Oracle Blog

Carol McDonald

Java Stammtisch

« JPA Caching | Main | Some Concurrency... »

JPA Performance, Don't Ignore the Database

By caroljmcdonald on Aug 28, 2009

Database Schema

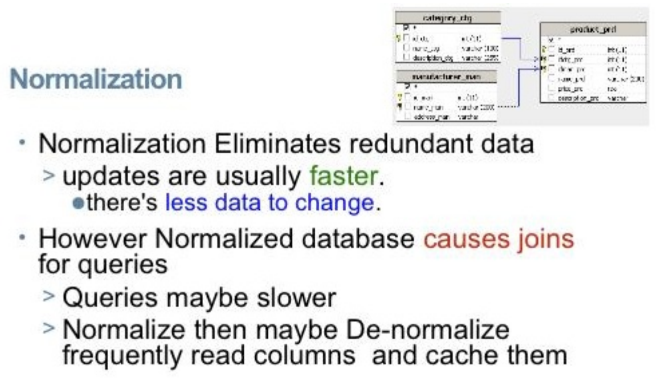
Good Database schema design is important for performance. One of the most basic optimizations is to design your tables to take as little space on the disk as possible , this makes disk reads faster and uses less memory for query processing.

Data Types

You should use the smallest data types possible, especially for indexed fields. The smaller your data types, the more indexes (and data) can fit into a block of memory, the faster your queries will be.

Normalization

Database Normalization eliminates redundant data, which usually makes updates faster since there is less data to change. However a Normalized schema causes joins for queries, which makes queries slower, denormalization speeds retrieval. More normalized schemas are better for applications involving many transactions, less normalized are better for reporting types of applications. You should normalize your schema first, then de-normalize later. Applications often need to mix the approaches, for example use a partially normalized schema, and duplicate, or cache, selected columns from one table in another table. With JPA O/R mapping you can use the @Embedded annotation for denormalized columns to specify a persistent field whose @Embeddable type can be stored as an intrinsic part of the owning entity and share the identity of the entity.

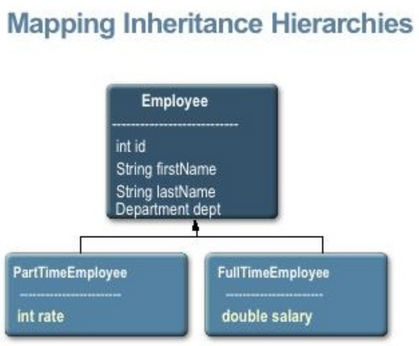


Normalization

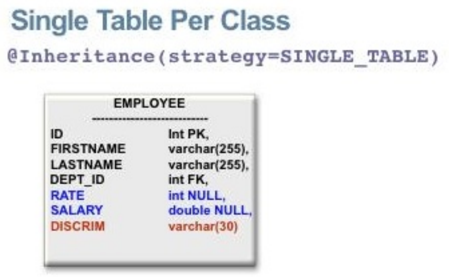
- Normalization Eliminates redundant data
  - > updates are usually faster.
    - there's less data to change.
- However Normalized database causes joins for queries
  - > Queries maybe slower
  - > Normalize then maybe De-normalize frequently read columns and cache them

Database Normalization and Mapping Inheritance Hiearchies

The Class Inheritance hierarchy shown below will be used as an example of JPA O/R mapping.



In the Single table per class mapping shown below, all classes in the hierarchy are mapped to a single table in the database. This table has a discriminator column (mapped by **@DiscriminatorColumn)**, which identifies the subclass. Advantages: This is fast for querying, no joins are required. Disadvantages: wastage of space since all inherited fields are in every row, a deep inheritance hierarchy will result in wide tables with many, some empty columns.

