DZone Refcardz

CONTENTS INCLUDE:

- Oracle Berkeley DB Family
- Berkeley DB Java Edition Features
- BDB JE Base API and Collections API
- BDB JE Direct Persistent Layer
- BDB JE Transaction Support and Performance Tuning
- BDB JE Backup and Recovery
- Hot Tips and more...

Getting Started with Oracle Berkeley DB

By Masoud Kalali

ABOUT ORACLE BERKELEY DB

The Oracle Berkeley DB (BDB) family consists of three open source data persistence products which provide developers with fast, reliable, high performance, enterprise ready local databases implemented in the ANSI C and Java programming languages. The BDB family typically stores key-value pairs but is also flexible enough to store complex data models. BDB and BDB Java Edition share the same base API, making it possible to easily switch between the two.

We will review the most important aspects of Oracle BDB family briefly. Then we will dig deep into Oracle BDB Java Edition and see what its exclusive features are. We discuss Based API and Data Persistence Layer API. We will see how we can manage transactions DPL and Base API in addition to persisting complex objects graph using DPL will form the overall development subjects. Backup recovery, tuning, and data migration utilities to migrate data between different editions and installations forms the administrations issues which we will discuss in this Refcard.

THE BDB FAMILY

Oracle BDB Core Edition

Berkeley DB is written in ANSI C and can be used as a library to access the persisted information from within the parent application address space. Oracle BDB provides multiple interfaces for different programming languages including ANSI C, the Java API through JNI in addition to Perl, PHP, and Python.

Oracle BDB XML Edition

Built on top of the BDB, the BDB XML edition allows us to easily store and retrieve indexed XML documents and to use XQuery to access stored XML documents. It also supports accessing data through the same channels that BDB supports.

BDB Java Edition

BDB Java Edition is a pure Java, high performance, and flexible embeddable database for storing data in a key-value format. It supports transactions, direct persistence of Java objects using EJB 3.0-style annotations, and provides a low level key-value retrieval API as well as an “access as collection” API.

KEY FEATURES

Each of the BDB family members supports different feature sets. BDB XML edition enjoys a similar set of base features as the Core BDB. BDB Java edition on the other hand is implemented in a completely different environment with an entirely different set of features and characteristics (See Table 4). The base feature sets are shown in Table 1.

Table 1: Family Feature Sets

Feature Set	Description
Data Store (DS)	Single writer, multiple reader
Concurrent Data Store (CDS)	Multiple writers, multiple snapshot readers
Transactional Data Store (TDS)	Full ACID support on top of CDS
High Availability (HA)	Replication for fault tolerance. Fail over recovery support

Table 2 shows how these features are distributed between the different BDB family members.

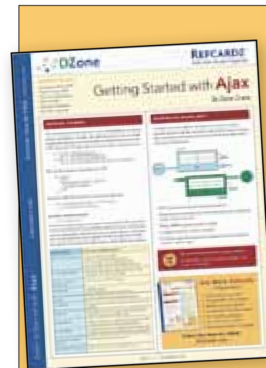
Table 2: Different Editions' Feature Sets	DS	CDS	TS	HA
BDB/BDB XML Edition	✓	✓	✓	✓
BDB Java Edition		✓	✓	

ADDITIONAL FEATURES

The BDB family of products has several special features and offers a range of unique benefits which are listed in Table 3.

Table 3: Family Features and Benefits

Feature	Benefit
Locking	High concurrency
Data stored in application-native format	Performance, no translation required
Programmatic API, no SQL	Performance, flexibility/control
In process, not client-server	Performance, no IPC required
Zero administration	Low cost of ownership
ACID transactions and recovery	Reliability, data integrity
Dual License	Open/Closed source distributions



Get More Refcardz
(They're free!)

- Authoritative content
- Designed for developers
- Written by top experts
- Latest tools & technologies
- Hot tips & examples
- Bonus content online
- New issue every 1-2 weeks

Subscribe Now for FREE!
Refcardz.com

In memory or on disk operation	Transacted caching/ persisted data store
Similar data access API	Easy switch between JE and BDB
Just a set of library	Easy to deploy and use
Very large databases	Virtually no limit on database size

Features unique to BDB Java Edition are listed in Table 4.

