CMPUT 291 - File and Database Management (Fall 2019)

Berkeley DB Tutorial (C & Java)

Introduction to Berkeley DB (in Java & C)

Introduction to Berkeley DB (in Java & C)

In this tutorial we are going to learn:

- How to set your Environment for BDB
- The Basics of the Berkeley DB API
- How to code simple programs using BDB API

Environment settings:

To enable the BDB APIs, you must set your CLASSPATH. Enter the Unix command "`echo $CLASSPATH` " to determine if you already have '/usr/share/java/db.jar'. If this path is not in your CLASSPATH, or nothing is output, then you have to set your CLASSPATH as below.

```
export CLASSPATH=$CLASSPATH:.:/usr/share/java/db.jar
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/oracle/lib
```

This is the bash syntax but you can do a similar setting in csh using *setenv* command. You may want to add the two lines to your .bashrc (or the respective lines to .cshrc) if you don't want to do the setting every time you log into the lab machines.

Intorduction to BDB API:

Berkeley DB is an open source library that provides a high-performance embedded database for key/value data. It stores arbitrary key/data pairs as byte arrays, and supports multiple data items for a single key. A program accessing the database is free to decide how the data is to be stored in a record. Berkeley DB puts no constraints on the record's data. The record and its key can both be up to four gigabytes long. Unlike previously seen Oracle DB, BDB is not a relational DBMS. Usually, Berkeley DB is used when (1) your applications doesn't need the full functionality of a DBMS but requires a high performance, (2) you are willing to design the file structure and build the indexes yourself.
Here are the four file structures supported by Berkeley DB

- Hash - a good way to store data that needs a quick random-access
- BTree - keeps keys that are close together near one another in storage, good for range-based search and partial matches
- Queue - supports fast access to fixed-length records accessed sequentially or by logical record number. The logical number is the primary key of the record.
- Recno - supports fixed- or variable-length records, accessed sequentially or by logical record number

How to code using BDB API:

To have an access to a Berkeley DB API in java you simply have to import the package com.sleepycat.db.* by using the following command:

```
import com.sleepycat.db.*
```

In C, you would need to include the header file *db.h* in your code.

Creating and opening a database:

Berkeley DB databases are represented either as physical files on the disk or in-memory. In this tutorial we're going to take a look only at those that are stored on the disk. To create a new Berkeley DB database file you need to follow two simple steps: First you need to specify the configurations, such as storage type, options that allow you to create a database file if it doesn't exist, allowing or disallowing to have duplicate keys in a database, etc. To do that you need to create an object of type *DatabaseConfiguration* and specify needed attributes:

```
DatabaseConfig dbConfig = new DatabaseConfig();
dbConfig.setSortedDuplicates(true); // to allow duplicate keys
dbConfig.setType(DatabaseType.BTREE); // sets storage type to BTree
dbConfig.setAllowCreate(true); // create a database if it doesn't exist
```

An equivalent C code for the same action will be

```
DB *db; // database object
int ret;
// create the database object
if ((ret = db_create(&db, NULL, 0)) != 0)
  {
    fprintf(stderr, "db_create: %s\n", db_strerror(ret));
    return 1;
  }

  db->set_flags(db, DB_DUP); //support sorted, duplicate data items

  // open the database
  ret = db->open(db, NULL, DATABASE, NULL, DB_BTREE, 0, 0664);
  if (ret != 0)
  {
    // create the database if it doesn't exist
    ret = db->open(db, NULL, DATABASE, NULL, DB_BTREE, DB_CREATE, 0664);
  }
```

