Java is a trademark of Sun Microsystems, Inc.

Java

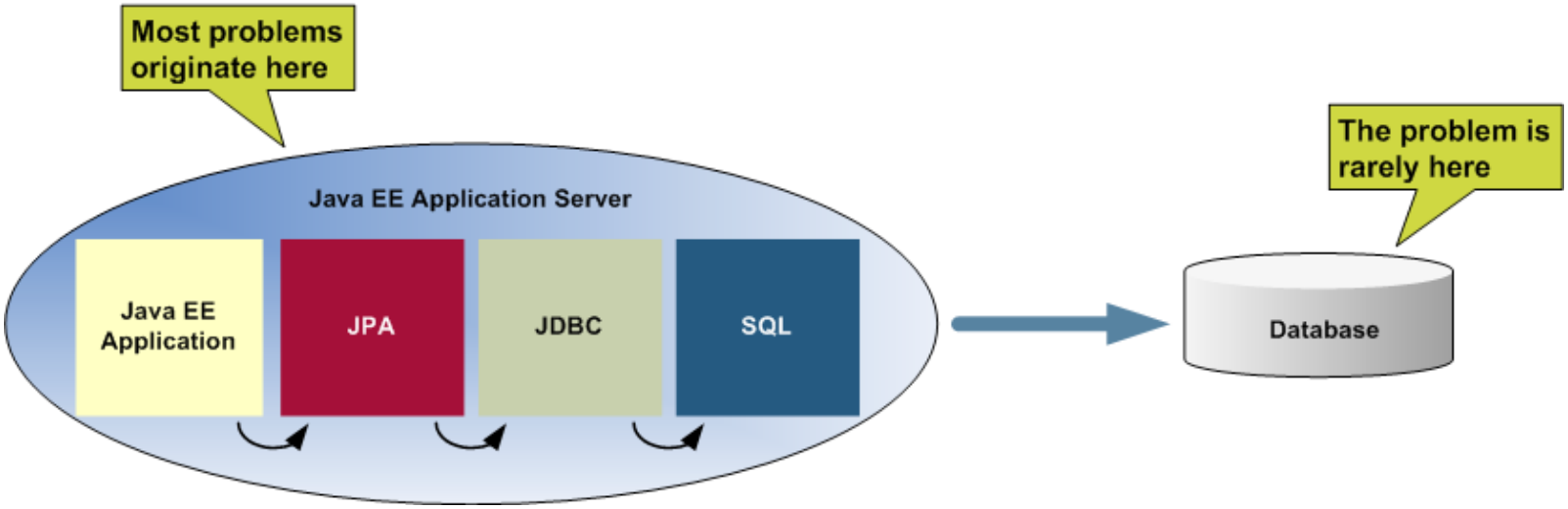
JavaOne℠

Keeping a Relational Perspective for Optimizing JPA

Debu Panda
Principal Product Manager, Oracle
Reza Rahman
Independent Consultant

JPA and Database Harmony

- JPA increases productivity by abstracting away low-level data access code
- Not thinking critically about how JPA affects the database leads to performance problems



Roadmap

- ORM mapping and schema refactoring
- Optimizing generated SQL
- Keeping JDBC in mind
- Reducing database access by caching
- Essential tools for tuning

ORM Mapping and Schema Refactoring
Denormalizing

- Normalization is over-emphasized
- Denormalization can significantly improve performance
- Be careful, denormalization does not come free
- Overdoing it can lead to maintenance problems and abuse disk-usage

ORM Mapping and Schema Refactoring

Denormalization Examples

```
@Entity
public class Item {
    ...
    @Column(name="DOMESTIC_SHIPPING_RATE")
    protected double domesticShippingRate;
    @Column(name="INTERNATIONAL_SHIPPING_RATE")
    protected double internationalShippingRate;
    ...
    @Column(name="NUMBER_OF_BIDS")
    protected long numberOfBids;
    ...
    @Embedded
    protected ItemDetails details;

@Embeddable
public class ItemDetails {
    @Column(name="MANUFACTURER")
    protected String manufacturer;
    @Column(name="YEAR_BUILT")
    protected Date yearBuilt;
```

In-line columns Instead of @OnetoMany with normalized SHIPPING_RATES table

Column with cached value instead of querying related Bid entities often

Embedded object instead of @OneToOne with ItemDetails entity

ORM Mapping and Schema Refactoring

Database Indexes

- Primary keys are automatically indexed
- Index frequently used foreign keys in larger tables
- Index any columns heavily used in joins
- Index columns often used as query parameters
- Index columns used in ordering
- Be careful, indexing is not free

