

Berkeley DB Java Edition Direct Persistence Layer Basics

*An Oracle White Paper
September 2006*

Berkeley DB Java Edition

Direct Persistence Layer Basics

OVERVIEW

Berkeley DB Java Edition is an embeddable database implemented in pure Java. It provides a transactional storage engine that reduces the overhead of object persistence dramatically while keeping the flexibility, speed and scalability of object to relational mapping (ORM) solutions. Berkeley DB Java Edition 3.0 introduces a major new interface called the Direct Persistence Layer (DPL). This API is designed to offer the same benefits of Enterprise Java Beans 3.0 (EJB3) persistence without the need to translate objects into tables.

Relational databases are the most sophisticated tool available to the developer for data storage and analysis. Most persisted object data is never analyzed using ad hoc SQL queries other than for purposes of retrieval for reconstitution as Java objects. The overhead of using a sophisticated analytical storage engine is wasted on this basic task of object retrieval. The analytical power of the relational model isn't required to persist Java objects efficiently. In most cases it is unnecessary overhead. In contrast, Berkeley DB Java Edition does not provide an ad hoc query language like SQL, and so does not incur this penalty. The result is faster storage, lower CPU and memory requirements, and a more efficient development process. Despite the lack of an ad hoc query language, Berkeley DB Java Edition can access Java objects in an ad hoc manner and it does provide the same fast, reliable and scalable data storage as any other database. The difference is that it does this in a small, efficient, and easy to manage package.

Using the new Direct Persistence Layer Java developers can now persist and retrieve inter-related groups of Java objects with confidence, speed, and a fraction of the complexity and overhead of a comparable ORM solution.

This document will review this new API by introducing the major concepts within the context of a simple example application.

INTRODUCTION

The Direct Persistence Layer is designed to meet the following requirements.

A type safe and convenient API is provided for accessing persistent objects.

The use of Java generic types, although optional, is fully exploited to provide type safety. For example:

```
PrimaryIndex<Long,Employer> employerById = ...;
long employerId = ...;
Employer employer = employerById.get(employerId);
```

All Java types are allowed to be persistent without requiring that they implement special interfaces.

Persistent fields may be private, package-private (default access), protected, or public. No hand-coding of bindings is required.

However, each persistent class must have a default constructor. For example:

```
@Persistent
class Address {
    String street;
    String city;
    String state;
    int zipCode;
    private Address() {}
}
```

It is easy to define primary and secondary keys. No external schema is required and Java annotations may be used for defining all metadata. Extensions may derive metadata from other sources. For example, the following `Employer` class is defined as a persistent entity with a primary key field `id` and the secondary key field `name`:

```
@Entity
class Employer {

    @PrimaryKey(sequence="ID")
    long id;

    @SecondaryKey(related=ONE_TO_ONE)
    String name;

    Address address;

    private Employer() {}
}
```

Interoperability with external components is supported via the Java collections framework. Any primary or secondary index can be accessed using a standard `java.util` collection. For example:

```
java.util.SortedMap<String,Employer> map =
    employerByName.sortedMap();
```

Class evolution is explicitly supported. Compatible changes (adding fields and type widening) are performed automatically and transparently. For example, without any special configuration a `street2` field may be added to the `Address` class and the type of the `zipCode` field may be changed from `int` to `long`:

```
@Persistent
class Address {
    String street;
    String street2;
    String city;
    String state;
    long zipCode;
    private Address() {}
}
```

Many incompatible class changes, such as renaming fields or re-factoring a single class, can be performed using `Mutations`. `Mutations` are automatically applied lazily as data is accessed, avoiding downtime to convert large databases during a software upgrade.

Complex re-factoring involving multiple classes may be performed using the `ConversionStore`. The Direct Persistence Layer always provides access to your data, no matter what changes have been made to persistent classes.