Second you would create a database handle using *Database* class (second parameter allows you to have multiple databases in a single file, here it's set to null, so we have only one Berkeley DB database in a file):

```
Database std_db = new Database(fileName, databaseName, databaseConfig);
```

where *fileName* is the name of an underlying file that will be used to back the database, *databaseName* is an optional parameter that allows applications to have multiple databases in a single file. If you want to have only one database per file, which is usually the case, set this parameter to null. *databaseConfig* gives the database open attributes, which would be null if default attributes are used. Remember to wrap the code in a try-catch block in order to avoid *unreported exception* error.
The syntax in C is

```
int db_create(DB **dbp, DB_ENV *dbenv, u_int32_t flags);
```

where dbp is a pointer to the pointer to the database object, dbenv is NULL if the database is standalone (i.e. it is not part of any Berkeley DB environment); if the dbenv parameter is not NULL, the database is created within the specified Berkeley DB environment. flags is set to 0 or DB_XA_CREATE.
To open an existing database you just have to provide the file of the existing database using *fileName* parameter.

Inserting a record into the database

As noted above, Berkeley DB stores records in a database in the form of a key/data pair. Both key and data items are represented by *DatabaseEntry* objects, encoded as a byte array. Key and data byte arrays may refer to arrays of zero to essentially unlimited length. To set a value for a *DatabaseEntry* object methods *setData* and *setSize* are used, which receive a value to be stored in the form of byte array and the size of the value in bytes respectively. For example, suppose we need to store student's id as a key and the name of the student as a data; the code in Java would look like as follows:

```
DatabaseEntry key, data;
key = new DatabaseEntry();
data = new DatabaseEntry();
String student_id = "102345";
String name = "John Doe";
key.setData(student_id.getBytes());
key.setSize(student_id.length());
data.setData(name.getBytes());
data.setSize(name.length());
```

And in C, it would look like

```
DBT key, data;
char id[6], name[40];
strcpy(id,"102345");
strcpy(name,"John Doe");
memset(&key, 0, sizeof(key));
memset(&data, 0, sizeof(data));
key.data = id;
key.size = (strlen(id)+1);
data.data = name;
data.size = (strlen(name)+1);
```

To store the prepared data in the database you have to simply call the method *put* of the *Database* object which returns the *OperationStatus* object (which helps to determine whether the operation was successful):

```
OperationStatus oprst = std_db.put(txn, key, data);
```

where *txn* is a transactional database parameter usually set to *null* in our applications. The syntax in C is

```
int DB->put(DB *db, DB_TXN *txnid, DBT *key, DBT *data, u_int32_t flags);
```

Remeber to close the database afterwards, in order to commit the changes. If the key already appears in the database and duplicates are not configured, the existing key/data pair will be replaced. If the key already appears in the database and sorted duplicates are configured, the new data value is inserted at the correct sorted location. If the key already appears in the database and unsorted duplicates are configured, the new data value is appended at the end of the duplicate set.

Retrieving a record from database

in Java, method *get* is called to retrieve a record from the database.

```
OperationStatus get(txn, key, data, lockMode)
```

where *key* is initialized to a non-null byte array by the caller and *data* is a byte array returned. The C syntax is

```
int DB->get(DB *db, DB_TXN *txnid, DBT *key, DBT *data, u_int32_t flags);
```

For example, let's try to retrieve a key/data pair that was stored earlier:

```
DatabaseEntry key, data;
key = new DatabaseEntry();
data = new DatabaseEntry();
String student_id = "102345";
key.setData(student_id.getBytes());
key.setSize(student_id.length());
OperationStatus oprStatus = std_db.get(null, key, data, LockMode.DEFAULT);
String name = new String(data.getData());
```

In C, the code would look like:

```
DBT key, data;
memset(&key, 0, sizeof(key));
memset(&data, 0, sizeof(data));
char id[6];
strcpy(id,"102345");
key.data = id;
key.size = (1+strlen(id));
ret = db->get(db, NULL, &key, &data, 0);
printf("data: %s\n", (char*)data.data);
```

Deleting a record from the database

The syntax in Java is

```
OperationStatus delete(txn, key);
```

and in C is

```
int DB->del(DB *db, DB_TXN *txnid, DBT *key, u_int32_t flags);
```

For example let's try to delete the key/data pair that was inserted into the database:

```
DatabaseEntry key;
key = new DatabaseEntry();
String student_id = "102345";
key.setData(student_id.getBytes());
key.setSize(student_id.length());
OperationStatus oprStatus = std_db.delete(null, key);
```

In C, the code would look like

```
DBT key;
memset(&key, 0, sizeof(key));
char *id = "102345"
key.data = id;
key.size = (1+strlen(id))*sizeof(char);
ret = db->del(db, NULL, &key, 0);
```

Don't forget to close the database afterwards. Note that if there's more than one record that has the specified key value, than all the records are going to be removed from the database.