Table 4: BDB Java Edition Exclusive Features	
Feature	**Benefit**
Fast, indexed, BTree	Ultra fast data retrieval
Java EE JTA and JCA support	Integration with Java EE application servers
Efficient Direct Persistence Layer	EJB 3.0 like annotation to store Java Objects graph
Easy Java Collections API	Transactional manipulation of Base API through enhanced Java Collections
Low Level Base API	Work with dynamic data schema
JMX Support	Monitor able from within parent application

These features, along with a common set of features, make the Java edition a potential candidate for use cases that require caching, application data repositories, POJO persistence, queuing/buffering, Web services, SOA, and Integration.

INTRODUCING BERKELEY DB JAVA EDITION

Installation

You can download BDB JE from http//bit.ly/APfJ5. After extracting the archive you'll see several directories with self-describing names. The only file which is required to be in the class path to compile and run the included code snippet is je-3.3.75.jar (the exact file name may vary) which is placed inside the lib directory. Notice that BDB JE requires J2SE JDK version 1.5.0_10 or later.

Hot Tip All editions of Berkeley DB are freely available for download and can be used in open source products which are not distributed to third parties. A commercial license is necessary for using any of the BDB editions in a closed source and packaged product. For more information about licensing visit: http://bit.ly/17pMwZ

Access APIs

BDB JE provides three APIs for accessing persisted data. The Base API provides a simple key-value model for storing and retrieving data. The Direct Persistence Layer (DPL) API lets you persist any Java class with a default constructor into the database and retrieve it using a rich set of data retrieval APIs. And finally the Collections API which extends the well known Java Collections API with data persistence and transaction support over data access.

Base API sample

The Base API is the simplest way to access data. It stores a key and a value which can be any serializable Java object.

```
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
Environment dbEnv = new Environment(new File("/home/masoud/dben"),
envConfig);
DatabaseConfig dbconf = new DatabaseConfig();
dbconf.setAllowCreate(true);
dbconf.setSortedDuplicates(false);//allow update
Database db = dbEnv.openDatabase(null, "SampleDB ", dbconf);
DatabaseEntry searchEntry = new DatabaseEntry();
DatabaseEntry dataValue = new DatabaseEntry(" data content".
getBytes("UTF-8"));
DatabaseEntry keyValue = new DatabaseEntry("key content".
getBytes("UTF-8"));
db.put(null, keyValue, dataValue);//inserting an entry
db.get(null, keyValue, searchEntry, LockMode.DEFAULT);//retrieving
record
String foundData = new String(searchEntry.getData(), "UTF-8");
dataValue = new DatabaseEntry("updated data content".
getBytes("UTF-8"));
db.put(null, keyValue, dataValue);//updating an entry
db.delete(null, keyValue);//delete operation
db.close();
dbEnv.close();
```

There are multiple overrides for the `Database.put` method to prevent duplicate records from being inserted and to prevent record overwrites.

DPL Sample

DPL sample consists of two parts, the entity class and the entity management class which handle CRUD over the entity class.

Entity Class

```
@Entity
public class Employee {
 @PrimaryKey
 public String empID;
 public String lastname;
 @SecondaryKey(relate = Relationship.MANY_TO_MANY,
 relatedEntity = Project.class,onRelatedEntityDelete =
DeleteAction.NULLIFY)
 public Set<Long> projects;
 public Employee() {    }
 public Employee(String empID, String lastname, Set<Long> projects)
{
    this.empID = empID;
    this.lastname = lastname;
    this.projects = projects;
     }
  }}
```

This is a simple POJO with few annotations to mark it as an entity with a `String` primary key. For now ignore the `@Secondarykey` annotation, we will discuss it later.

The data management Class

```
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
Environment dbEnv = new Environment(new File("/home/masoud/dben-
dpl"), envConfig);
StoreConfig stConf = new StoreConfig();
stConf.setAllowCreate(true);

EntityStore store = new EntityStore(dbEnv, "DPLSample", stConf);

PrimaryIndex<String, Employee> userIndex;
userIndex = store.getPrimaryIndex(String.class, Employee.class);
userIndex.putNoReturn(new Employee("u180", "Doe", null));//insert
Employee user = userIndex.get("u180");//retrieve
userIndex.putNoReturn(new Employee("u180", "Locke", null));//
Update
userIndex.delete("u180");//delete

store.close();
dbEnv.close();
```

These two code snippets show the simplest from of performing CRUD operation without using transaction or complex object relationships.

