An Oracle White Paper
April 2010

MySQL Connector for Java

ORACLE

Disclaimer

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Introduction 1
Technical Overview 2
ClusterJ 3
ClusterJPA 5
Tutorials 8
Prerequisites 8
ClusterJ Tutorial 10
OpenJPA / ClusterJPA Tutorial 17
Conclusion 25
Appendix: References 25

Introduction

MySQL Cluster database has been widely adopted for a range of telecommunications, Web, and enterprise workloads that demand carrier-grade availability with high-transaction throughput and low latency.

MySQL Cluster is architected with a distributed cluster of data nodes that are used to store database tables. The data can be accessed via a standard structured query language (SQL) interface implemented by the MySQL server or through native APIs (C++, LDAP, and HTTP).

Designed for Java developers, the new MySQL Connector for Java implements an easy-to-use, high-performance native Java interface and OpenJPA plug-in that directly map Java objects to relational tables stored in the MySQL Cluster database.

Eliminating data transformations into SQL results in higher throughput and reduced data access latency for users. In addition, Java developers have a more-natural programming method to directly manage their data with a complete, feature-rich solution for object-relational mapping. The simplified development of Java applications results in faster development cycles and an accelerated time to market for new services.

The purpose of this white paper is to introduce you to the technology behind the MySQL Connector for Java (called “ClusterJ”) and to provide tutorials demonstrating how to compile and run code.

Technical Overview

Prior to MySQL Cluster 7.1, the options for Java developers were somewhat limited. The only supported method was JDBC—either directly or through a third-party layer such as Java Persistence API (JPA) compliant middleware. If Java developers wanted faster performance, then they had to write their own Java Native Interface (JNI) layer, which operated between their (Java) application and the (C++) NDBCLUSTER (NDB) API. As shown from left to right in Figure 1, MySQL Cluster 7.1 provides more options for Java developers.

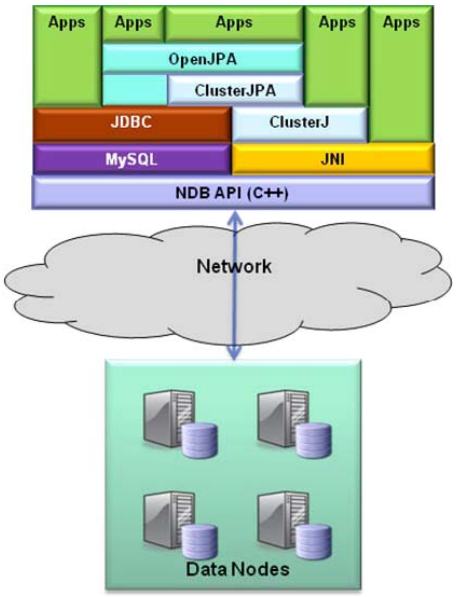

Figure 1. Options for Java applications

- Developers can have the Java application make JDBC calls through a JDBC driver for MySQL (Connector/J). Although many developers are comfortable with this flexible technology, they are required to map their Java objects to relational SQL statements. In addition, performance can be compromised as Connector/J accesses the data through a MySQL server rather than directly accessing the data nodes through the NDB API.
- Developers can shift the responsibility for the object-relational mapping from the Java application to a third-party JPA solution such as Hibernate, TopLink, and OpenJPA. Although this approach allows the developer to avoid working with a relational data model, performance can still be limited if the JPA layer must access the data via JDBC.
- Developers can use ClusterJPA, an OpenJPA plug-in introduced in MySQL Cluster 7.1 that enables most JPA operations to use the NDB API (via a new ClusterJ layer) while the remaining operations use JDBC. This approach allows the developer to work only with objects and also receive many of the performance benefits of using the NDB API.

- Developers can bypass the JPA layer and instead use the ClusterJ layer directly. This approach introduces some limitations, but it might be appropriate for developers who want the best-possible performance and who want to avoid using additional third-party components (OpenJPA).
- Developers can implement their own wrapper (typically using JNI) to act as the Java/C++ mediator.

ClusterJ

ClusterJ provides a high-performance method for Java applications to store and access data in a MySQL Cluster database. It is designed to be easy for Java developers to use and is in the style of Hibernate, Java Data Objects, and JPA. It uses the Domain Object Model DataMapper pattern:

- Data is represented as domain objects.
- Domain objects are separate from business logic.
- Domain objects are mapped to database tables.

The purpose of ClusterJ is to provide a mapping from the relational view of the data stored in MySQL Cluster to the Java objects used by the application. This is achieved by annotating interfaces representing the Java objects, where each persistent interface is mapped to a table and each property in that interface to a column. By default, the table name will match the interface name, and the column names match the property names; but this can be overwritten using the annotations.

If the table does not already exist (for example, this is a brand-new application with new data), then the table must be created manually. Unlike OpenJPA, ClusterJ will not create the table automatically. Figure 2 shows an example of an interface that has been created to represent the data held in the Employee table.