The performance of the Berkeley DB transactional engine is not compromised. Operations are internally mapped directly to the engine API, object bindings are lightweight, and all engine-tuning parameters are available. For example, a "dirty read" may be performed using an optional `LockMode` parameter:

```
Employer employer = employerByName.get(null, "Gizmo Inc",
                                         LockMode.READ_UNCOMMITTED);
```

For high performance applications, `DatabaseConfig` parameters may be used to tune the performance of the Berkeley DB engine. For example, the size of an internal Btree node can be specified as follows:

```
DatabaseConfig config = store.getPrimaryConfig(Employer.class);
config.setNodeMaxEntries(64);
store.setPrimaryConfig(config);
```

THE ENTITY MODEL

The Direct Persistence Layer is intended for applications that represent persistent domain objects using Java classes. An entity class is an ordinary Java class that has a primary key and is stored and accessed using a primary index. It may also have any number of secondary keys, and entities may be accessed by secondary key using a secondary index.

An entity class may be defined with the `Entity` annotation. For each entity class, its primary key may be defined using the `PrimaryKey` annotation and any number of secondary keys may be defined using the `SecondaryKey` annotation.

In the following example, the `Person.ssn` (social security number) field is the primary key and the `Person.employerIds` field is a many-to-many secondary key.

```
@Entity
class Person {

    @PrimaryKey
    String ssn;

    String name;
    Address address;

    @SecondaryKey(related=MANY_TO_MANY,
                 relatedEntity=Employer.class)
    Set<Long> employerIds = new HashSet<Long>();

    private Person() {} // For bindings
}
```

A set of entity classes constitutes an entity model. In addition to isolated entity classes, an entity model may contain relationships between entities. Relationships may be defined using the `SecondaryKey` annotation. Many-to-one, one-to-many, many-to-many and one-to-one relationships are supported, as well as foreign key constraints.

In the example above, a relationship between the `Person` and `Employer` entities is defined via the `Person.employerIds` field. The `relatedEntity=Employer.class` annotation property establishes foreign key constraints to guarantee that every element of the `employerIds` set is a valid `Employer` primary key.

For more information on the entity model, see the `AnnotationModel` and the `Entity` annotation.

The root object in the Direct Persistence Layer is the `EntityStore`. An entity store manages any number of objects for each entity class defined in the model. The store provides access to the primary and secondary indices for each entity class, for example:

```
EntityStore store = new EntityStore(...);

PrimaryIndex<String, Person> personBySsn =
    store.getPrimaryIndex(String.class, Person.class);
```

A BRIEF EXAMPLE

The following example shows how to define an entity model and how to store and access persistent objects. Exception handling is omitted for brevity.

```
import java.io.File;
import java.util.HashSet;
import java.util.Set;

import com.sleepycat.je.DatabaseException;
import com.sleepycat.je.Environment;
import com.sleepycat.je.EnvironmentConfig;
import com.sleepycat.persist.EntityCursor;
import com.sleepycat.persist.EntityIndex;
import com.sleepycat.persist.EntityStore;
import com.sleepycat.persist.PrimaryIndex;
import com.sleepycat.persist.SecondaryIndex;
import com.sleepycat.persist.StoreConfig;
import com.sleepycat.persist.model.Entity;
import com.sleepycat.persist.model.Persistent;
import com.sleepycat.persist.model.PrimaryKey;
import com.sleepycat.persist.model.SecondaryKey;
import static com.sleepycat.persist.model.DeleteAction.NULLIFY;
import static com.sleepycat.persist.model.Relationship.ONE_TO_ONE;
import static com.sleepycat.persist.model.Relationship.ONE_TO_MANY;
import static com.sleepycat.persist.model.Relationship.MANY_TO_ONE;
import static com.sleepycat.persist.model.Relationship.MANY_TO_MANY;

// An entity class.
//
@Entity
class Person {

    @PrimaryKey
    String ssn;

    String name;
    Address address;

    @SecondaryKey(related=MANY_TO_ONE, relatedEntity=Person.class)
    String parentSsn;

    @SecondaryKey(related=ONE_TO_MANY)
    Set<String> emailAddresses = new HashSet<String>();

    @SecondaryKey(related=MANY_TO_MANY, relatedEntity=Employer.class,
                  onRelatedEntityDelete=NULLIFY)
    Set<Long> employerIds = new HashSet<Long>();
    Person(String name, String ssn, String parentSsn) {
        this.name = name;
    }
}
```

```

        this.ssn = ssn;
        this.parentSsn = parentSsn;
    }

    private Person() {} // For bindings
}

// Another entity class.
//
@Entity
class Employer {

    @PrimaryKey(sequence="ID")
    long id;

    @SecondaryKey(related=ONE_TO_ONE)
    String name;

    Address address;

    Employer(String name) {
        this.name = name;
    }

    private Employer() {} // For bindings
}

// A persistent class used in other classes.
//
@Persistent
class Address {
    String street;
    String city;
    String state;
    int zipCode;
    private Address() {} // For bindings
}

// The data accessor class for the entity model.
//
class PersonAccessor {

    // Person accessors
    //
    PrimaryIndex<String,Person> personBySsn;
    SecondaryIndex<String,String,Person> personByParentSsn;
    SecondaryIndex<String,String,Person> personByEmailAddresses;
    SecondaryIndex<Long,String,Person> personByEmployerIds;

    // Employer accessors
    //
    PrimaryIndex<Long,Employer> employerById;