Sample code description

An `Environment` provides a unit of encapsulation for one or more databases. Environments correspond to a directory on disk. The `Environment` is also used to manage and configure resources such as transactions. `EnvironmentConfig` is used to configure the `Environment`, with options such as transaction configuration, locking, caching, getting different types of statistics including database, locks and transaction statistics, etc.

One level closer to our application is `DatabaseConfig` and Database object when we use Base API. When we use DPL these objects are replaced by `StoreConfig` and `EntityStore`.

In Base API `DatabaseConfig` and Database objects provide access to the database and how the database can be accessed.

Configurations like read-only access, record duplication handling, creating in-memory databases, transaction support, etc. are provided through `DatabaseConfig`.

In DPL `StoreConfig` and `EntityStore` objects provide access to object storage and how the object storage can be accessed. Configurations such as read only access, data model mutation, creating in-memory databases, transaction support, etc. are provided through `StoreConfig`.

The `PrimaryIndex` class provides the primary storage and access methods for the instances of a particular entity class. There are multiple overrides for the `PrimaryIndex.put` method to prevent duplicate entity insertion and provide entity overwrite prevention.

Hot Tip When closing an Environment or Database or when we commit a Transaction in a multi thread application we should ensure that no thread still has in-progress tasks.

BDB JAVA EDITION ENVIRONMENT ANATOMY

A BDB JE database consists of one or more log files which are placed inside the environment directory.

The log files are named NNNNNNNN.jdb where NNNNNNNN is an 8-digit hexadecimal number that increases by 1 (starting from 00000000) for each log file written to disk. BDB JE rolls to next file when the current file size reaches the predefined configurable size. The predefined size is 10MB.

A BDB database can be considered like a relational table in an RDBMS. In Base API we directly use the database we need to access, while in DPL we use an `EntityStore` which may interact with multiple databases under the hood.

Each BDB environment can contain tens of databases and all of these databases will be stored in a single row of log files. (No separate log files per-database). Figure 1 shows the concept visually.

Hot Tip To create in-memory database we can use DatabaseConfig.setTemporary(true) and StoreConfig.setTemporary(true) to get an in-memory instance with no data persisted beyond the current session.

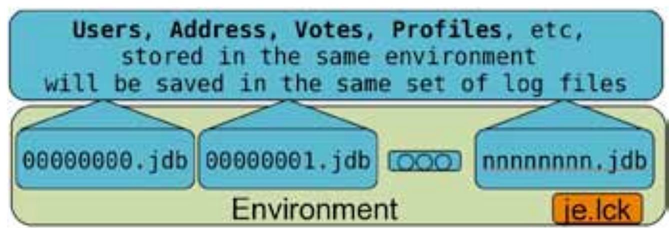


Figure 1: BDB JE environment and log files.

Hot Tip The environment path should point to an already existing directory, otherwise the application will face and exception. When we create an environmnt object for the first time, necessary files are created inside that direcory.

BDB JAVA EDITION ENVIRONMENT ANATOMY

Table 5 shows Base API characteristics and benefits. The key-value access model provides the most flexibility.

Table 5: Base API Features
Key value store retrieval, value can be anything
Cursor API to traverse in a dataset forward and backward
JCA (Java Connectivity Architecture) support
JMX (Java Management eXtension) support

Table 6 shows most important DPL API characteristics and capabilities. Annotation and Object Mapping make rapid application development possible.

Table 6: DPL API Features
Type Safe access to persisted objects
Updating classes with adding new fields is supported
Persistent class fields can be private, package-private, protected or public
Automatic and extendable data binding between Objects and underlying storage
Index fields can be accessed using a standard java.util collection.
Java annotations are used to define metadata like relations between objects
Field refactoring is supported without changing the stored date.(Called mutation)

Table 6 and Table 7 list the features that mostly determine when we should use which API. Table 7 lists possible use cases for each API.