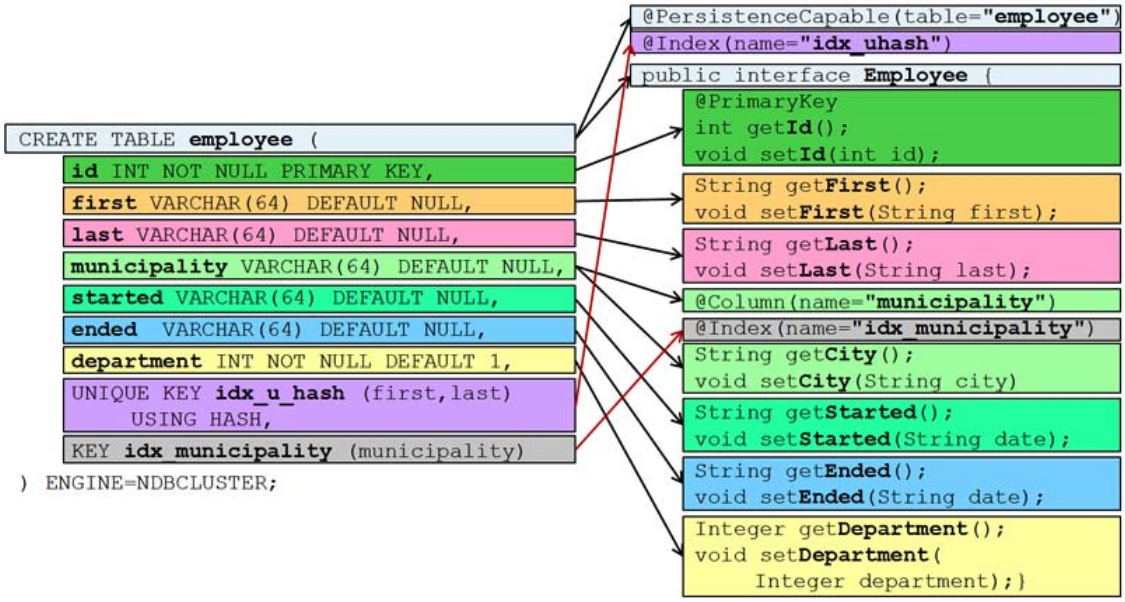

Figure 2. ClusterJ annotation example

ClusterJ uses the following concepts:

- **SessionFactory.** There is one instance per MySQL Cluster instance for each Java Virtual Machine (JVM). The application uses the SessionFactory object to get hold of sessions. The configuration details for the ClusterJ instance are defined in the configuration properties, which are an artifact associated with the SessionFactory.
- **Session.** There is one instance per user (per MySQL Cluster, per JVM) and represents a MySQL Cluster connection.
- **Domain object.** Objects represent the data from a table. The domain objects and their relationships to the MySQL Cluster tables are defined by annotated interfaces (as shown on the right in Figure 3.)
- **Transaction.** There is one transaction per session at a time. By default, each query is run under its own transaction.

Figure 3. ClusterJ terminology

ClusterJ is suitable for many Java developers, but it has some restrictions that could make OpenJPA with the ClusterJPA plug-in more appropriate. These ClusterJ restrictions are as follows:

- **Persistent interfaces rather than persistent classes.** The developer provides the signatures for the getter/setter methods rather than the properties. No extra methods can be added.
- **Properties are primitive types.** Relationships between properties or objects cannot be defined in the domain objects.
- **No multitable inheritance.** There is a single table per persistent interface.
- **No joins in queries.** All data being queried must be in the same table/interface.
- **No table creation.** The user must create tables and indexes.
- **No lazy loading.** The entire record is loaded at one time, including large objects.

ClusterJPA

JPA is the Java standard for persistence. Different vendors can implement their own implementation of this API and can add proprietary extensions. Three of the most common implementations are OpenJPA, Hibernate, and TopLink. JPA can be used within server containers or outside them—that is, with either Java 2 Platform, Enterprise Edition (J2EE) or Java 2 Platform, Standard Edition (J2SE).

Typically, a JPA implementation accesses the database (for example, MySQL Cluster) using JDBC, which provides a great deal of flexibility to the JPA implementer. However, JDBC cannot provide the best performance with MySQL Cluster because Connector/J performs an internal conversion to SQL, and the MySQL server performs a subsequent translation from SQL to the C++ NDB API. In MySQL Cluster 7.1, OpenJPA can be configured to use the high-performance NDB API (via ClusterJ) for most operations, and configured to use JDBC for more-complex queries, as shown in the table.

OPENJPA → CLUSTERJ → NDB API	OPENJPA → CONNECTOR/J → MYSQL → NDB API
Insert	
Delete	
Find (primary key reads)	
Update	
	Other queries

The first implementation of ClusterJPA is as an OpenJPA BrokerFactory but, in the future, it might be extended to work with other JPA implementations.

ClusterJPA overcomes ClusterJ limitations:

- Persistent classes
- Relationships
- Joins in queries
- Lazy loading
- Table and index creation from object model

Typically, developers base their selection of a JPA solution on factors such as proprietary extensions, which existing applications are already in use, and performance (increasingly a factor with ClusterJPA).

