Performing Queries in Oracle Berkeley DB Direct Persistence Layer

An Oracle White Paper

ORACLE

Performing Queries in Oracle Berkeley DB Direct Persistence Layer

Oracle Berkeley DB Direct Persistence Layer is designed to offer the same benefits of Enterprise Java Beans 3.0 (EJB3) persistence without the overhead of translating objects into tables.

INTRODUCTION

Oracle Berkeley DB Java Edition (BDB-JE) as well as its sibling Oracle Berkeley DB (BDB) both are small footprint, transactional, embeddable, scalable, data storage engines, without the overhead required for client/server communication or SQL parsing, planning and execution. Still, you can perform queries in both.

For Java developers who may approach BDB-JE and/or BDB with past SQL experience, this white paper illustrates the data model and API by translating some of the most common SQL queries to their equivalent form using Oracle Berkeley DB's Direct Persistence Layer API (DPL) which is available in both BDB-JE and BDB. The DPL offers many of the same benefits of Java Persistence APIs that perform object-to-relational mapping within Enterprise Java Beans 3.0 (EJB3), but without the need to translate objects into SQL expressions persisted in relational tables. This is accomplished without losing any of the transactional (ACID), concurrent, or scalability qualities provided by RDBMS systems. Please refer to the *Translating SQL Queries Example* in BDB-JE Installation Notes found in the documentation online and included within the BDB-JE distribution (http://www.oracle.com/technology/documentation/berkeley-db/je/installation.html) for more detailed information and the entire example program.

TERMINOLOGY SIDE-BY-SIDE: DPL AND SQL

To understand Oracle Berkeley DB Java Edition and Oracle Berkeley DB it helps to understand how common concepts are expressed. Some of the terminology differs from standard RDBMS, but the concepts are the same.

Here are some common SQL terms and their equivalents in DPL:

SQL Term	Oracle DPL Equivalent
Database	Environment
Table	Primary index and its related secondary indices

Primary index	Primary index
Secondary index	Secondary index
Tuple/row	Entity

Basic Persistence Models

In Oracle Berkeley DB Java Edition and Oracle Berkeley DB, records are represented as key/data pairs. The keys and data are stored as sequences of bytes. Environments exist as a set of on-disk files within one or more directories. An in-memory cache stores active parts of the indices and frequently accessed data. There are two APIs that allow access to data for basic create, read, update and delete (CRUD) operations. The *Base* API available in both Oracle Berkeley DB Java Edition and Oracle Berkeley DB allows programmers to manage opaque key/data pairs (byte arrays), their encoding, and any relationships directly. In some cases this is all that is required; simple lookup tables or other simplistic storage requirements can quickly store and retrieve information using this simple put/get/cursor based API. This API is intended for use cases where the schema is not tied to a set of predefined Java classes but instead managed by the application. For example, in an implementation of an LDAP directory server or a JINI JavaSpaces storage container, the schema is simplistic and user-defined, very nearly identical to a basic key/data layout with well known encoding. For such applications, our Base API provides a low-level, but useful interface for storing data.

The Direct Persistence Layer, an Advanced Entity Persistence Model

The Direct Persistence Layer is a layer on top of the byte array Base API. It is a higher level abstraction that manages the marshaling, indexing, and access to graphs of Java objects stored as byte arrays in the database files. It also provides a type-safe interface based on the Java classes using that layout as the database schema. Using the DPL, an entity class is an ordinary Java class (plain old Java object - POJO) that has a primary key and is stored and accessed using a primary index. An entity class may also have any number of secondary keys, and a secondary index may be used to access entities by secondary key. A set of entity classes constitutes an entity model. In addition to isolated entity classes, an entity model may contain relationships between entities. Many-to-one, one-to-many, many-to-many and one-to-one relationships (M:1, 1:M, M:M, and 1:1) are supported including foreign key constraints on these relationships.

Persistent Java Collections

In addition to the Base API and the DPL, the standard Java Collections interfaces can be used for persistent data access. The Persistent Collections API provides a

standard API that is familiar to all Java programmers and enables interoperability with any component that uses these Java interfaces. The Persistent Collections API can be used along with the base API or the DPL.