```

```

SecondaryIndex<String,Long,Employer> employerByName;

// Opens all primary and secondary indices.
//
public PersonAccessor(EntityStore store)
    throws DatabaseException {

    personBySsn = store.getPrimaryIndex(
        String.class, Person.class);

    personByParentSsn = store.getSecondaryIndex(
        personBySsn, String.class, "parentSsn");

    personByEmailAddresses = store.getSecondaryIndex(
        personBySsn, String.class, "emailAddresses");

    personByEmployerIds = store.getSecondaryIndex(
        personBySsn, Long.class, "employerIds");

    employerById = store.getPrimaryIndex(
        Long.class, Employer.class);

    employerByName = store.getSecondaryIndex(
        employerById, String.class, "name");
}
}

// Open a transactional Berkeley DB engine environment.
//
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
envConfig.setTransactional(true);
Environment env = new Environment(new File("/my/data"), envConfig);

// Open a transactional entity store.
//
StoreConfig storeConfig = new StoreConfig();
storeConfig.setAllowCreate(true);
storeConfig.setTransactional(true);
EntityStore store = new EntityStore(env, "PersonStore", storeConfig);

// Initialize the data access object.
//
PersonAccessor dao = new PersonAccessor(store);

// Add a parent and two children using the Person primary index.  Specifying a
// non-null parentSsn adds the child Person to the sub-index of children for
// that parent key.
//
dao.personBySsn.put(new Person("Bob Smith", "111-11-1111", null));
dao.personBySsn.put(new Person("Mary Smith", "333-33-3333", "111-11-1111"));
dao.personBySsn.put(new Person("Jack Smith", "222-22-2222", "111-11-1111"));

```

```

// Print the children of a parent using a sub-index and a cursor.
//
EntityCursor<Person> children =
    dao.personByParentSsn.subIndex("111-11-1111").entities();
try {
    for (Person child : children) {
        System.out.println(child.ssn + ' ' + child.name);
    }
} finally {
    children.close();
}

// Get Bob by primary key using the primary index.
//
Person bob = dao.personBySsn.get("111-11-1111");
assert bob != null;

// Create two employers. Their primary keys are assigned from a sequence.
//
Employer gizmoInc = new Employer("Gizmo Inc");
Employer gadgetInc = new Employer("Gadget Inc");
dao.employerById.put(gizmoInc);
dao.employerById.put(gadgetInc);

// Bob has two jobs and two email addresses.
//
bob.employerIds.add(gizmoInc.id);
bob.employerIds.add(gadgetInc.id);
bob.emailAddresses.add("bob@bob.com");
bob.emailAddresses.add("bob@gmail.com");

// Update Bob's record.
//
dao.personBySsn.put(bob);

// Bob can now be found by both email addresses.
//
bob = dao.personByEmailAddresses.get("bob@bob.com");
assert bob != null;
bob = dao.personByEmailAddresses.get("bob@gmail.com");
assert bob != null;

// Bob can also be found as an employee of both employers.
//
EntityIndex<String,Person> employees;
employees = dao.personByEmployerIds.subIndex(gizmoInc.id);
assert employees.contains("111-11-1111");
employees = dao.personByEmployerIds.subIndex(gadgetInc.id);
assert employees.contains("111-11-1111");

// When an employer is deleted, the onRelatedEntityDelete=NULLIFY for the