Table 7: Which API is suitable for your case	
Use case characteristics	**Suitable API**
Data model is highly dynamic and changing	Base API
Data model has complex object model and relationships	DPL
Need application portability between Java Edition and Core Edition	DPL, Base API

TRANSACTION SUPPORT

Transaction Support is an inseparable part of enterprise software development. BDB JE supports transaction and provides concurrency and record level locking. To add transaction support to DPL in our DPL sample code we can introduce the following changes:

```
...
envConfig.setTransactional(true);
stConf.setTransactional(true);
TransactionConfig txConf = new TransactionConfig();
stConf.setTransactional(true);
txConf.setReadCommitted(true);
Transaction tx= dbEnv.getThreadTransaction();
dbEnv.beginTransaction(tx, txConf);
...
userIndex.putNoReturn(tx, new Employee(“u180”, “Doe”, null));//
insert
...
tx.commit();
...
```

The simplicity of BDB JE Transaction Support makes it very suitable for transactional cache systems. The isolation level, deferrable and manual synchronization of transactional data with hard disk (Durability), replication policy, and transaction lock request and transaction lifetime timeout can be configured using the `Transaction` and `TransactionConfig` objects.

Hot Tip Environment, Database, and EntityStore are thread safe meaning that we can use them in multiple threads without manual synchronization

Transaction support in Base API is a bit different, as in Base API we directly deal with databases while in DPL we deal with environment and `EntityStore` objects. The following changes will allow transaction support in Base API.

```
...
envConfig.setTransactional(true);
dbconf.setTransactional(true);
TransactionConfig txConf = new TransactionConfig();
txConf.setSerializableIsolation(true);
txConf.setNoSync(true);
Transaction tx = dbEnv.getThreadTransaction();
tx.setLockTimeout(1000);
tx.setTxnTimeout(5000);
Database db = dbEnv.openDatabase(tx, «SmpleDB», dbconf);
dbEnv.beginTransaction(tx, txConf);
...
db.put(tx, keyValue, dataValue);//inserting an entry
...
tx.commit();
```

More transaction related supported features are demonstrated in this snippet. `Environment`, `Database`, `EntityStore`, etc. configuration is omitted from these snippet for sake of simplicity.

Hot Tip Once a transaction is committed, the transaction handle is no longer valid and a new transaction object is required for further transactional activities.

PERSISTING COMPLEX OBJECT GRAPH USING DPL

For this section we leave Base API alone and focus on using DPL for complex object graphs. We continue with introducing secondary index and many-to-many mapping.

Let's look at some important annotations that we have for defining the object model.

Table 8: BDB JE annotations

Annotation	Description
@Entity	Declares an entity class; that is, a class with a primary index and optionally one or more indices.
@PrimaryKey	Defines the class primary key and must be used one and only one time for every entity class.
@SecondaryKey	Declares a specific data member in an entity class to be a secondary key for that object. This annotation is optional, and can be used multiple times for an entity class.
@Persistent	Declares a persistent class which lives in relation to an entity class.
@NotTransient	Defines a field as being persistent even when it is declared with the transient keyword.
@NotPersistent	Defines a field as being non-persistent even when it is not declared with the transient keyword.
@KeyField	Indicates the sorting position of a key field in a composite key class when the Comparable interface is not implemented. The KeyField integer element specifies the sort order of this field within the set of fields in the composite key.

We used two of these annotations in practice and you saw `@SecondaryKey` in the `Employee` class. Now we are going to see how the `@SecondaryKey` annotation can be used. Let's create the Project entity which the `Employee` class has a many-to-many relation with.

```
@Entity
public class Project {

    public String projName;
    @PrimaryKey(sequence = «ID»)
    public long projID;

    public Project() {
    }
    public Project(String projName) {
        this.projName = projName;
    }
}
```

The `@PrimaryKey` annotation has a string element to define the name of a sequence from which we can assign primary key values automatically. The primary key field type must be numerical and a named sequence can be used for multiple entities.

