Using CA IDMS with the Java Persistence API (JPA)

David Hearn
Sr. Sustaining Engineer

Object-Relational Mapping (ORM) solutions such as Hibernate and OpenJPA have become increasingly popular with Java developers in recent years. With an ORM, developers are able to work with data as Java objects rather than as the rows and columns of a relational database. Developers can then focus on the business logic of their applications and generally don't have to be concerned with the mechanics of how the data is being *persisted;* that is, read or written to a permanent data store such as a DBMS. For the most part, the ORM *provider* (Hibernate, OpenJPA, and so on) takes care of building SQL statements and making calls to the appropriate DBMS interface.

Some ORM providers have their own APIs (Hibernate is a notable example), but in many cases developers prefer to use the Java Persistence API (JPA) which is part of J2EE 1.5. This allows them the option of changing ORM providers if they desire without having to do extensive recoding of their applications. JPA's use of the Annotations feature of Java 1.5 also makes coding easier.

Using ORM solutions successfully with CA IDMS largely depends on how the data is defined. Data defined using SQL DDL in IDMS will generally be handled as it would be for any other relational DBMS, at least for IDMS r17 and later. However, most IDMS sites developing distributed applications are interested in accessing their legacy data, usually defined as one or more network databases. IDMS network databases can indeed be accessed with SQL, but there are some special considerations for doing so, which we will discuss in this article.

Worth noting for our discussion is that in IDMS network terminology, a *record* is equivalent to an SQL table, a *record occurrence* is equivalent to a row and an *element* is equivalent to a column.

The most notable issue with accessing IDMS network data using SQL is that SQL provides no direct way of navigating the set structures that define the relationships between records. In SQL-defined databases, whether in IDMS or another DBMS, the relationships between rows are defined using foreign keys, not with pointers as is the case with sets. While it is possible (and recommended) to define IDMS network databases with foreign keys which can be used in addition to sets, most sites have continued to rely solely on sets. Adding foreign keys requires database restructuring and changes to

applications, and most sites have not committed the time and effort to implement them.

Related to this is the fact that some IDMS records do not have an unambiguous primary key since it's assumed that they will always be accessed through the associated "owner" record defined in the set. This is particularly true for records defined in IDMS as *VIA location mode.* With JPA a programmer can get around this problem to an extent by using *composite* keys comprising multiple data elements in the record, but in some cases there is simply no way to guarantee the uniqueness of such a key, even if every element in the record is included.

One possible solution to these issues is to use IDMS SQL procedures or routines to handle set navigation. This allows most of the complexities of the IDMS database to be hidden from a distributed application. However, this requires a developer skilled in IDMS application programming, not something typically found in Java development teams.

So, for a Java developer using JPA, what other options are available for set navigation? Luckily, IDMS provides proprietary join syntax for fetching record occurrences linked by sets, and provides a ROWID "pseudo-column" which can uniquely identify each one, even if such occurrences lack a unique primary key and can be reliably retrieved only by going through the set. Using these two features in combination solves most set navigation issues.

Joining Records Using a Set

With all this in mind, let's see how to retrieve an owner record and its associated member records using a join. Note that the examples shown here utilize the Employee Demo database distributed with IDMS; the complete Java code and related configuration files are attached to this article on support.ca.com under TEC542192.

IDMS supports joins of network records by means of the *set-specification* statement of the WHERE parameter, for example:

```
SELECT E.*, C.* FROM EMPLOYEE E, COVERAGE C WHERE "EMP-COVERAGE"
```

The WHERE parameter in this case causes the EMP-COVERAGE set to be navigated, obtaining each EMPLOYEE record (Employee.java in the examples) and its associated COVERAGE record(s). The result is the same as an inner join. Within JPA, a query like this must be defined as a *native query*. This should be defined in the entity class for the EMPLOYEE record using a @NamedNativeQuery annotation as shown in the excerpt below.

```
@NamedNativeQuery(
     name="GetEmpCoverageOpt",
     query="SELECT e.*, c.*, c.ROWID FROM " +
        "EMPSCHM.EMPLOYEE e, EMPSCHM.COVERAGE c " +
        "WHERE EMP_ID_0415 = :empID and \"EMP-COVERAGE\"",
        resultSetMapping="EmpCoverageResultOpt"
```

Here we've added a couple of additional parameters to the basic Select statement: 1) an additional test in the WHERE parameter to specify a single EMPLOYEE record occurrence and its related COVERAGE occurrence(s), as determined by the ":empID" named parameter; 2) a reference to the ROWID element of the COVERAGE record. The latter allows JPA to uniquely identify a COVERAGE record occurrence even if its business data is not unique. As you can guess from the example, you must explicitly include ROWID in the column projection list if you want it; "c.*" will retrieve only the data elements.