ORM Mapping and Schema Refactoring

Indexing Candidates

```
@Entity
public class Bid {
    ...
    @ManyToOne
    @JoinColumn(name="BID_ITEM_ID",
        referencedColumnName="ITEM_ID")
    protected Item item;
    ...
}

Query query = entityManager.createQuery(
    "SELECT s " +
    "FROM Seller s, Buyer b " +
    "WHERE s.location = b.location " +
    "AND b.firstName = :firstName");
```

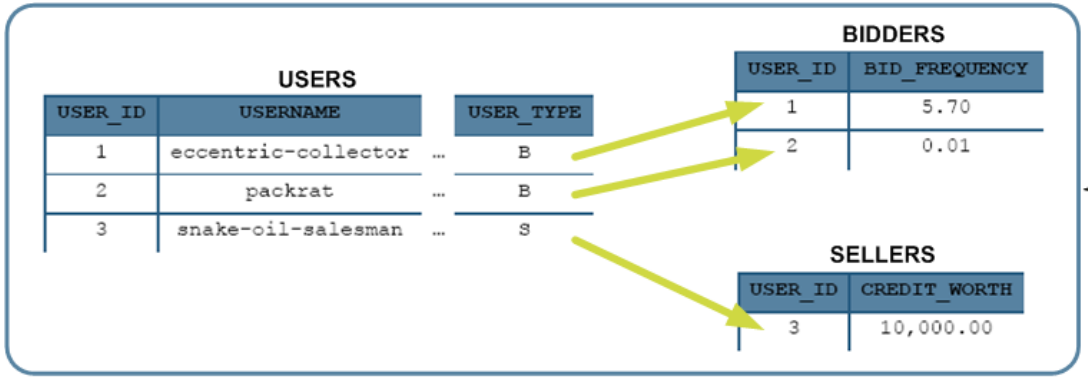
The BID_ITEM_ID foreign key should be indexed

The location columns should be indexed, although they are not foreign keys

The first name column should be indexed

ORM Mapping and Schema Refactoring

Inheritance Strategies



Join table strategy is normalized, but has worst performance

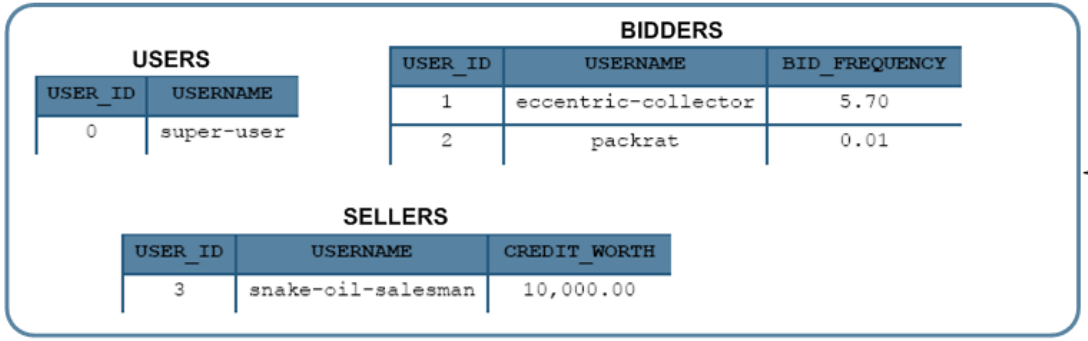
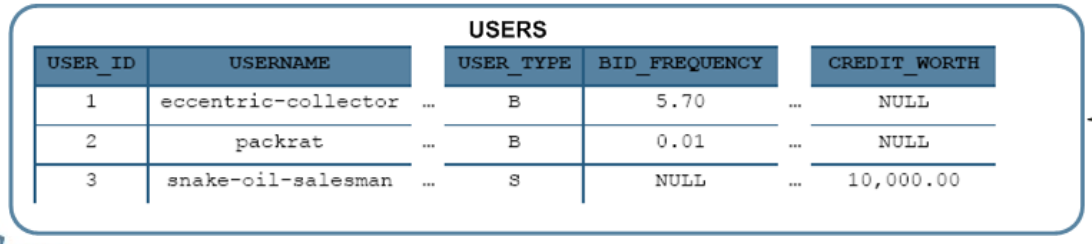


Table per class strategy avoids joins but causes unions



Single table strategy is most performance friendly, but is denormalized

ORM Mapping and Schema Refactoring
Cascading Deletes

- Consider cascading deletes on the database

```
@Entity
@DiscriminatorValue(value = "S")
public class Seller extends User {
    ...
    @OneToOne(cascade=CascadeType.REMOVE)
    protected BillingInfo billingInfo;

CREATE TABLE billing_info (
    billing_id NUMBER(10) NOT NULL,
    ...
    CONSTRAINT fk_user FOREIGN KEY
  (billing_user_id)
    REFERENCES user (user_id)
    ON DELETE CASCADE
)
```