Now let's see how we can store and retrieve an employee with its related project objects.

```
...
PrimaryIndex<String, Employee> empByID;
PrimaryIndex<Long, Project> projByID;

empByID = store.getPrimaryIndex(String.class, Employee.class);
projByID = store.getPrimaryIndex(Long.class, Project.class);

SecondaryIndex<Long, String, Employee> empsByProject;
empsByProject = store.getSecondaryIndex(empByID, Long.class,
"projects");

Set<Long> projects = new HashSet<Long>();
Project proj = null;

proj = new Project("Develop FX");
projByID.putNoReturn(proj);
projects.add(proj.projID);
proj = new Project("Develop WS");
projByID.putNoReturn(proj);
projects.add(proj.projID);

empByID.putNoReturn(new Employee("u146", "Shephard", projects));//
insert
empByID.putNoReturn(new Employee("u144", "Locke", projects));//
insert

EntityIndex<String, Employee> projs = empsByProject.subIndex(proj.
projID);
EntityCursor<Employee> pcur = projs.entities();
for (Employee entity : pcur) {
    //process the employees
}

EntityCursor<Employee> emplRange = empByID.entities("e146", true,
"u148", true);
for (Employee entity : emplRange) {
    //process the employees
}

emplRange.close();
pcur.close();
store.close();
dbEnv.close();
```

The `Environment` and `EntityStore` definitions are omitted. The `SecondaryIndex` provides primary methods for retrieving objects related to the Secondary Key of a particular object. The `SecondaryIndex` can be used to retrieve the related objects through a traversable cursor. We can also use `SecondaryIndex` to query for a specific range of objects in a given range for its primary key.

Table 9: Supported Object Relation

Relation	Description
ONE_TO_ONE	A single entity is related to a single secondary key value.
ONE_TO_MANY	A single entity is related to one or more secondary key values.
MANY_TO_ONE	One or more entities are related to a single secondary key value.
MANY_TO_MANY	One or more entities are related to one or more secondary key values.

A `SecondaryIndex` can be used to traverse over the collection of secondary key's values to retrieve the secondary objects.

Hot Tip Multiple processes can open a database as long as only one process opens it in read-write mode and other processes open the database in read-only mode. The read-only processes get an open-time snapshot of the database and won't see any changes coming from other process.

BDB JE COLLECTIONS API

The only different between using BDB JE collections API and classic collections is the fact that when we use BDB JE Collections API we are accessing persisted objects instead of in-memory

objects which we usually access in classic collection APIs.

The Collections API Characteristics
An implementation Map, SortedMap, Set, SortedSet, and Iterator.
To stay compatible with Java Collections, Transaction is supported using TransactionWorker and TransactionRunner which the former one is the interface which we can implement to execute our code in a transaction and later one process the transaction.
Keys and values are represented as Java objects. Custom binding can be defined to bind the stored bytes to any type or format like XML, for example.
Data binding should be defined to instruct the Collections API about how keys and values are represented as stored data and how stored data is converted to and from Java objects. We can use one of the two (SerialBinding, TupleBinding) default data bindings or a custom data binding.
Environment, EnvironmentConfig, Database and DatabaseConfig stay the same as it was for Base API.
Collections API extends Java serialization to store class description separately to make data records much more compact.

To get a real sense about BDB JE Collections API think of it as we can persist and retrieve objects using a collection class like `SortedMap's` methods like `tailMap`, `subMap` or `put`, `putall`, `get`, and so on.

But before we use the `SortedMap` object to access the stored data, we need to initialize the base objects like Database and Environment; we should create the `ClassCatalog` object, and finally we should define bindings for our key and value types.

Collections API sample

Now let's see how we can store and retrieve our our objects using Collections API. In this sample we are persisting a pair of Integer key and String value using `SortedMap`.