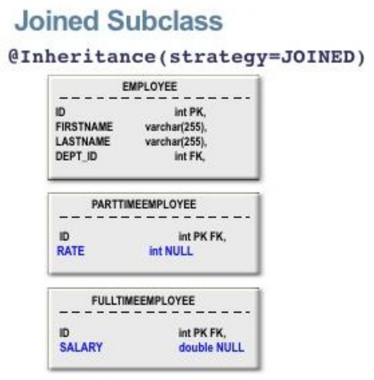


Single Table Per Class

```
@Inheritance(strategy=SINGLE_TABLE)
```

EMPLOYEE	
ID	Int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
RATE	int NULL,
SALARY	double NULL,
DISCRIM	varchar(30)

In the Joined Subclass mapping shown below, the root of the class hierarchy is represented by a single table, and each subclass has a separate table that only contains those fields specific to that subclass. This is normalized (eliminates redundant data) which is better for storage and updates. However queries cause joins which makes queries slower especially for deep hierachies, polymorphic queries and relationships.



Joined Subclass

```
@Inheritance(strategy=JOINED)
```

EMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,

PARTTIMEEMPLOYEE	
ID	int PK FK,
RATE	int NULL

FULLTIMEEMPLOYEE	
ID	int PK FK,
SALARY	double NULL

In the Table per Class mapping (in JPA 2.0, optional in JPA 1.0), every **concrete** class is mapped to a table in the database and all the inherited state is repeated in that table. This is not normlalized, inherited data is repeated which wastes space. Queries for Entities of the same type are fast, however polymorphic queries cause unions which are slower.

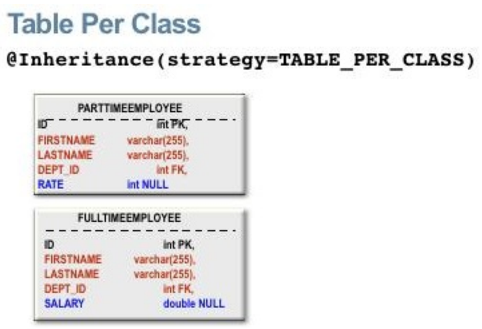


Table Per Class

```
@Inheritance(strategy=TABLE_PER_CLASS)
```

PARTTIMEEMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
RATE	int NULL

FULLTIMEEMPLOYEE	
ID	int PK,
FIRSTNAME	varchar(255),
LASTNAME	varchar(255),
DEPT_ID	int FK,
SALARY	double NULL

Know what SQL is executed

You need to understand the SQL queries your application makes and evaluate their performance. Its a good idea to enable SQL logging, then go through a use case scenario to check the executed SQL. Logging is not part of the JPA specification, With EclipseLink you can enable logging of SQL by setting the following property in the persistence.xml file:

```
<properties>
   <property name="eclipselink.logging.level" value="FINE"/>
</properties>
```

With Hibernate you set the following property in the persistence.xml file:

```
<properties>
   <property name="hibernate.show_sql" value="true" />
</properties>
```

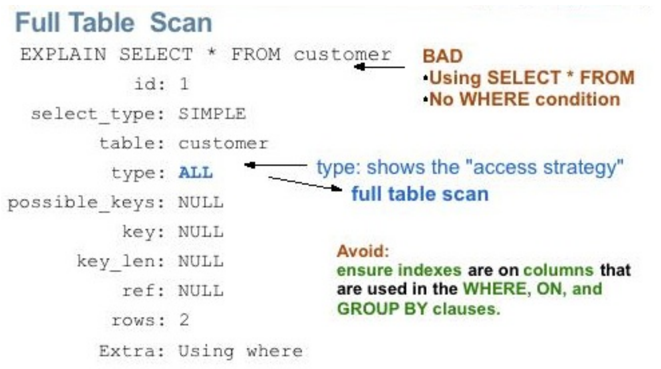
Basically you want to make your queries access less data, is your application retrieving more data than it needs, are queries accessing too many rows or columns? Is the database query analyzing more rows than it needs? Watch out for the following:

- queries which execute too often to retrieve needed data
- retrieving more data than needed
- queries which are too slow
  - you can use EXPLAIN to see where you should add indexes

With MySQL you can use the slow query log to see which queries are executing slowly, or you can use the MySQL query analyzer to see slow queries, query execution counts, and results of EXPLAIN statements.