See Figure 4 for a comparison of the performance of ClusterJPA (OpenJPA using ClusterJ) versus OpenJPA using JDBC. Note that the performance is significantly better when using ClusterJPA (as shown by the yellow bar). It is hoped that in the future the performance can be improved even further for finds, updates, and deletes.

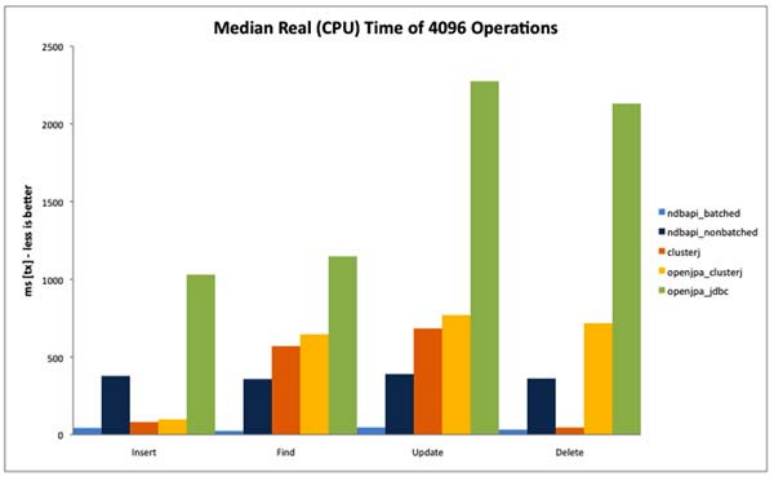

Figure 4. Performance of OpenJPA

Adapting an OpenJPA-based application to use ClusterJPA with MySQL Cluster is straightforward; the main change is in the definition of the persistence unit in persistence.xml:

```
<persistence xmlns=http://java.sun.com/xml/ns/persistence
      xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance version="1.0">
 <persistence-unit name="clusterdb" transaction-type="RESOURCE_LOCAL">
  <provider> org.apache.openjpa.persistence.PersistenceProviderImpl
   </provider>
   <class>Employee</class>
   <class>Department</class>
   <properties>
    <property name="openjpa.jdbc.SynchronizeMappings"
       value="buildSchema" />
    <property name="openjpa.ConnectionDriverName"
       value="com.mysql.jdbc.Driver" />
   <property name="openjpa.ConnectionURL"
       value="jdbc:mysql://localhost:3306/clusterdb" />
    <property name="openjpa.ConnectionUserName" value="root" />
    <property name="openjpa.ConnectionPassword" value="" />
    <property name="openjpa.BrokerFactory" value="ndb" />
    <property name="openjpa.jdbc.DBDictionary"
    value="TableType=ndbcluster" />
    <property name="openjpa.ndb.connectString" value="localhost:1186"
          />
    <property name="openjpa.ndb.database" value="clusterdb" /
   </properties>
 </persistence-unit>
</persistence>
```

To define the object-to-table mappings, annotate the persistent class for the domain object. If the table does not exist, OpenJPA will create it. The property openjpa.jdbc.DBDictionary tells OpenJPA to create the tables using NDB as the storage engine.

This white paper does not describe the details of JPA; it focuses instead on the specifics of using OpenJPA with MySQL Cluster / ClusterJPA.[1]

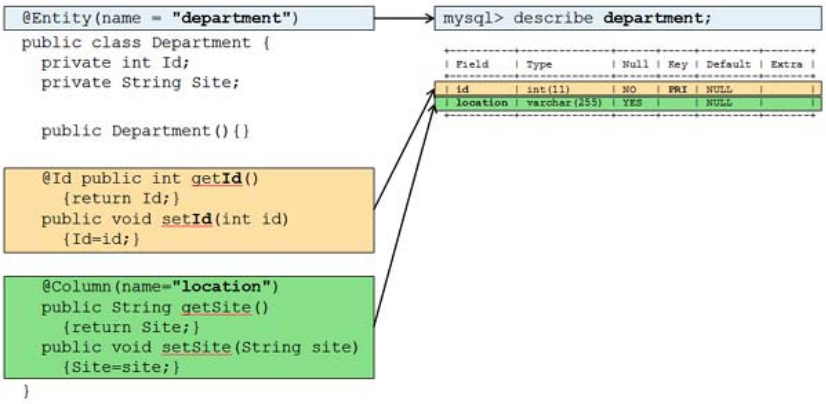

Figure 5. ClusterJPA class annotation

Tutorials

Prerequisites

The following tutorials use MySQL Cluster 7.1.2a on Fedora 12. If you are using a more-recent version of MySQL Cluster, then you might need to change the classpaths, as explained in

- dev.mysql.com/doc/ndbapi/en/mccj-using-clusterj.html
- dev.mysql.com/doc/ndbapi/en/mccj-using-jpa.html

You must have MySQL Cluster running for both tutorials. For simplicity, all the nodes (processes) making up the MySQL Cluster will be run on the same physical host along with the application.