An Example Schema

The sample schema below defines two entity classes: the `Employee` class and the `Department` class. Their DPL-annotated definitions are shown below.

```
@Entity
class Employee {

    @PrimaryKey
    int employeeId;

    /* Many Employees may have the same name. */
    @SecondaryKey(relate=MANY_TO_ONE)
    String employeeName;

    /* Many Employees may have the same salary. */
    @SecondaryKey(relate=MANY_TO_ONE)
    float salary;

    @SecondaryKey(relate=MANY_TO_ONE,
                  relatedEntity=Employee.class,
                  onRelatedEntityDelete=NULLIFY)
    Integer managerId; // Use "Integer" to allow null
                       // values.

    @SecondaryKey(relate=MANY_TO_ONE,
                  relatedEntity=Department.class,
                  onRelatedEntityDelete=NULLIFY)
    Integer departmentId;

    String address;

    public Employee(int employeeId,
                    String employeeName,
                    float salary,
                    Integer managerId,
                    int departmentId,
                    String address) {
        this.employeeId = employeeId;
        this.employeeName = employeeName;
        this.salary = salary;
        this.managerId = managerId;
        this.departmentId = departmentId;
        this.address = address;
    }

    private Employee() {} // Needed for deserialization
}

@Entity
class Department {

    @PrimaryKey
    int departmentId;
```

```
    @SecondaryKey(relate=ONE_TO_ONE)
    String departmentName;

    String location;

    public Department(int departmentId,
                      String departmentName,
                      String location) {
        this.departmentId = departmentId;
        this.departmentName = departmentName;
        this.location = location;
    }

    private Department() {} // Needed for deserialization
}
```

Entity classes whose instances are stored and retrieved separately and have a unique primary key are annotated `@Entity`. The primary key field is denoted with the `@PrimaryKey` annotation, akin to the `@Id` annotation in Java Persistence API (JPA). The DPL `@SecondaryKey` annotation is similar to JPA's `@OneToOne`, `@ManyToOne`, etc. Using the DPL `@SecondaryKey` will create a secondary index whereas with JPA's equivalent annotations (`@OneToOne`, `@ManyToOne`, etc) simply define the association to another entity class.

Preload Sample Records

When using the DPL, data is always accessed through an index. The primary and secondary keys defined above map directly to their primary and secondary indices.

```
/* Employee Accessors */
PrimaryIndex<Integer, Employee>
    employeeById;
SecondaryIndex<String, Integer, Employee>
    employeeByName;
SecondaryIndex<Float, Integer, Employee>
    employeeBySalary;
SecondaryIndex<Integer,Integer,Employee>
    employeeByManagerId;
SecondaryIndex<Integer,Integer,Employee>
    employeeByDepartmentId;

/* Department Accessors */
PrimaryIndex<Integer, Department>
    departmentById;
SecondaryIndex<String, Integer, Department>
    departmentByName;

/* Primary index of the Employee database. */
employeeById =
    store.getPrimaryIndex(Integer.class,
                          Employee.class);

/* Secondary index of the Employee database. */
employeeByName =
    store.getSecondaryIndex(employeeById,
```

```
                                String.class,
                                "employeeName");
employeeBySalary =
    store.getSecondaryIndex(employeeById,
                                Float.class,
                                "salary");
employeeByManagerId =
    store.getSecondaryIndex(employeeById,
                                Integer.class,
                                "managerId");
employeeByDepartmentId =
    store.getSecondaryIndex(employeeById,
                                Integer.class,
                                "departmentId");

/* Primary index of the Department database. */
departmentById =
    store.getPrimaryIndex(Integer.class,
                          Department.class);

/* Secondary index of the Department database. */
departmentByName =
    store.getSecondaryIndex(departmentById,
                                String.class,
                                "departmentName");
```