Understanding EXPLAIN

For slow queries, you can precede a SELECT statement with the keyword EXPLAIN to get information about the query execution plan, which explains how it would process the SELECT, including information about how tables are joined and in which order. This helps find missing indexes early in the development process.



Full Table Scan

```
EXPLAIN SELECT * FROM customer
           id: 1
  select_type: SIMPLE
        table: customer
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 2
        Extra: Using where
```

BAD
•Using SELECT * FROM
•No WHERE condition

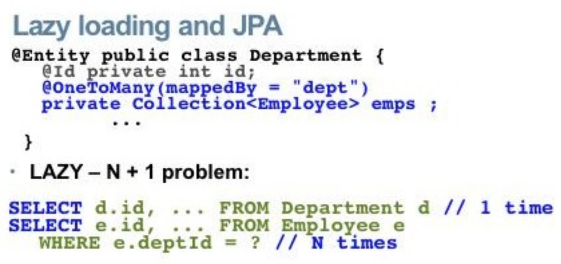
type: shows the "access strategy"
full table scan

Avoid:
ensure indexes are on columns that are used in the WHERE, ON, and GROUP BY clauses.

You should index columns that are frequently used in Query WHERE, GROUP BY clauses, and columns frequently used in joins, but be aware that indexes can slow down inserts and updates.

Lazy Loading and JPA

With JPA many-to-one and many-to-many relationships lazy load by default, meaning they will be loaded when the entity in the relationship is accessed. Lazy loading is usually good, but if you need to access all of the "many" objects in a relationship, it will cause n+1 selects where n is the number of "many" objects.



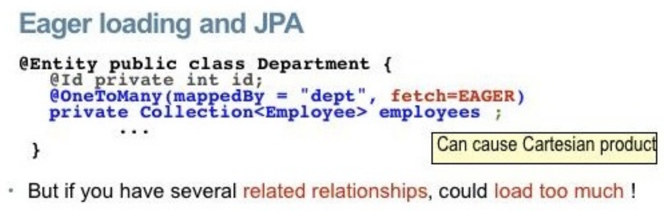
Lazy loading and JPA

```
@Entity public class Department {
   @Id private int id;
   @OneToMany(mappedBy = "dept")
   private Collection<Employee> emps ;
         ...
}
```

- LAZY – N + 1 problem:

```
SELECT d.id, ... FROM Department d // 1 time
SELECT e.id, ... FROM Employee e
   WHERE e.deptId = ? // N times
```

You can change the relationship to be loaded eagerly as follows :



Eager loading and JPA

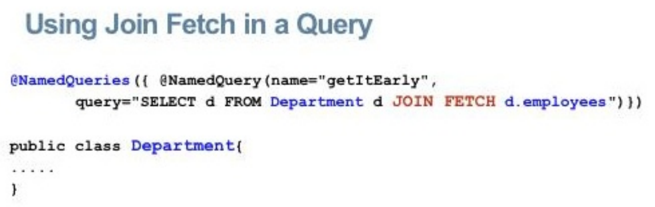
```
@Entity public class Department {
   @Id private int id;
   @OneToMany(mappedBy = "dept", fetch=EAGER)
   private Collection<Employee> employees ;
         ...
}
```

Can cause Cartesian product

- But if you have several related relationships, could load too much !

However you should be careful with eager loading which could cause SELECT statements that fetch too much data. It can cause a Cartesian product if you eagerly load entities with several related collections.