Although most of the database access is performed through the NDB API, MySQL Cluster includes a MySQL server process for OpenJPA to use for complex queries and to enable the user to check the contents of the database manually.

[1]For more information on the use of JPA and OpenJPA, refer to openjpa.apache.org/.

The following MySQL Cluster configuration files are used:[2]

config.ini:

```
[ndbd default]noofreplicas=2
datadir=/home/billy/mysql/my_cluster/data

[ndbd]
hostname=localhost
id=3

[ndbd]
hostname=localhost
id=4

[ndb_mgmd]
id = 1
hostname=localhost
datadir=/home/billy/mysql/my_cluster/data

[mysqld]
hostname=localhost
id=101

[api]
hostname=localhost
```

my.cnf:

```
[mysqld]
ndbcluster
datadir=/home/billy/mysql/my_cluster/data
basedir=/usr/local/mysql
```

These tutorials focus on ClusterJ and ClusterJPA rather than on running MySQL Cluster. If you are unfamiliar with MySQL Cluster, refer to clusterdb.com/mysql-cluster/creating-a-simple-clusteron-a-single-linux-host/ before trying these tutorials.

[2]Typically, you would specify many more parameters to get the best performance out of MySQL Cluster, but that is beyond the scope of this tutorial.

ClusterJ Tutorial

ClusterJ must be configured with parameters that specify how to connect to the MySQL Cluster database. These parameters include the connect string (the address/port for the management node), the database to use, the login username, and the attributes for the connection such as the time-out values. If these parameters aren't defined, then ClusterJ will fail with runtime exceptions. This information represents the configuration properties shown in Figure 3.

These parameters can be hard coded in the application code, but it is more maintainable to create a clusterj.properties file that will be imported by the application. This file should be stored in the same directory as your application source code.

clusterj.properties:

```
com.mysql.clusterj.connectstring=localhost:1186
com.mysql.clusterj.database=clusterdb
com.mysql.clusterj.connect.retries=4
com.mysql.clusterj.connect.delay=5
com.mysql.clusterj.connect.verbose=1
com.mysql.clusterj.connect.timeout.before=30
com.mysql.clusterj.connect.timeout.after=20
com.mysql.clusterj.max.transactions=1024
```

Because ClusterJ will not create tables automatically, the next step is to create the "clusterdb" database (referred to in clusterj.properties) and the Employee table:

```
[billy@ws1 ~]$ mysql -u root -h 127.0.0.1 -P 3306 -u root
mysql> create database clusterdb;use clusterdb;
mysql> CREATE TABLE employee (
-> id INT NOT NULL PRIMARY KEY,
-> first VARCHAR(64) DEFAULT NULL,
-> last VARCHAR(64) DEFAULT NULL,
-> municipality VARCHAR(64) DEFAULT NULL,
-> started VARCHAR(64) DEFAULT NULL,
-> ended VARCHAR(64) DEFAULT NULL,
-> department INT NOT NULL DEFAULT 1,
-> UNIQUE KEY idx_u_hash (first,last) USING HASH,
-> KEY idx_municipality (municipality)
-> ) ENGINE=NDBCLUSTER;
```

The next step is to create the annotated interface:

Employee.java:

```
import com.mysql.clusterj.annotation.Column;
```

```
import com.mysql.clusterj.annotation.Index;
import com.mysql.clusterj.annotation.PersistenceCapable;
import com.mysql.clusterj.annotation.PrimaryKey;

@PersistenceCapable(table="employee")
@Index(name="idx_uhash")
public interface Employee {
@PrimaryKey
int getId();
void setId(int id);

String getFirst();
void setFirst(String first);
String getLast();
void setLast(String last);

@Column(name="municipality")
@Index(name="idx_municipality")
String getCity();
void setCity(String city);

String getStarted();
void setStarted(String date);

String getEnded();
void setEnded(String date);

Integer getDepartment();
void setDepartment(Integer department);
}
```

The name of the table is specified in the following annotation:

```
@PersistenceCapable(table="employee")
```

Each column from the Employee table has an associated getter/setter method defined in the interface. By default, the property name in the interface is the same as the column name in the table, but this is overridden for the City property by the `@Column(name="municipality")` annotation before the associated getter/setter methods. The `@PrimaryKey` annotation is used to identify the property

associated with the primary key column in the table. The inclusion of indexes in the database is specified by the `@Index` annotation.

The next step is to write the application code, which is shown below in blocks. The first block lists the import statements and loads the contents of the clusterj.properties defined above. For details on compiling and running the tutorial code, see the subsection titled "Compiling and Running the ClusterJPA Tutorial Code."