Generates extra SQL

Much faster alternative

Optimizing Generated SQL
Lazy Loading

- Lazy load fields and relationships not used often
- One-many/many-many relationships lazy loaded by default
- Carefully consider lazy loading one-one and many-one relationships
- Lazy loading CLOB/BLOB fields is usually a good idea
- Eagerly load relationships and data that are often used

Optimizing Generated SQL

Lazy Loading Example

```
@Entity
@Table(name="USERS")
@SecondaryTable(name="USER_PICTURES",
    pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))
public class User {
    ...
    @Column(name="USER_NAME")
    protected String username;
    ...
    @Column(name="PICTURE", table="USER_PICTURES")
    @Lob
    @Basic(fetch=FetchType.LAZY)
    protected byte[] picture;
    ...
    @OneToOne(fetch=FetchType.LAZY)
    protected ContactInfo contactInfo;
```

Infrequently used column put into secondary table and lazily loaded

Infrequently used relationship loaded lazily

Optimizing Generated SQL
Join Fetches

- Join fetch used to override relationships configured as lazy loaded but should be loaded eagerly in certain use cases
- Avoids the N+1 SELECTs problem

```
Query query = entityManager.createQuery(
    "SELECT i " +
    "FROM Item i FETCH JOIN i.category " +
    "WHERE i.itemId = :itemId");
query.setParameter("itemId", itemId);
Item item = (Item)
  query.getSingleResult();
```

Lazy loading of category being overridden at runtime

Optimizing Generated SQL
Bulk Updates and Deletes

- Bulk updates and deletes reduce number of queries issued

```
Query query = entityManager.createQuery(
    "SELECT i FROM Item i " +
    "WHERE i.bidEndDate < :currentDate");
query.setParameter("currentDate", now);
for (Item item : query.getResultList()) {
    item.setStatus("Closed");
}
```

Generates many SQL statements

```
Query query = entityManager.createQuery(
    "UPDATE Item i SET i.status = 'Closed' " +
    "WHERE i.bidEndDate < :currentDate");
query.setParameter("currentDate", now);
query.executeUpdate();
```

Generates one SQL statement

Optimizing Generated SQL

Native Queries

- Use native queries to take advantage of database specific optimization features or very fine tuning
- Hints are an obvious optimization case

```
Query query = entityManager.createNativeQuery(
    "SELECT /*+ FULL(u) */ * " +
    "FROM users u " +
    "WHERE status = 'Active'",
    User.class);

List<User> users = (List<User>) query.getResultList();
```

Most users are 'Active', so using the index is expensive and a table scan is faster

Keeping JDBC in Mind

Named Queries

- Named queries reduce parsing overhead
- Enforce best practice of parameterized queries
- Use query parameters even for dynamic queries
- Use the database connection pool's prepared statement cache or database query caching

```
@NamedQuery(
    name="User.getBySsn",
    query="SELECT u FROM User u WHERE u.ssn = :ssn")

Query query =
    entityManager.createNamedQuery("User.getBySsn");
query.setParameter("ssn", ssn);
User user = (User) query.getSingleResult();
```