If you want to override the LAZY fetch type for specific use cases, you can use Fetch Join. For example this query would eagerly load the employee addresses:



Using Join Fetch in a Query

```
@NamedQueries({ @NamedQuery(name="getItEarly",
        query="SELECT d FROM Department d JOIN FETCH d.employees")})

public class Department{
.....
}
```

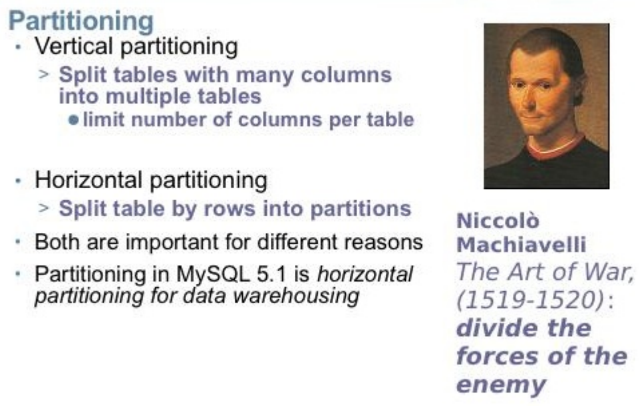
In General you should lazily load relationships, test your use case scenarios, check the SQL log, and use @NameQueries with JOIN FETCH to eagerly load when needed.

Partitioning

the main goal of partitioning is to reduce the amount of data read for particular SQL operations so that the overall response time is reduced

Vertical Partitioning splits tables with many columns into multiple tables with fewer columns, so that only certain columns are included in a particular dataset, with each partition including all rows.

Horizontal Partitioning segments table rows so that distinct groups of physical row-based datasets are formed. All columns defined to a table are found in each set of partitions. An example of horizontal partitioning might be a table that contains historical data being partitioned by date.



Partitioning

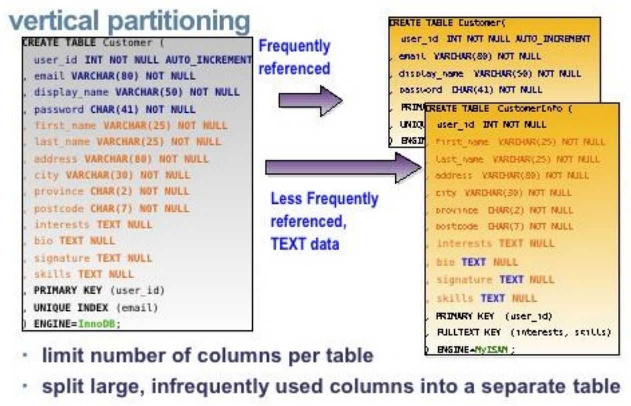
- Vertical partitioning
  - > Split tables with many columns into multiple tables
    - limit number of columns per table
- Horizontal partitioning
  - > Split table by rows into partitions
- Both are important for different reasons
- Partitioning in MySQL 5.1 is *horizontal partitioning for data warehousing*

Niccolò Machiavelli
The Art of War, (1519-1520): divide the forces of the enemy

Vertical Partitioning

In the example of vertical partitioning below a table that contains a number of very wide text or BLOB columns that aren't referenced often is split into two tables with the most referenced columns in one table and the seldom-referenced text or BLOB columns in another.

By removing the large data columns from the table, you get a faster query response time for the more frequently accessed Customer data. Wide tables can slow down queries, so you should always ensure that all columns defined to a table are actually needed.



vertical partitioning

```
CREATE TABLE Customer (
  user_id INT NOT NULL AUTO_INCREMENT
, email VARCHAR(80) NOT NULL
, display_name VARCHAR(50) NOT NULL
, password CHAR(41) NOT NULL
, first_name VARCHAR(25) NOT NULL
, last_name VARCHAR(25) NOT NULL
, address VARCHAR(80) NOT NULL
, city VARCHAR(30) NOT NULL
, province CHAR(2) NOT NULL
, postcode CHAR(7) NOT NULL
, interests TEXT NULL
, bio TEXT NULL
, signature TEXT NULL
, skills TEXT NULL
, PRIMARY KEY (user_id)
, UNIQUE INDEX (email)
) ENGINE=InnoDB;
```