Once all indexes are initialized, sample records can be created like this:

```
/* Load the Department database. */
private void loadDepartmentDb()
    throws DatabaseException {

    departmentById.put
        (new Department(1, "CEO Office", "North
America"));
    departmentById.put
        (new Department(2, "Sales", "EURO"));
    departmentById.put
        (new Department(3, "HR", "MEA"));
    departmentById.put
        (new Department(4, "Engineering", "APAC"));
    departmentById.put
        (new Department(5, "Support", "LATAM"));
}

/* Load the Employee database. */
private void loadEmployeeDb()
    throws DatabaseException {

    /*
     * Add the corporate's CEO using the Employee
     * primary index.
     */
    employeeById.put
        (new Employee(1, // employeeId
                      "Abraham Lincoln", //employeeName
                      10000.0f, //salary
                      null, //managerId
```

```
                    1, //departmentId
                    "Washington D.C., USA")); //address

    /* Add 4 managers responsible for 4 departments. */
    employeeById.put
        (new Employee(2,
                    "Augustus",
                    9000.0f,
                    1,
                    2,
                    "Rome, Italy"));
    employeeById.put
        (new Employee(3,
                    "Cleopatra",
                    7000.0f,
                    1,
                    3,
                    "Cairo, Egypt"));
    employeeById.put
        (new Employee(4,
                    "Confucius",
                    7500.0f,
                    1,
                    4,
                    "Beijing, China"));
    employeeById.put
        (new Employee(5,
                    "Toussaint Louverture",
                    6800.0f,
                    1,
                    5,
                    "Port-au-Prince, Haiti"));

    /* Add 2 employees per department. */
    employeeById.put
        (new Employee(6,
                    "William Shakespeare",
                    7300.0f,
                    2,
                    2,
                    "London, England"));
    employeeById.put
        (new Employee(7,
                    "Victor Hugo",
                    7000.0f,
                    2,
                    2,
                    "Paris, France"));
    employeeById.put
        (new Employee(8,
                    "Yitzhak Rabin",
                    6500.0f,
                    3,
                    3,
                    "Jerusalem, Israel"));
    employeeById.put
        (new Employee(9,
                    "Nelson Rolihlahla Mandela",
                    6400.0f,
```

```
                        3,
                        3,
                        "Cape Town, South Africa"));
    employeeById.put
        (new Employee(10,
                        "Meiji Emperor",
                        6600.0f,
                        4,
                        4,
                        "Tokyo, Japan"));
    employeeById.put
        (new Employee(11,
                        "Mohandas Karamchand Gandhi",
                        7600.0f,
                        4,
                        4,
                        "New Delhi, India"));
    employeeById.put
        (new Employee(12,
                        "Ayrton Senna da Silva",
                        5600.0f,
                        5,
                        5,
                        "Brasilia, Brasil"));
    employeeById.put
        (new Employee(13,
                        "Ronahlinho De Assis Moreira",
                        6100.0f,
                        5,
                        5,
                        "Brasilia, Brasil"));
}
```

Translating SQL Queries

Oracle Berkeley DB Direct Persistence Layer queries are implemented procedurally in Java rather than using a query language. Using SQL, a query processor evaluates your query and decides how to implement it internally using indices. Using the DPL, you make the decisions about how to implement the query and use indices explicitly. This takes little work and gives you complete control over the performance of the query and does not require a non-Java query language.

This example translates some common SQL queries to the DPL including simple data retrieval and prefix, range, two equality-conditions and equi-join queries. Because of the DPL's flexible interface, most of the approaches shown here can be easily adapted for data modification operations with only minor changes. Refer to the `com.sleepycat.persist` Javadoc for BDB-JE (http://www.oracle.com/technology/documentation/berkeley-db/je/java/com/sleepycat/persist/package-summary.html) or for BDB (http://www.oracle.com/technology/documentation/berkeley-db/db/java/com/sleepycat/persist/package-summary.html) for further information.

- **Simple Data Retrieval** –

```
SELECT * FROM tab ORDER BY col ASC;
```

Simple data retrieval can be achieved by calling the DPL method `EntityIndex.entities()` which returns an `EntityCursor` object. This allows you to traverse entity or key values as well as deleting or updating the entity at the current cursor position. Note that `EntityCursor` objects are *not* thread-safe so they should be opened, used, and closed by a single thread.

For example, to print all `Department` records, ordered by `departmentId`:

```
/* SELECT * FROM department ORDER BY departmentId; */
EntityCursor<Department> deptCursor =
    departmentById.entities();
try {
   /*
    * Walk an EntityCursor over the Department database.
    */
    for (Department dept : deptCursor) {
        System.out.println(dept);
    }
    System.out.println();
} finally {
    deptCursor.close();
}
```

To retrieve all `Department` records, ordered by `departmentName`:

```
/* SELECT * FROM department ORDER BY departmentName; */
EntityCursor<Department> c =
    departmentByName.entities();
```

- **Prefix Query** –

```
SELECT * FROM tab WHERE col LIKE 'prefix%';
```

Use an `EntityIndex` in the following manner to perform a prefix query:

```
EntityIndex.entities(K fromKey,
                     boolean fromInclusive,
                     K toKey,
                     boolean toInclusive)
```

This will open a cursor that can then be used to traverse entities in a key range. See the Javadoc (http://www.oracle.com/technology/documentation/berkeley-db/je/java/com/sleepycat/persist/EntityIndex.html or http://www.oracle.com/technology/documentation/berkeley-db/db/java/com/sleepycat/persist/EntityIndex.html) for several similar methods.