Main.java (part 1):

```
import com.mysql.clusterj.ClusterJHelper;
import com.mysql.clusterj.SessionFactory;
import com.mysql.clusterj.Session;
import com.mysql.clusterj.Query;
import com.mysql.clusterj.query.QueryBuilder;
import com.mysql.clusterj.query.QueryDomainType;

import java.io.File;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.*;

import java.util.Properties;
import java.util.List;

public class Main {

public static void main (String[] args) throws
java.io.FileNotFoundException,java.io.IOException {

// Load the properties from the clusterj.properties file

File propsFile = new File("clusterj.properties");
InputStream inStream = new FileInputStream(propsFile);
Properties props = new Properties();
props.load(inStream);

//Used later to get userinput
BufferedReader br = new BufferedReader(new
InputStreamReader(System.in));
```

The next step is to get a handle for a SessionFactory from the ClusterJHelper class, and then use that factory to create a session based on the properties imported from the clusterj.properties file.

Main.java (part 2):

```
// Create a session (connection to the database)
SessionFactory factory = ClusterJHelper.getSessionFactory(props);
Session session = factory.getSession();
```

Now that there is a session, it is possible to instantiate new Employee objects and save them in the database. Each operation involving the database is treated as a separate transaction if there are no transaction BEGIN() or COMMIT() statements.

Main.java (part 3):

```
// Create and initialise an Employee
Employee newEmployee = session.newInstance(Employee.class);
newEmployee.setId(988);
newEmployee.setFirst("John");
newEmployee.setLast("Jones");
newEmployee.setStarted("1 February 2009");
newEmployee.setDepartment(666);

// Write the Employee to the database
session.persist(newEmployee);
```

At this point, a row will have been added to the Employee table. To verify this, a new Employee object is created and used to read the data back from the Employee table using the primary key (Id) value of 998:

Main.java (part 4):

```
// Fetch the Employee from the database
Employee theEmployee = session.find(Employee.class, 988);

if (theEmployee == null)
 {System.out.println("Could not find employee");}
else
 {System.out.println ("ID: " + theEmployee.getId() + "; Name: " +
  theEmployee.getFirst() + " " + theEmployee.getLast());
 System.out.println ("Location: " + theEmployee.getCity());
 System.out.println ("Department: " + theEmployee.getDepartment());
 System.out.println ("Started: " + theEmployee.getStarted());
 System.out.println ("Left: " + theEmployee.getEnded());
}
```

This is the output seen at this point:

```
ID: 988; Name: John Jones
Location: null
Department: 666
```

```
Started: 1 February 2009
Left: null
Check the database before I change the Employee - press Return when
you are done
```

The next step is to modify this data, but it does not write it back to the database yet:

Main.java (part 5):

```
// Make some changes to the Employee & write back to the database
theEmployee.setDepartment(777);
theEmployee.setCity("London");

System.out.println("Check the database before I change the Employee -
 press Return when you are done");
String ignore = br.readLine();
```

The application will pause at this point and give you a chance to check the database to confirm that the original data has been added as a new row, but the changes have not been written back to the database yet:

```
mysql> select * from clusterdb.employee;
+-----+-------+-------+--------------+-----------------+-------+------------+
| id | first | last | municipality | started    | ended | department |
+-----+-------+-------+--------------+-----------------+-------+------------+
| 988 | John | Jones | NULL    | 1 February 2009 | NULL |    666 |
+-----+-------+-------+--------------+-----------------+-------+------------+
```

After the user presses Return, the application will continue and write the changes to the table.

Main.java (part 6):

```
session.updatePersistent(theEmployee);

System.out.println("Check the change in the table before I bulk add
Employees - press Return when you are done");
ignore = br.readLine();
```

The application will again pause, so you can now check that the change has been written back (persisted) to the database:

```
mysql> select * from clusterdb.employee;
+-----+-------+-------+--------------+-----------------+-------+------------+
| id | first | last | municipality | started    | ended | department |
+-----+-------+-------+--------------+-----------------+-------+------------+
| 988 | John | Jones | London   | 1 February 2009 | NULL |    777 |
+-----+-------+-------+--------------+-----------------+-------+------------+
```

The application then creates and writes 100 new employees. To improve performance, a single transaction is used so that all the changes can be written to the database at once when the COMMIT() statement is run:

Main.java (part 7):

```
// Add 100 new Employees - all as part of a single transaction
newEmployee.setFirst("Billy");
newEmployee.setStarted("28 February 2009");

session.currentTransaction().begin();

 for (int i=700;i<800;i++) {
  newEmployee.setLast("No-Mates"+i);
  newEmployee.setId(i+1000);
  newEmployee.setDepartment(i);
  session.persist(newEmployee);
 }
session.currentTransaction().commit();
```

The 100 new employees will now have been written back to the database. The next step is to create and execute a query that will search the database for all employees in department 777 by using a QueryBuilder and using that query to build a QueryDomain that compares the Department column with a parameter. After creating the query, the DEPARTMENT parameter is set to 777 (the query could subsequently be reused with different department numbers). The application then runs the query and iterates through and displays each of the employees in the result set:

Main.java (part 8):

```
// Retrieve the set all of Employees in department 777
QueryBuilder builder = session.getQueryBuilder();
QueryDomainType<Employee> domain =
 builder.createQueryDefinition(Employee.class);
domain.where(domain.get("department").equal(domain.param(
 "department")));
Query<Employee> query = session.createQuery(domain);
query.setParameter("department",777);

List<Employee> results = query.getResultList();
for (Employee deptEmployee: results) {
 System.out.println ("ID: " + deptEmployee.getId() + "; Name: " +
   deptEmployee.getFirst() + " " + deptEmployee.getLast());
 System.out.println ("Location: " + deptEmployee.getCity());
 System.out.println ("Department: " + deptEmployee.getDepartment());
```

```
 System.out.println ("Started: " + deptEmployee.getStarted());
 System.out.println ("Left: " + deptEmployee.getEnded());
}

System.out.println("Last chance to check database before emptying table
- press Return when you are done");
ignore = br.readLine();
```

At this point, the application will display the following output and prompt the user to enable it to continue:

```
ID: 988; Name: John Jones
Location: London
Department: 777
Started: 1 February 2009
Left: null
ID: 1777; Name: Billy No-Mates777
Location: null
Department: 777
Started: 28 February 2009
Left: null
```

You can compare that output with an SQL query performed on the database:

```
mysql> select * from employee where department=777;
+------+-------+-------------+-------------+------------------+-------+------------+
| id | first | last    | municipality | started     | ended | department |
+------+-------+-------------+-------------+------------------+-------+------------+
| 988 | John | Jones   | London   | 1 February 2009 | NULL |    777 |
| 1777 | Billy | No-Mates777 | NULL    | 28 February 2009 | NULL |    777 |
+------+-------+-------------+-------------+------------------+-------+------------+
```

Finally, after the user presses Return, the application will remove all employees:

Main.java (part 9):

```
  session.deletePersistentAll(Employee.class);
 }
}
```

As a final check, an SQL query confirms that all the rows have been deleted from the Employee table.

```
mysql> select * from employee;
Empty set (0.00 sec)
```

Compiling and Running the ClusterJ Tutorial Code

```
javac -classpath /usr/local/mysql/share/mysql/java/clusterj-api.jar:.
Main.java
```

```
Employee.java

java -classpath
usr/local/mysql/share/mysql/java/clusterj.jar:. -
Djava.library.path=/usr/local/mysql/lib Main
```

OpenJPA / ClusterJPA Tutorial

JPA, OpenJPA, and ClusterJPA can be used within or outside a container (that is, they can be used with J2EE or J2SE). For simplicity, this tutorial does not use a container (that is, it is written using J2SE).

Before you can run any ClusterJPA code, you must first download and install OpenJPA from openjpa.apache.org/. This tutorial uses OpenJPA 1.2.1. Extract the contents of the binary tarball to the host where you want to run your application. For this tutorial, use /usr/local/openjpa, for example.

Because ClusterJPA must sometimes use JDBC to satisfy certain queries, the JDBC Driver for MySQL (Connector/J) should also be installed. This can be downloaded from dev.mysql.com/downloads/connector/j/. Extract the contents of the tarball. For this tutorial, the files are stored in /usr/local/connectorj, and version 5.1.12 is used.

If the ClusterJ tutorial has already been run on this MySQL Cluster database, drop the tables so that you can observe them being created automatically. In a real application, you might prefer to create them manually.

A configuration file is required to indicate how persistence is to be handled for the application. Create a new directory called META-INF in the application source directory, and create a file in that directory called persistence.xml:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0">
 <persistence-unit name="clusterdb" transaction-type="RESOURCE_LOCAL">
 <provider>
  org.apache.openjpa.persistence.PersistenceProviderImpl
 </provider>
 <class>Employee</class>
 <class>Department</class>
 <properties>
  <property name="openjpa.jdbc.SynchronizeMappings"
   value="buildSchema" />
  <property name="openjpa.ConnectionDriverName"
   value="com.mysql.jdbc.Driver" />
  <property name="openjpa.ConnectionURL"
   value="jdbc:mysql://localhost:3306/clusterdb" />
  <property name="openjpa.ConnectionUserName" value="root" />
  <property name="openjpa.ConnectionPassword" value="" />
  <property name="openjpa.BrokerFactory" value="ndb" />
```

```
  <property name="openjpa.jdbc.DBDictionary" value="TableType=ndb"/>
  <property name="openjpa.ndb.connectString" value="localhost:1186"
  />
  <property name="openjpa.ndb.database" value="clusterdb" />
 </properties>
 </persistence-unit>
</persistence>
```

A persistence unit called "clusterdb" is created, and the provider (the implementation for the persistence) is set to "openjpa". Two classes are specified— Employee and Department—which relate to the persistent classes that the application will define. Connector/J is defined as the JDBC connection (together with the host and the port of the MySQL server to be used). The key to having OpenJPA use ClusterJPA is to set the BrokerFactory to ndb, and specify the connect string (host:port) for the MySQL Cluster management node. The database is defined as "clusterdb" for both the JDBC and ClusterJ connections.

If you have not already done so, create the "clusterdb" database.