ROWID must also be defined as a primary key in the entity class for the COVERAGE record (Coverage.java in the examples) as shown below. Note that the definition of RowId as a byte array works with Hibernate but not OpenJPA; see "OpenJPA Considerations" below for details.

```
private byte[] RowId;
…
public Coverage(byte[] RowId, …) {
        this.RowId = RowId;
…
@Id
@Column(name="ROWID", nullable=false, precision=4, scale=0)
public byte[] getRowId() {
    return this.RowId;
}

public void setRowId(byte[] RowId) {
    this.RowId = RowId;
}
```

The result set must be mapped by a SQLResultSetMapping annotation which specifies the entity classes that should be populated by JPA. In our example, this is specified in the Employee entity class, as follows:

```
@SqlResultSetMapping(name="EmpCoverageResultOpt",
     entities={
          @EntityResult(entityClass=Employee.class),
          @EntityResult(entityClass=Coverage.class)})
```

Within the business logic, we acquire an EntityManagerFactory referencing the persistence unit name (NonSqlJPA) defined in the persistence.xml file and acquire an EntityManager object. Then we instantiate the native query, set its named parameter, execute the query and retrieve the result set as follows:

```
EntityManagerFactory emf =
     Persistence.createEntityManagerFactory("NonSqlJPA");
EntityManager em = emf.createEntityManager();

Iterator<?> pairs = em.createNamedQuery("GetEmpCoverageOpt")
     .setParameter("empID", 23)
     .getResultList()
     .iterator();

Employee e = null;
Coverage c = null;

while (pairs.hasNext()) {
     Object[] pair = (Object[]) pairs.next();
     e = (Employee) pair[0];
     c = (Coverage) pair[1];
     …
```

After the query is executed its result set is mapped to a java.util.List, each element of which comprises an Employee entity instance and a matching Coverage entity instance. If the selected EMPLOYEE record occurrence has more than one COVERAGE record occurrence in the set, there will be more than one element in the list but each would point to the same Employee object. Finally, the List is then returned as an Iterator to make it easy to walk through its elements. The result of this is that we have references to an Employee object which can be uniquely identified by its primary key, empId0415, and references to one or more Coverage objects, each uniquely identified by RowId.

Updating a Record Using RowId

With this in mind, let's next see how we could update a Coverage object. The inclusion of RowId makes this a relatively simple matter:

```
EntityTransaction tx = em.getTransaction();
tx.begin();
…
while (pairs.hasNext()) {
     Object[] pair = (Object[]) pairs.next();
```

```
     e = (Employee) pair[0];
     c = (Coverage) pair[1];
     …
          c.setType0400('M');      // update one column
}
try {
     tx.commit();
}
```

All we need to do is invoke a "setter" method on the Coverage object; in this case it's setType0400. The JPA provider will keep track of this and, assuming that a value of 'M' is different than the value that already existed in the object, will write the changed data back to IDMS when the tx.commit() method is called.

Considerations for Concurrency and Locking

Applications using JPA generally use *optimistic locking,* a concurrency control method based on the assumption that there is very little risk of conflicting updates among concurrently executing database transactions. Optimistic locking employs a versioning technique to detect the possibility of lost updates, and gives the programmer an opportunity to take corrective action.