For example, to find those employees whose name starts with a given prefix:

```
/*
 * SELECT * FROM employee
 * WHERE employeeName LIKE 'prefix%';
 */
```

```
char[] ca = prefix.toCharArray();
final int lastCharIndex = ca.length - 1;
ca[lastCharIndex]++;
return employeeByName.entities(prefix, true,
                               String.valueOf(ca),
                               false);
```

- **Range Query** –

```
SELECT * FROM tab WHERE col >= A AND col <= B;
```

For a range query, you can again use the `EntityIndex.entities()` method. As an example, to find all employees with salary between 6,000 and 8,000:

```
/*
 * SELECT * FROM employee
 * WHERE salary >= 6000 AND salary <= 8000;
 */
employeeBySalary.entities(new Float(6000), //fromKey
                          true, //fromInclusive
                          new Float(8000), //toKey
                          true);
```

- **Two equality-conditions query on a single primary database** –

```
SELECT * FROM tab WHERE col1 = A AND col2 = B;
```

A query specifying two equality-conditions is a match on all entities in a given primary index that have two or more specific secondary key values. The `EntityJoin` class is used to perform such a query on two or more secondary keys. Once instantiated, it is easy to add a secondary key condition to perform the query.

```
addCondition(SecondaryIndex<SK, PK, E> index, SK key)
```

The `EntityJoin` class can be used to perform an equality-conditions query on a single primary database:

```
/*
 * SELECT * FROM employee
 * WHERE employeeName = 'key1' AND departmentId = key2;
 */
public <PK, SK1, SK2, E> ForwardCursor<E>
   doTwoConditionsJoin(PrimaryIndex<PK, E> pk,
                       SecondaryIndex<SK1, PK, E> sk1,
                       SK1 key1,
                       SecondaryIndex<SK2, PK, E> sk2,
                       SK2 key2)
   throws DatabaseException {

   assert (pk != null);
   assert (sk1 != null);
   assert (sk2 != null);
```

```
    EntityJoin<PK, E> join = new EntityJoin<PK, E>(pk);
    join.addCondition(sk1, key1);
    join.addCondition(sk2, key2);

    return join.entities();
}
```

The `EntityJoin` class can only be used to perform an equality-conditions query for a single primary index. It may be not be used to join two or more primary indices. Joining multiple primary indices is discussed in the next section.

- **An equi-join on two primary databases** –

```
SELECT t1.* FROM table1 t1, table2 t2

WHERE t1.col1 = t2.col1 AND t2.col2 = A;
```

Let's say we want to find all the employees that belong to a given department and we know the department name, but not the department ID.

This query can be implemented as a simple lookup by department name, followed by a loop that iterates over all employees in that department. Both operations are fast, because they both use a secondary index.

```
/*
 * SELECT * FROM employee e, department d
 * WHERE e.departmentId = d.departmentId
 * AND d.departmentName = 'deptName';
 */
Department dept = departmentByName.get(deptName);
EntityCursor<Employee> empCursor =
    employeeByDepartmentId.
    subIndex(dept.getDepartmentId()).entities();
try {
    /* Do an inner join on Department and Employee. */
    for (Employee emp : empCursor) {
        System.out.println(emp);
    }
} finally {
    empCursor.close();
}
```

The `subIndex` method that is used above allows iterating over all employees for a given department ID. This method is generally useful for all secondary indices that represent many-to-one and many-to-many relationships. See the `SecondaryIndex` class Javadoc (http://www.oracle.com/technology/documentation/berkeley-db/je/java/com/sleepycat/persist/SecondaryIndex.html or http://www.oracle.com/technology/documentation/berkeley-db/db/java/com/sleepycat/persist/SecondaryIndex.html) for more information.

