

- Object Identity
- Performance – Avoid extra database trips
  - Reference Read-Only
  - Read-mostly
Caching

**OO Query**

- **NO** – Build bean/object from results

**SQL Query**

- **Does PK for row exist in cache?**

**Results(s)**

- **YES** – Get from Cache

- **NO** – Build bean/object from results

**Return object results**
ORM Caching Options

- Isolated cache
  - Per Client/Thread
- Transactional
  - Conforming
- Shared cache
Caching Architecture

- Distributed/Coordinated
- Configurable/Flexible

**Session**
- Isolated Cache

**UnitOfWork**
- TX Cache

**Server**
- Shared Cache

**Session Bus**

**JDBC/JCA**

**Server**
- Shared Cache

**JMS (MDB)**
- RMI
- CORBA
- IIOP
Cache Configuration

- Categorize your persistent entity types
  - Isolated: short cache life
  - Shared: medium to long cache life (app/db specific)
  - Reference: Long cache life

- Configure each type
  - Cache type
  - Initial and/or maximum size
  - Invalidation (time-to-live)
Caching –

Stale Cache Management

- Non-locking read = stale data
- Trying to eliminate this issue can lead to major performance penalties
- Addressing stale cache
  - Configurable cache life-cycle
  - Refresh using query when required
  - Use optimistic lock value to avoid extra refresh
  - Avoid turning off the cache
Distributed/Coordinated Caching

- Following successful database commit
  - Changes are merged into shared cache
  - Changes messaged to clustered sessions

- Ensure messaging is warranted
  - Fail-over
  - Shared objects between users
Agenda

Persistence Architecture Overview

Performance Points:
- Models and Mapping
- Inheritance
- Locking
- Querying
  - Lazy Reading
  - N+1 Reads
  - Projections
- Transactions
- Caching

Leveraging the Database and JDBC
Leverage the Relational Database

- Schema adjustments
- Indexes
- Order of calls & columns
- Stored Procedures
- PK Sequencing (pre-allocation)
- Hints in SQL
Leverage JDBC Features

- Prepared Statement Caching
- Parameter binding
- Large data element streaming
- Result scrolling
Work with Your DBA

- Some ORM approaches ignore RDBMS best-practices
- Don’t restrict the evolution of the relational schema
- Allow for the use of indices and stored procedure where necessary
- Leverage advanced features of the database
  - Hints in SQL
  - Custom types (ORM data types)
  - XML storage
Summary

- Know or learn your application data and usage
- Know your ORM persistence layer
- Leverage the database & JDBC capabilities
- General Performance Tuning
Q&A