Frequently referenced

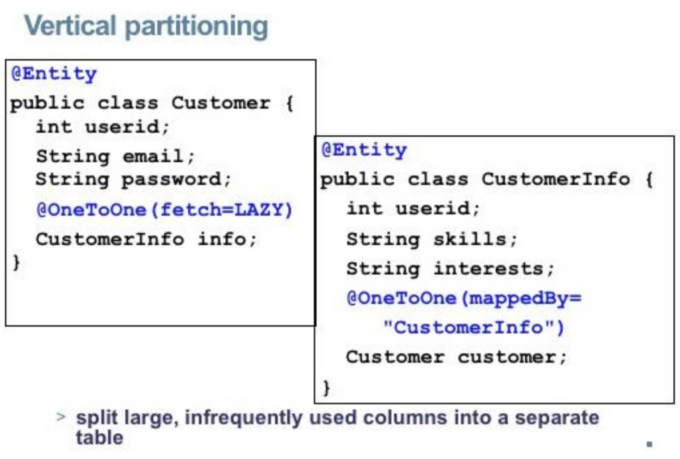
```
CREATE TABLE Customer(
  user_id INT NOT NULL AUTO_INCREMENT
, email VARCHAR(80) NOT NULL
, display_name VARCHAR(50) NOT NULL
, password CHAR(41) NOT NULL
, PRIM
, UNIQ
) ENGI
```

Less Frequently referenced, TEXT data

```
CREATE TABLE CustomerInfo (
  user_id INT NOT NULL
, first_name VARCHAR(25) NOT NULL
, last_name VARCHAR(25) NOT NULL
, address VARCHAR(80) NOT NULL
, city VARCHAR(30) NOT NULL
, province CHAR(2) NOT NULL
, postcode CHAR(7) NOT NULL
, interests TEXT NULL
, bio TEXT NULL
, signature TEXT NULL
, skills TEXT NULL
, PRIMARY KEY (user_id)
, FULLTEXT KEY (interests, skills)
) ENGINE=MyISAM;
```

- limit number of columns per table
- split large, infrequently used columns into a separate table

The example below shows the JPA mapping for the tables above. The Customer data table with the more frequently accessed and smaller data types is mapped to the Customer Entity, the CustomerInfo table with the less frequently accessed and larger data types is mapped to the CustomerInfo Entity with a lazily loaded one to one relationship to the Customer.



Vertical partitioning

```
@Entity
public class Customer {
  int userid;
  String email;
  String password;
  @OneToOne(fetch=LAZY)
  CustomerInfo info;
}
```

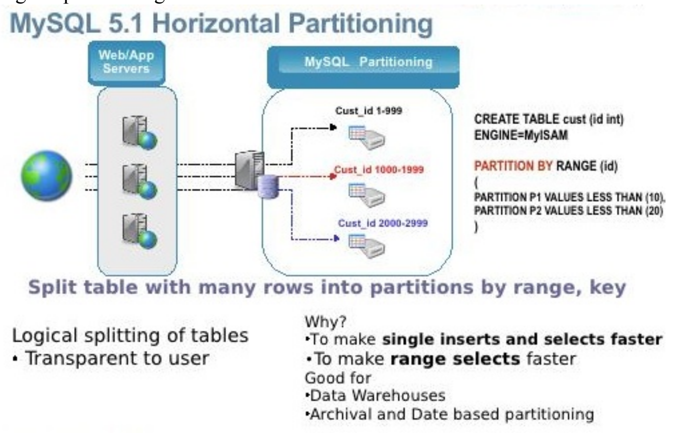
```
@Entity
public class CustomerInfo {
  int userid;
  String skills;
  String interests;
  @OneToOne(mappedBy=
     "CustomerInfo")
  Customer customer;
}
```

> split large, infrequently used columns into a separate table

Horizontal Partitioning

The major forms of horizontal partitioning are by Range, Hash, Hash Key, List, and Composite.