```
mysql> create database clusterdb;
```

If the database contains tables from the ClusterJ tutorial, drop them.

The next step is to create the persistent class definitions for the Department and Employee entities:

Department.java:

```
import javax.persistence.*;

@Entity(name = "department")
public class Department {

private int Id;
private String Site;

public Department(){}

@Id public int getId() {return Id;}
public void setId(int id) {Id=id;}

@Column(name="location")
public String getSite() {return Site;}
public void setSite(String site) {Site=site;}

public String toString() {
return "Department: " + getId() + " based in " + getSite();
}
```

```
}
```

The `@Entity` tag specifies the table name to be "department". Note that unlike ClusterJ, ClusterJPA uses persistent classes (rather than interfaces), and consequently, it is necessary to define the properties as well as the getter and setter methods. The `@Id` tag defines the primary key, and the `@Column` tag specifies that the column associated with the Site property is called "location."

Because this is a class, it is possible to add other useful methods, in this case, `toString()`.

Employee.java:

```
import javax.persistence.*;

@Entity(name = "employee") //Name of the table

public class Employee {

private int Id;
private String First;
private String Last;
private String City;
private String Started;
private String Ended;
private int Department;

public Employee(){}

@Id public int getId() {return Id;}
public void setId(int id) {Id=id;}

public String getFirst() {return First;}
public void setFirst(String first) {First=first;}

public String getLast() {return Last;}
public void setLast(String last) {Last=last;}

@Column(name="municipality")
public String getCity() {return City;}
public void setCity(String city) {City=city;}

public String getStarted() {return Started;}
public void setStarted(String date) {Started=date;}

public String getEnded() {return Ended;}
public void setEnded(String date) {Ended=date;}
```

```
public int getDepartment() {return Department;}
public void setDepartment(int department) {Department=department;}

public String toString() {
 return getFirst() + " " + getLast() + " (Dept " +
   getDepartment()+ ") from " + getCity() +
   " started on " + getStarted() + " & left on " + getEnded();
}
}
```

The next step is to write the application code, which is shown below in blocks. For details on compiling and running the tutorial code, see the subsection titled "Compiling and Running the ClusterJPA Tutorial Code."[3]

Main.java (part 1):

```
import java.util.List;
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;
import java.io.*;

public class Main {

public static void main (String[] args) throws java.io.IOException {

EntityManagerFactory entityManagerFactory =
  Persistence.createEntityManagerFactory("clusterdb");
EntityManager em = entityManagerFactory.createEntityManager();
 EntityTransaction userTransaction = em.getTransaction();

BufferedReader br = new BufferedReader(new
  InputStreamReader(System.in));

System.out.println("Check engine used for clusterdb.employee &
 Change to ndb if needed.");
```

[3]If you run the tutorial more than once, you must delete all rows from the two tables after each run.

```
System.out.println("mysql> alter table clusterdb.employee
  engine=ndb;");
System.out.println("mysql> alter table clusterdb.department
  engine=ndb;");
System.out.println("Press Return when you are done");
String ignore = br.readLine();
```

As part of creating the EntityManagerFactory and EntityManager, OpenJPA creates the tables for the two classes specified for the "clusterdb" persistence unit. While the application waits for the user to press Return, this can be checked:

```
mysql> use clusterdb
mysql> show tables;
+---------------------+
| Tables_in_clusterdb |
+---------------------+
| department     |
| employee       |
+---------------------+
```

For ClusterJPA to be able to access the data via ClusterJ and the NDB API, the tables must use the MySQL Cluster storage engine. At the time of writing, the tables are created using the InnoDB storage engine as can be seen using `show create table:`

```
mysql> show create table department;
+------------+--------------------------------------+
| Table      | Create Table                         |
| department | CREATE TABLE `department` (     |
       | `id` int(11) NOT NULL,       |
       | `location` varchar(255) DEFAULT NULL,|
       | PRIMARY KEY (`id`))          |
       | ENGINE=InnoDB DEFAULT CHARSET=latin1 |
+------------+--------------------------------------+
```

If the engine is not already set to ndb, then change it:

```
mysql> alter table department engine=ndb;
mysql> alter table employee engine=ndb;
```

After the user presses Return, the application creates an Employee object and stores it in the Employee table. A second Employee object is then created and populated with the data read back from the database, using a primary key lookup on the Id property with a value of 1.

Main.java (part 2):

```
userTransaction.begin();
Employee emp = new Employee();
emp.setId(1);
emp.setDepartment(666);
```

```
emp.setFirst("Billy");
emp.setLast("Fish");
emp.setStarted("1st February 2009");
em.persist(emp);
userTransaction.commit();

userTransaction.begin();
Employee theEmployee = em.find(Employee.class, 1);
userTransaction.commit();

System.out.println(theEmployee.toString());

System.out.println("Chance to check the database before City is set");
System.out.println("Press Return when you are done");
ignore = br.readLine();
```

The Employee object that is read back from the database is displayed:

```
Billy Fish (Dept 666) from null started on 1st February 2009 & left on
null
Chance to check the database before City is set
Press Return when you are done
```

At this point, the application waits to give the user a chance to confirm that the employee really has been written to the database:

```
mysql> select * from employee;
+----+--------------+------------+-------+-------+------+-------------------+
| id | municipality | department | ended | first | last | started     |
+----+--------------+------------+-------+-------+------+-------------------+
| 1 | NULL    |    666 | NULL | Billy | Fish | 1st February 2009 |
+----+--------------+------------+-------+-------+------+-------------------+
```

After the user presses Return, the application continues and an update is made to the persisted Employee object. Note that there is no need to explicitly ask for the changes to be persisted; this happens automatically when the transaction is committed.

Main.java (part 3):

```
userTransaction.begin();
theEmployee.setCity("London");
theEmployee.setDepartment(777);
userTransaction.commit();

System.out.println("Chance to check the City is set in the database");
System.out.println("Press Return when you are done");
ignore = br.readLine();
```

At this point, the application waits while the user has a chance to confirm that the changes did indeed get written through to the database:

```
mysql> select * from employee;
+----+--------------+------------+-------+-------+------+------------------+
| id | municipality | department | ended | first | last | started     |
+----+--------------+------------+-------+-------+------+------------------+
| 1 | London   |    777 | NULL | Billy | Fish | 1st February 2009 |
+----+--------------+------------+-------+-------+------+------------------+
```

When allowed to continue, the application creates and persists an additional 100 Employee and Department entities. Next, it creates and executes a query to find all employees with a department number of 777, and then looks up the location of the site for that department.

Main.java (part 4):

```
Department dept;

userTransaction.begin();
for (int i=700;i<800;i++) {
emp = new Employee();
dept = new Department();
emp.setId(i+1000);
emp.setDepartment(i);
emp.setFirst("Billy");
emp.setLast("No-Mates-"+i);
emp.setStarted("1st February 2009");
em.persist(emp);
dept.setId(i);
dept.setSite("Building-"+i);
em.persist(dept);
}
userTransaction.commit();

userTransaction.begin();

Query q = em.createQuery("select x from Employee x where
  x.department=777");
Query qd;

for (Employee m : (List<Employee>) q.getResultList()) {
System.out.println(m.toString());
qd = em.createQuery("select x from Department x where x.id=777");
for (Department d : (List<Department>) qd.getResultList()) {
```

```
    System.out.println(d.toString());
 }
}
userTransaction.commit();
```

These are the results displayed:

```
Billy No-Mates-777 (Dept 777) from null started on 1st February 2009 &
left on null
Department: 777 based in Building-777
Billy Fish (Dept 777) from London started on 1st February 2009 & left on
null
Department: 777 based in Building-777
```

Note that joins between tables are possible with JPA, but that is beyond the scope of this tutorial.

Finally, the EntityManager and EntityManagerFactory are closed:

Main.java (part 5):

```
em.close();
entityManagerFactory.close();
}
}
```

Compiling and Running the ClusterJPA Tutorial Code

```
javac -classpath /usr/local/mysql/share/mysql/java/*.jar:
  /usr/local/openjpa/openjpa-
1.2.1.jar:/usr/local/openjpa/lib/geronimo-jpa_3.0_spec-1.0.jar:
  . Main.java
Employee.java Department.java


java -Djava.library.path=/usr/local/mysql/lib -classpath
/usr/local/mysql/share/mysql/java/*:/usr/local/mysql/share/java/*:/usr
  /local/openjpa/
openjpa-1.2.1.jar:/usr/local/openjpa/lib/*:
  /usr/local/connectorj/mysql-connectorjava-5.1.12-bin.jar:. Main
```

Conclusion

The MySQL Connector for Java makes it much easier for Java application developers to take advantage of the high-performance and availability features offered by the MySQL Cluster database.

This white paper has provided an overview of the technology and tutorials to give you a head start in extending your services with the persistence capabilities offered by MySQL Cluster.

Appendix: References

- **MySQL Cluster on the Web:** mysql.com/products/database/cluster/
- ***MySQL Cluster 7: Architecture and New Features* white paper:** mysql.com/why-mysql/white-papers/
  mysql_wp_cluster7_architecture.php
- ***MySQL Cluster Connector for Java*:** dev.mysql.com/doc/ndbapi/en/mccj.html
- **OpenJPA:** openjpa.apache.org/
- **Source code from the tutorials:** ftp.mysql.com/pub/mysql/download/
  ClusterJ_Examples.tar.gz
- **Running MySQL Cluster tutorial (single site):** clusterdb.com/mysql-cluster/creating-a-simple-cluster-on-a-single-linux-host/
- **Running MySQL Cluster tutorial (multiple sites):** clusterdb.com/mysql-cluster/deploying-mysql-cluster-over-multiple-hosts/

Oracle is committed to developing practices and products that help protect the environment

MySQL Connector for Java
April 2010

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200
oracle.com

SOFTWARE. HARDWARE. COMPLETE.