First lets analyze the `TransactionWorker` implementation.

```
public class TransWorker implements TransactionWorker {
    private ClassCatalog catalog;
    private Database db;
    private SortedMap map;
    public TransWorker(Environment env) throws Exception {
        DatabaseConfig dbConfig = new DatabaseConfig();
        dbConfig.setTransactional(true);
        dbConfig.setAllowCreate(true);
  Database catalogDb = env.openDatabase(null, "catalog", dbConfig);
        catalog = new StoredClassCatalog(catalogDb);
        // use Integer tuple binding for key entries
        TupleBinding keyBinding =
                TupleBinding.getPrimitiveBinding(Integer.class);
        // use String serial binding for data entries
        SerialBinding dataBinding = new SerialBinding(catalog,
String.class);
        db = env.openDatabase(null, "dben-col", dbConfig);
        map = new StoredSortedMap(db, keyBinding, dataBinding,
true);
    }
    /** Performs work within a transaction. */
    public void doWork() throws Exception {
        // check for existing data and writing
        Integer key = new Integer(0);
        String val = (String) map.get(key);
        if (val == null) {
            map.put(new Integer(10), "Second");
        }
        //Reading Data
       Iterator iter = map.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry entry = (Map.Entry) iter.next();
          //Process the entry
        }
    }
}
```

`TransWorker` implements `TransactionWorker` which makes it necessary to implement the `doWork` method. This method is called by `TransactionRunner` when we pass an object of `TransWorker` to its run method. The `TransWorker` constructor simply receive an Environment object and construct other required objects . Then it opens the database in Collection mode, creates the required binding for the key and values we want to store in the database and finally it creates the `SortedMap` object which we can use to put and retrieve objects using it.

Now let's see the driver code which put this class in action.

```
public class CollectionSample  {
    public static void main(String[] argv)
        throws Exception {
        // Creating the environment
        EnvironmentConfig envConfig = new EnvironmentConfig();
        envConfig.setTransactional(true);
        envConfig.setAllowCreate(true);
        Environment dbEnv = new Environment(new File("/home/
masoud/dben-col"), envConfig);
        // creating an instance of our TransactionWorker
        TransWorker worker = new TransWorker(dbEnv);
        TransactionRunner runner = new TransactionRunner(dbEnv);
            runner.run(worker);
    }
}
```

The steps demonstrated in the `CollectionSample` are self describing. The only new object in this snippet is the `TransactionRunner` object which we used to run the `TransWorker` object. I omit many of the safe programming portions to keep the code simple and conscious. we need exception handling and properly closure of all BDB JE objects to ensure data integrity

BDB JE BACKUP/RECOVERY AND TUNING

Backup and Recovery

We can simply backup the BDB databases by creating an operating system level copy of all jdb files. When required we can put the archived files back into the environment directory to get a database back to the state it was at. The best option is to make sure all transactions and the write process are finished to have a consistent backup of the database.

The BDB JE provides a helper class located at `com.sleepycat.je.util.DbBackup` to perform the backup process from within a Java application. This utility class can create an incremental backup of a database and later on can restore from that backup. The helper class ideally freezes the BDB JE activities during the backup to ensure that the created backup exactly represents the database state when the backup process started.

Tuning

Berkeley DB JE has 3 daemon threads and configuring these threads affects the overall application performance and behavior. These 3 threads are as follow:

Cleaner Thread	Responsible for cleaning and deleting unused log files. This thread is run only if the environment is opened for write access.
Checkpointer Thread	Basically keeps the BTree shape consistent. Checkpointer thread is triggered when environment opens, environment closes, and database log file grows by some certain amount.
Compressor Thread	For cleaning the BTree structure from unused nodes.

These threads can be configured through a properties file named `je.properties` or by using the `EnvironmentConfig` and `EnvironmentMutableConfig` objects. The `je.properties` file, which is a simple key-value file, should be placed inside the environment directory and override any further configuration which we may make using the `EnvironmentConfig` and `EnvironmentMutableConfig` in the Java code.

The other performance effective factor is cache size. For on-disk instances cache size determines how often the application needs to refer to permanent storage in order to retrieve some data bucket. When we use in-memory instances cache size determines whether our database information will be paged into swap space or it will stay in the main memory.

je.cleaner. minUtilization	Ensures that a minimum amount of space is occupied by live records by removing obsolete records. Default occupied percentage is 50%.
je.cleaner. expunge	Determines the cleaner behavior in the event that it is able to remove an entire log file. If "true" the log file will be deleted, otherwise it will be renamed to nnnnnnnn.del
je.checkpointer. bytesInterval	Determines how often the Checkpointer should check the BTree structure. If it performs the checks little by little it will ensure a faster application startup but will consume more resources specially IO.
je.maxMemory Percent	Determines what percentage of JVM maximum memory size can be used for BDB JE cache. To determine the ideal cache size we should put the application in the production environment and monitor its behavior.