Versioning can be done in one of two ways – by means of a version column such as a sequence number or timestamp, or by checking all columns of a row to make sure that its values have not been changed. Hibernate supports both techniques, the latter of which is of great value to programmers accessing IDMS legacy data which cannot be easily changed to add a version column. However, the JPA spec only addresses versioning using a version column, so a Hibernate-specific annotation must be used to specify versioning using all columns:

```
@org.hibernate.annotations.Entity(
     dynamicUpdate = true,
     optimisticLock =
     org.hibernate.annotations.OptimisticLockType.ALL)
```

This annotation is applied to an entity class (Coverage in our case). When it is present, Hibernate will update the entity in this manner:

```
update EMPSCHM.COVERAGE set TYPE_0400=?
where ROWID=? and INS_PLAN_CODE_0400=? and SELECTION_DAY_0400=?
and SELECTION_MONTH_0400=? and SELECTION_YEAR_0400=? and
TERMINATION_DAY_0400=? and TERMINATION_MONTH_0400=? and
TERMINATION_YEAR_0400=? and TYPE_0400=?
```

After executing this statement, Hibernate will check the JDBC return code to see if a row has been updated by this statement; if not, then it knows that the row has been updated by another transaction and will return an exception to the application. The application must then take steps to resolve the conflicting updates.

OpenJPA, unfortunately, is not as flexible in this regard, which may also be the case with other non-Hibernate JPA providers. OpenJPA's versioning technique is based solely on use of a version column defined in the table. Of course, IDMS users are free to add such a column, but this does require restructuring of the database and applications. However, there is another alternative for OpenJPA users. OpenJPA 2.0 added support for pessimistic locking, which may not scale as well as optimistic locking in some cases, but is the type of locking normally used by IDMS applications anyway. Note that Hibernate also supports pessimistic locking, but optimistic locking is usually the better choice.

With pessimistic locking, a row to be updated must first be locked:

```
if (c != null)
     em.lock(c, LockModeType.PESSIMISTIC_WRITE);
     c.setType0400('M');      // update one column
}
try {
     tx.commit();
```

The em.lock method call causes OpenJPA to re-fetch the row as follows:

```
SELECT t0.ROWID FROM EMPSCHM.COVERAGE t0
WHERE t0.ROWID = ? FOR UPDATE
```

The sole intent of this rather unusual statement is to lock the row and so it doesn't matter that ROWID cannot actually be updated. Note that the "FOR UPDATE" clause only influences the access plan within the IDMS SQL engine and does not affect locking; a read lock will always be placed on the current row (which is the only row in this case) regardless of the presence of "FOR UPDATE". However, IDMS Server does behave differently when the "FOR UPDATE" clause is specified. Normally, IDMS Server will issue a commit immediately after the row is fetched, releasing the read lock. Using "FOR UPDATE" causes the read lock to be left in place, preventing the row from being updated by another transaction. Note that "FOR UPDATE" also causes IDMS Server to fetch rows one at a time so that currency is maintained on

the requested row (although in our example there would be only a single row in the result set anyway).

After c.setType0400 is executed and we commit the transaction with tx.commit(), the update is done in this manner:

```
UPDATE EMPSCHM.COVERAGE SET TYPE_0400 = ? WHERE ROWID = ?
```

If this is successful, OpenJPA will issue a commit to apply the changes and release the lock(s).

Deleting and Inserting Network Records
Record deletion should work in the normal manner:

```
em.remove(c);
```

This method call results in the following SQL statement:

```
DELETE FROM EMPSCHM.COVERAGE WHERE ROWID = ?
```

Within IDMS, deletes will cause the record to be disconnected from sets and indexes of which it's a member. This is equivalent to ERASE REC in IDMS DML.

Inserts require records to contain foreign keys, which as we discussed previously is uncommon for IDMS network records. If foreign keys have not been implemented, then a SQL procedure of some sort can be used to insert records.

Recommended Software
The following software components and releases were used in testing the examples associated with this article.

- CA IDMS Server 17.0
- CA IDMS 17.0
- CA IDMS SQL Option
- Hibernate 3.2.5, with IDMSDialect provided as part of CA IDMS Server 17.0
- OpenJPA 2.0.0

OpenJPA Considerations

OpenJPA cannot support a use of a byte array as a primary key field, so ROWID (which is essentially defined as BINARY(8) in IDMS and normally used in Java as a byte array) must instead be defined in an entity class as a String. This technique also requires an IDMS-specific DBDictionary class to perform conversion between a String object and the ROWID native format. Currently no OpenJPA DBDictionary class is distributed with the IDMS suite of products. For purposes of completing this article, a very basic class named IDMSDictionary was written and has been included in the OpenJPA examples. Note that this class has not been tested except with the examples referenced in this article and no guarantees can be made regarding its successful use with other OpenJPA applications.