```

```
// employerIds key causes the deleted ID to be removed from Bob's employerIds.  
//  
dao.employerById.delete(gizmoInc.id);  
bob = dao.personBySsn.get("111-11-1111");  
assert !bob.employerIds.contains(gizmoInc.id);  
  
store.close();  
env.close();
```

The preceding example illustrates several characteristics of the Direct Persistence Layer:

Persistent data and keys are defined in terms of instance fields. For brevity the example does not show getter and setter methods, although these would normally exist to provide encapsulation. The Direct Persistence Layer accesses fields during object serialization and deserialization, rather than calling getter/setter methods, leaving business methods free to enforce arbitrary validation rules. For example:

```
@Persistent
public class ConstrainedValue {

    private int min;
    private int max;
    private int value;

    private ConstrainedValue() {} // For bindings

    public ConstrainedValue(int min, int max) {
        this.min = min;
        this.max = max;
        value = min;
    }

    public setValue(int value) {
        if (value < min || value > max) {
            throw new IllegalArgumentException("out of range");
        }
        this.value = value;
    }
}
```

The above `setValue` method would not work if it were called during object deserialization, since the order of setting fields is arbitrary. The `min` and `max` fields may not be set before the value is set.

The example creates a transactional store and therefore all operations are transaction protected. Because no explicit transactions are used, auto-commit is used implicitly.

Explicit transactions may also be used to group multiple operations in a single transaction, and all access methods have optional transaction parameters. For example, the following two operations are performed atomically in a transaction:

```
Transaction txn = env.beginTransaction(null, null);
dao.employerById.put(txn, gizmoInc);
dao.employerById.put(txn, gadgetInc);
txn.commit();
```

To provide maximum performance, the Direct Persistence Layer operations map directly to the Btree operations of the Berkeley DB engine. Unlike other

persistence approaches, keys and indices are exposed for direct access and performance tuning.

Queries are implemented by calling methods of the primary and secondary indices. An `EntityJoin` class is also available for performing equality joins. For example, the following code queries all of Bob's children that work for Gizmo Inc:

```
EntityJoin<String,Person> join = new
EntityJoin(dao.personBySsn);

join.addCondition(dao.personByParentSsn, "111-11-1111");
join.addCondition(dao.personByEmployerIds, gizmoInc.id);

ForwardCursor<Person> results = join.entities();
try {
    for (Person person : results) {
        System.out.println(person.ssn + ' ' + person.name);
    }
} finally {
    results.close();
}
```

Object relationships are based on keys. When a `Person` with a given employer ID in its `employerIds` set is stored, the `Person` object becomes part of the collection of employees for that employer. This collection of employees is accessed using a `SecondaryIndex.subIndex` for the employer ID, as shown below:

```
EntityCursor<Person> employees =
    dao.personByEmployerIds.subIndex(gizmoInc.id).entities();
try {
    for (Person employee : employees) {
        System.out.println(employee.ssn + ' ' + employee.name);
    }
} finally {
    employees.close();
}
```

Note that when Bob's employer is deleted in the example, the `Person` object for Bob is re-fetched to see the change to its `employerIds`. This is because objects are accessed by value, not by reference. In other words, no object cache or "persistence context" is maintained by the Direct Persistence Layer. The low level caching of the embedded Berkeley DB engine, combined with lightweight object bindings, provides maximum performance.

WHICH API TO USE?

The Berkeley DB Java Edition engine has a Base API, a Collections API and a Direct Persistence Layer. Follow these guidelines if you are not sure which API to use:

- When Java classes are used to represent domain objects in an application, meaning that the schema is relatively static, the Direct Persistence Layer is recommended.
- When porting an application between Berkeley DB and Berkeley DB Java Edition, or when implementing your own dynamic schema (the schema of an LDAP server, for example) then the Base API is recommended. You may also prefer to use this API if you have very few domain classes
- The Collections API is useful for interoperating with external components because it conforms to the standard Java Collections Framework. It is therefore useful in combination with both the Base API and the Direct Persistence Layer. You may prefer this API because it provides the familiar Java Collections interface.

PERFORMANCE

Berkeley DB Java Edition is uniquely qualified to perform well within the Java environment. Its underlying architecture is unlike other databases. It is designed to minimize disk access. When it does access the disk, it's done such that there is minimal movement of the disk heads (seeking) allowing for the fastest possible writes.