It is often useful to think about multiple approaches for implementing a query, since not all approaches will perform equally well. When using SQL, this

determination is made internally by the SQL query optimizer and what happens under the covers is not always transparent. With the DPL, you explicitly decide on the approach for implementing a query.

For example, a second possible approach for implementing the query above is to iterate over all employees, retrieve the department record for each employee, and examine the department name. If the department name matches the one you're interested in, you can print the employee.

Hopefully it is obvious that this second approach will perform much more poorly than the first approach shown above, because all employee records and all department records must be retrieved, even if they are not selected by the query. And each department record is retrieved multiple times -- once for each of its employees. With the first approach shown above, the only records retrieved are those that are selected by the query.

- **Filtering along with an equi-join** –

```
SELECT t1.* FROM table1 t1, table2 t2

WHERE t1.col1 = t2.col1 AND t2.col2 = A;
```

Often the need arises to query by fields that are not indexed. For example, you may wish to print all employees at a particular department location. Let's imagine that this query is performed infrequently, and therefore you don't want the overhead of a secondary index on the location field. This matches our schema, where the location field is not indexed.

In this case, you must iterate over all departments and select (filter) only those departments with the location in which you are interested. Then, using the secondary index on department ID, you can use the `subIndex` method to find all employees in that department.

This is an example of combining filtering with the use of an index. Once again, this is something that a SQL query engine may do internally when an appropriate index is not available to satisfy the conditions of the query. With the DPL, you explicitly use filtering in this situation.

```
/*
 * SELECT * FROM employee e, department d
 * WHERE e.departmentId = d.departmentId
 * AND d.location = 'deptLocation';
 */
public void getEmployeeByDeptLocation(String
deptLocation)
    throws DatabaseException {

    /* Iterate over Department database. */
    Collection<Department> departments =
         departmentById.sortedMap().values();
    for (Department dept : departments) {
        if (dept.getLocation().equals(deptLocation)) {
            /* A nested loop to do an equi-join. */
```

```
            EntityCursor<Employee> empCursor =
                employeeByDepartmentId.
                subIndex(dept.getDepartmentId()).
                entities();
            try {
                /* Iterate over all employees in dept. */
                for (Employee emp : empCursor) {
                    System.out.println(emp);
                }
            } finally {
                empCursor.close();
            }
        }
    }
}
```

The `EntityIndex.sortedMap()` returns a `java.util.SortedMap` based on this entity index, which may be passed to other components which accept a `Map` argument.

Oracle Berkeley DB and Oracle Berkeley DB Java Edition provide a mature solution for data storage in a small and manageable package with a programmer friendly, yet powerful API.

CONCLUSION

Oracle Berkeley DB Java Edition and Oracle Berkeley DB both are open source, embeddable, transactional storage engines. The Direct Persistence Layer (DPL) available in both is a simple and effective way to persist Java objects. Despite its simplicity BDB/BDB-JE provides full transactional (ACID) support with recovery, it can scale to store terabytes of data and be concurrently accessed by thousands of Java threads. Oracle Berkeley DB Java Edition and Oracle Berkeley DB's Direct Persistence Layer provides many of the same benefits as EJB3 and the Java Persistence API (JPA) without the overhead associated with object-to-relational mapping (ORM), query planning, processing, and execution over client/server connections. Oracle Berkeley DB Java Edition and Oracle Berkeley DB provide a mature solution for data storage in a small and manageable package with a programmer friendly, yet powerful API.

Choosing between the JPA or the DPL depends on many things related to application architecture.

- Both the JPA and the DPL make it easy to persist Java objects using Java class annotations. However with the DPL, no table mappings are necessary.
- With the DPL, queries are expressed as Java code rather than as SQL statements to be processed by an RDBMS.
- The DPL provides complete control over the execution and performance of the query, while the JPA and SQL support the execution of ad-hoc queries at run-time.
- Because JE is an embedded database, no database server installation or administration is required to use the DPL.

You can download Berkeley DB and Oracle Berkeley DB Java Edition at:

http://www.oracle.com/database/berkeley-db/

The Oracle Technology Network (OTN) forums for Berkeley DB (http://forums.oracle.com/forums/forum.jspa?forumID=271) and for Berkeley DB Java Edition (http://forums.oracle.com/forums/forum.jspa?forumID=273) are the preferred places for comments and questions.

Performing Queries in Oracle Berkeley DB Direct Persistence Layer

Author: Chao Huang

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