A complete list of all configurable properties, with explanations, is available in `EnvironmentConfig` Javadoc. The list is comprehensive and allows us to configure the BDB JE at granular level.

All of these parameters can be set from Java code using the `EnvironmentConfig` object. The properties file overrides the values set by using `EnvironmentConfig` object.

Helper Utilities

Three command line utilities are provided to facilitate dumping the databases from one environment, verifying the database structure, and loading the dump into another environment.

DbDump	Dumps a database to a user-readable format.
DbLoad	Loads a database from the DbDump output.
DbVerify	Verifies the structure of a database.

To run each of these utilities, switch to BDB JE directory, switch to lib directory and execute as shown in the following command:

```
java -cp je-3.3.75.jar com.sleepycat.je.util.DbVerify
```

The JAR file name may differ depending on your version of BDB JE. These commands can also be used to port a BDB JE database to BDB Core Edition.

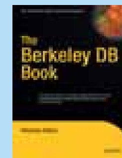
Hot Tip A very good set of tutorials for different set of BDB JE APIs are available inside the docs folder of BDB JE package. Several examples for different set of functionalities are provided inside the examples directory of the BDB JE package.

ABOUT THE AUTHOR

Masoud Kalali holds a software engineering degree and has been working on software development projects since 1998. He has experience with a variety of technologies (.Net, J2EE, CORBA, and COM+) on diverse platforms (Solaris, Linux, and Windows). His experience is in software architecture, design and server side development. Masoud has several articles in Java.net. He is one of founder members of NetBeans Dream Team. Masoud's main area of research and interest includes Web Services and Service Oriented Architecture along with large scale and high throughput systems' development and deployment.

Blog: http://weblogs.java.net/blog/kalali/
Contact: Kalali@gmail.com

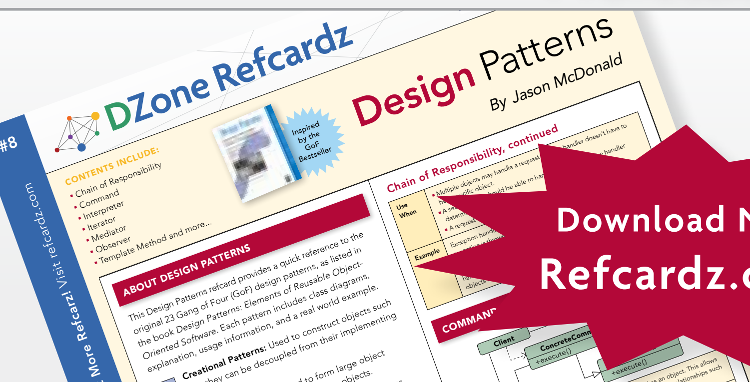
RECOMMENDED BOOK



The Berkeley DB Book is a practical guide to the intricacies of the Berkeley DB. This book covers in-depth the complex design issues that are mostly only touched on in terse footnotes within the dense Berkeley DB reference manual. It explains the technology at a higher level and also covers the internals, providing generous code and design examples.

BUY NOW
books.dzone.com/books/berkeley-db

Professional Cheat Sheets You Can Trust



Download Now
Refcardz.com

"Exactly what busy developers need: simple, short, and to the point."

James Ward, Adobe Systems

Upcoming Titles
Expression Web
Eclipse Plug-In Development
Adobe Live Cycle
Adobe Flash Builder 4
Java Performance Tuning
WPF
F#

Most Popular
Spring Configuration
jQuery Selectors
Windows Powershell
Dependency Injection with EJB 3
Netbeans IDE JavaEditor
Getting Started with Eclipse
Very First Steps in Flex

DZone communities deliver over 6 million pages each month to more than 3.3 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more. **"DZone is a developer's dream,"** says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513
888.678.0399
919.678.0300
Refcardz Feedback Welcome
refcardz@dzone.com
Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-62-2
ISBN-10: 1-934238-62-7
50795
9 781934 238622
$7.95

Copyright © 2009 DZone, Inc. All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher. Reference:

Version 1.0