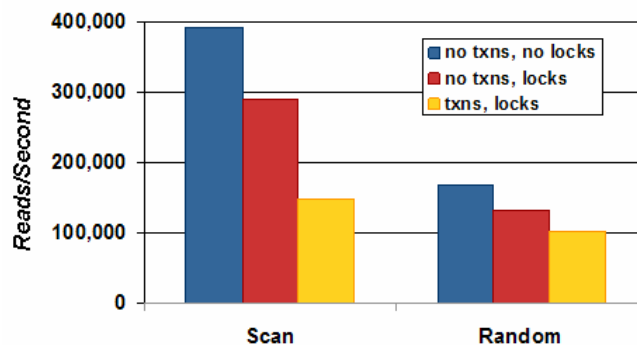
Berkeley DB Java Edition and its Direct Persistence Layer are a more efficient storage combination than comparable EJB and other ORM-style solutions. With Berkeley DB Java Edition only one data cache is required. With EJB there are a minimum of two, one managed by the relational database and one managed by the EJB layer. This introduces inefficiencies and requires more overall memory for the same operations.

When using an ORM solution, even one running a relational database within the same JVM accessed with a JDBC Type 4 driver, there is still the processing overhead of each of the three layers involved. First, the dynamic generation (by the ORM layer) of the SQL required to retrieve the desired object instances. Next the processing overhead when traversing the JDBC layer. Once the SQL arrives in the relational database it must be parsed, a query plan is computed and then executed. Relational data, as rows and columns, is returned via JDBC and then parceled out to the various objects being reconstituted by the ORM solution converting the encoded text into proper Java types along the way. These intermediate steps are avoided in Berkeley DB Java Edition. Berkeley DB Java Edition stores the data directly into its database. The results, object persistence and retrieval, are equivalent whether using EJB/ORM or Berkeley DB Java Edition. The benefit of the Berkeley DB Java Edition Direct Persistence Layer approach is obvious in terms of memory usage and execution time for similar operations.

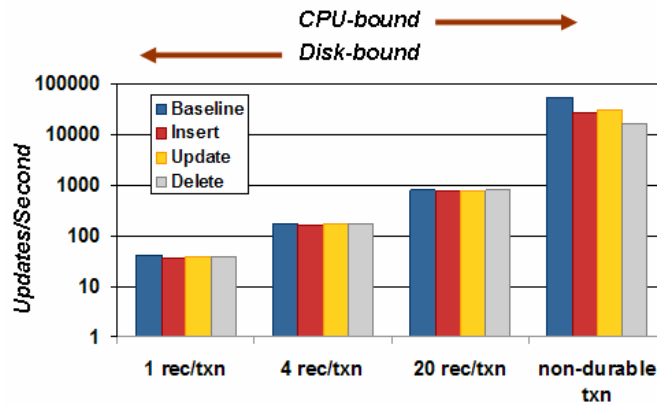
Test Results

The following tests were performed on a 2-way 1.8GHz AMD Opteron with 2GB RAM running Solaris 10 and using the J2SE 1.5.05 JVM. The results demonstrate the speed of read and write operations when using Berkeley DB Java Edition.

The graph below shows read throughput in operations per second when accessing records sequentially and then at random.



The graph below shows write throughput in operations per second on a logarithmic scale. The results show how Berkeley DB Java Edition goes from disk bound on the left to CPU bound on the right as the number of records updated increases.



CONCLUSION

As you can see from the examples, the Berkeley DB Java Edition Direct Persistence Layer is a complete solution for Java object persistence. It is an object mapping and transactional storage solution that can be used in place of ORM-based EJB solutions where ad hoc analysis using SQL is not a requirement.

Additional information about Berkeley DB Java Edition and the full product including source code, documentation, sample code and test code is available for download from:

<http://www.oracle.com/technology/products/berkeley-db/je/index.html>.



Berkeley DB Java Edition Direct Persistence Layer Basics
September 2006

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

Copyright © 2006, Oracle. All rights reserved.

This document is provided for information purposes only and the contents hereof are subject to change without notice.

This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission. Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.