Horizontal partitioning can make queries faster because the query optimizer knows what partitions contain the data that will satisfy a particular query and will access only those necessary partitions during query execution. Horizontal Partitioning works best for large database Applications that contain a lot of query activity that targets specific ranges of database tables.



```
CREATE TABLE cust (id int)
ENGINE=MyISAM

PARTITION BY RANGE (id)
(
PARTITION P1 VALUES LESS THAN (10),
PARTITION P2 VALUES LESS THAN (20)
)
```

Split table with many rows into partitions by range, key

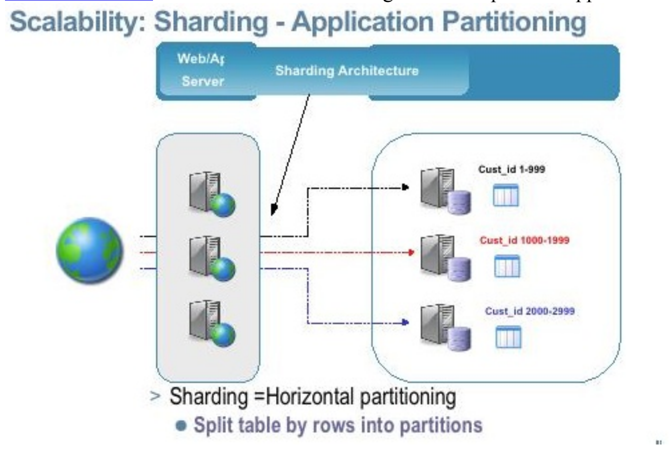
Logical splitting of tables
• Transparent to user

Why?
•To make **single inserts and selects faster**
•To make **range selects** faster
Good for
•Data Warehouses
•Archival and Date based partitioning

Hibernate Shards

Partitioning data horizontally into "Shards" is used by google, linkedin, and others to give extreme scalability for very large amounts of data. eBay "shards" data horizontally along its primary access path.

Hibernate Shards is a framework that is designed to encapsulate support for horizontal partitioning into the Hibernate Core.



> Sharding =Horizontal partitioning
• Split table by rows into partitions

Caching

JPA Level 2 caching avoids database access for already loaded entities, this make reading reading frequently accessed unmodified entities faster, however it can give bad scalability for frequent or concurrently updated entities.

You should configure L2 caching for entities that are:

- read often
- modified infrequently
- Not critical if stale

You should also configure L2 (vendor specific) caching for maxElements, time to expire, refresh...

References and More Information:

JPA Best Practices presentation
MySQL for Developers Article
MySQL for developers presentation
MySQL for developers screencast
Keeping a Relational Perspective for Optimizing Java Persistence
Java Persistence with Hibernate
Pro EJB 3: Java Persistence API
Java Persistence API 2.0: What's New ?
High Performance MySQL book
Pro MySQL, Chapter 6: Benchmarking and Profiling
EJB 3 in Action
sharding the hibernate way
JPA Caching
Best Practices for Large-Scale Web Sites: Lessons from eBay

Category: Sun

Tags: eclipselink hibernate java jpa mysql persistence

Permanent link to this entry

« JPA Caching | Main | Some Concurrency... »

Comments:

Never use L2 caching for tabels modified outside application. You can see no changes to database.

Posted by **Radosław Smogura** on September 09, 2009 at 11:54 PM EDT #

Hi,

Thanks for the interesting article. I have a question regarding the SELECT NEW clause. Specifically what exactly goes on behind the scenes compared to a normal SELECT that returns mapped entities? Does the SELECT NEW clause take advantage of the 2nd level cache?

When I profiled a simple test application querying (essentially read-only) product catalogue data I got much lower memory usage for SELECT NEW, and for larger results sets consistently better response times.

both queries involved at the same number of joins, although the SELECT NEW returned a slightly smaller subset of the data.