Keeping JDBC in Mind

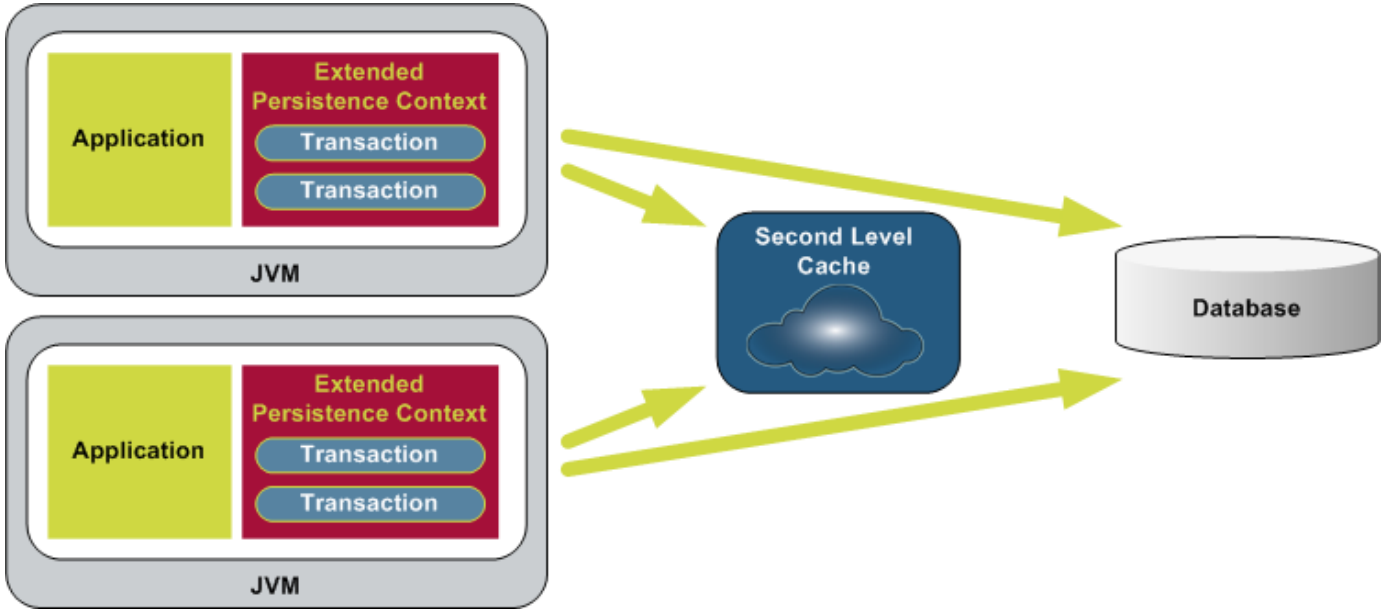
Use Transactions Properly

- Eliminate transaction overhead for “read-only” queries

```
@TransactionAttribute(NOT_SUPPORTED)
public List<Category> getAllCategories() {
    return (List<Category>) entityManager.createQuery(
        "SELECT c FROM Category c").getResultList();
}
```

Caching to Reduce Database Load

- Transactional cache
- Extended persistence context
- Second Level Cache
- Second level cache can be clustered



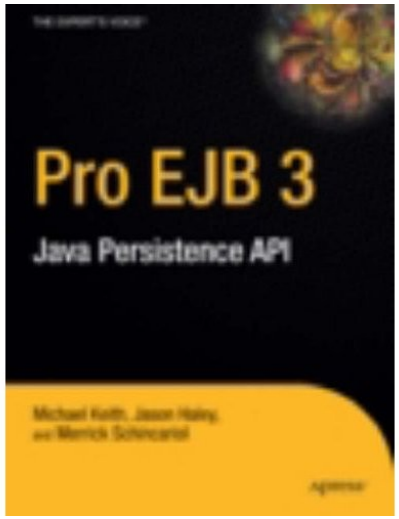
Essential Tools for Tuning

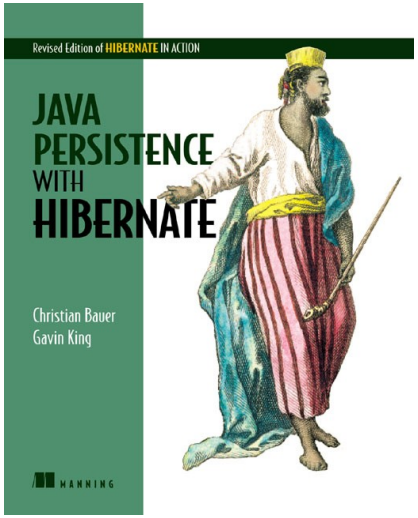
- Persistence provider tools
  - hibernate.show_sql, hibernate.format_sql, hibernate.use_sql_comments, toplink.logging.*
  - JMX integration
- Database monitoring tools
  - Oracle Enterpriser Manager, SQL Profiler, MySQL Enterprise Monitor
- Query analysis tools
  - Oracle explain plan, SQL Query Analyzer, MySQL Query Analyzer
- Java Diagnostics Tools
  - Oracle AD4J, Wily Introscope

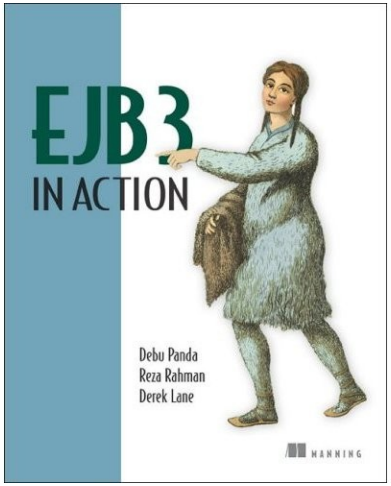
Summary

- JPA is not a substitute for thinking critically about the database
- Many factors to keep in mind such as domain modeling, database refactoring, ORM mapping, query tuning, caching and monitoring
- Solid grasp of SQL and working closely with an experienced DBA, in addition to having a deep understanding of JPA is key
- Tuning is a natural and essential part of the application life-cycle

References







Shameless plug alert!

Thank You

Debu Panda
debabrata.panda@oracle.com
Reza Rahman
reza@rahmannet.net