Posted by **Jonathan** on September 14, 2009 at 10:54 AM EDT #

how we maintain database using java techonology

Posted by **nitin kumar** on September 15, 2009 at 01:06 AM EDT #

Unfortunately JPA 1 and 2 do not support join fetches over multiple tables.

The spec says: "It is not permitted to specify an identification variable for the objects referenced by the right side of the FETCH JOIN clause, and hence references to the implicitly fetched entities or elements cannot appear elsewhere in the query."

I think this is the most annoying part of the specification, because it prohibits you from writing really high-performance queries for each user case of your application. As long as this will not be fixed, yo can not really use the JPA alone to build your enterprise application on it.
- Sorry guys, it's hard, but it's reality

Posted by **Scotty24** on September 15, 2009 at 01:29 AM EDT #

You can write the query the old fashion way - without the keyword JOIN. Now the FETCH is another story that quite honestly, I have not played with too much. I abhor lazy loading in distributed applications. I create specific fetch methods for dependent data in most cases.

Posted by **darted** on September 15, 2009 at 05:50 AM EDT #

NOJODA VALEN VERGA ESTO NO ME SIRVE PARA NADA PERROS MALPARIDOS!!!!

Posted by **RUPERTO DE JESUS** on December 01, 2009 at 08:48 AM EST #

NADA SOLO PASABA POR AQUI :D

Posted by
CHRISTIANCAMILODELSANTOSOCORROTRINIDADTOBAGODELASPERPETUASLUCESDELSEÑORJESUCRISTOPORLAINFINITAMISERIC GOMEZ on December 01, 2009 at 08:50 AM EST #

Hi Carol,
I couldn't find way to declare index in JPA tutorial (http://download-llnw.oracle.com/javaee/6/tutorial/doc/bnbpy.html). Your blog seems to suggest that I have to go to the RDBMS directly and add any index I feel necessary. Did I understand your blog correctly? Thanks for any pointer

PS: I got to know JPA by way of AribaWeb.org framework, so pardon my lack of depth JPA

Sincerely,
Paul Pambudi

Posted by **Paul Pambudi** on September 26, 2010 at 08:04 PM EDT #

Post a Comment:
Comments are closed for this entry.

About

caroljmcdonald

Search

Enter search term:

☑ Search only this blog

Recent Posts

- Working for Sun, Oh, the places I have been
- Wicket, JPA, GlassFish and Java Derby or MySQL
- Java Garbage Collection, Monitoring and Tuning
- Web Site Performance Tips and Tools
- The Top 10 Web Application security vulnerabilities
- OWASP Top 10 number 3: Malicious File Execution
- Top 10 web security vulnerabilities number 2: Injection Flaws
- The Top 10 Web Application security vulnerabilities starting with XSS
- Some Concurrency Tips
- JPA Performance, Don't Ignore the Database

Top Tags

- ajax
- bayeux
- collection
- comet
- concurrency
- dojo
- eclipselink
- economy
- ejb
- esapi
- esb
- garbage
- glassfish
- grails
- grizzly
- groovy
- hibernate
- java
- javaee
- javafx
- javascript
- jax-rs
- jax-ws
- jmaki
- jpa
- jruby
- jsf
- memory
- monitoring
- mysql
- netbeans
- owasp
- performance
- persistence
- rails
- rest
- seam
- security
- services
- spring
- tuning
- web
- wicket
- xss

Categories

- Personal
- Sun

Archives

« July 2015

Sun	Mon	Tue	Wed	Thu	Fri	Sat
			1	2	3	4
5	6	7	8	9	10	11

12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	

Today

Bookmarks

- blogs.sun.com
- java.com
- java.net
- opensolaris.org

Menu

- Blogs Home
- Weblog
- Login

Feeds

RSS

- All
- /Personal
- /Sun
- Comments

Atom

- All
- /Personal
- /Sun
- Comments

The views expressed on this blog are those of the author and do not necessarily reflect the views of Oracle. Terms of Use | Your Privacy Rights | Cookie Preferences