Closing a database

In Java,

```
std_db.close();
```

and in C

```
ret = db->close(db, 0);
```

Cursors

Cursors are used for operating on collections of records, for iterating over a database, and for saving handles to individual records, so that they can be modified after they have been read. The other thing you should keep in mind is that you can iterate through duplicate records by using a cursor. The method *openCursor* is used to create a cursor for the handle to a current database:

```
Cursor openCursor(txn, cursorConfig)
```

In C,

```
Int DB->cursor(DB *db, DB_TXN *txnid, DBC **cursor, u_int32_t flags);
```

Cursors can be used to search for database records. In java, *getSearchKey* method moves the cursor to the first record in the database w.r.t. the specified key:

```
OperationStatus getSearchKey(key, data, lockMode)
```

The equivalent C code looks like:

```
int DBcursor->c_get(DBC *cursor, DBT *key, DBT *data, u_int32_t flags);
```

Note that the "flags" parameter should be set to "DB_SET" in order for the cursor to move to the specified key. If the next record of the database is a duplicate data record for the current key/data pair (which the cursor is pointing to), you can move the cursor to that duplicate record via method *getNextDup* in Java:

```
public OperationStatus getNextDup( key, data, lockMode)
```

The equivalent methid is C is the same *c_get* method but with the "flags" parameter set to "DB_NEXT_DUP".
As an example, let's try to show the records of the database with the same key (assuming there exists duplicate key/data pairs with the key of "102345") in Java:

```
Cursor cursor = db.openCursor(null, null);
String searchkey="102345";
DatabaseEntry key = new DatabaseEntry(), data = new DatabaseEntry();
key.setData(searchkey.getBytes()); key.setSize(searchkey.length());
OperationStatus oprStatus;
oprStatus = cursor.getSearchKey(key, data, LockMode.DEFAULT);
// Returns OperationStatus
while (oprStatus == OperationStatus.SUCCESS)
{
        System.out.println (new String(data.getData()));
        oprStatus = cursor.getNextDup(key, data, LockMode.DEFAULT);  // get next duplicate
}
```

and in C:

```
DBT key, data;
DBC *dbcp;
/* Get a cursor */
db->cursor(db, NULL, &dbcp, 0);
memset(&key,0,sizeof(DBT));
memset(&data,0,sizeof(DBT));
char id[6];
strcpy(id, "102345");
key.data=id;
key.size = 1+strlen(id);
// go to first record, such that key = id
ret = dbcp->c_get(dbcp, &key, &data, DB_SET);
while (ret != DB_NOTFOUND)
{
        printf("key: %s, data: %s\n", (char *) key.data, (char *) data.data);
        ret = dbcp->c_get(dbcp, &key, &data, DB_NEXT_DUP);
}
```

Sample programs

The first example application is bdb_populate, named bdb_populate.java for Java code or bdb_populate.c for C code. In this example, we create a BDB database file named "students.db" for Java and "cstudents.db" for C. The key is student name and data is student grade. The user is asked to enter information about students, and if there already exist students with the specified name, then the list of all students with the name and their marks will be printed out and the user is asked if a student should be added to the list. The Java code can be compiled and run as

```
javac bdb_populate.java
java bdb_populate
```

and the C code can be compiled and run as

```
gcc bdb_populate.c -o bdb_populate -ldb
./bdb_populate
```

The second example application is bdb_traverse, named bdb_ traverse.java for Java and bdb_ traverse.c for C. In this example we output all the records of the database file "students.db" for Java and "cstudents.db" for C. Again the Java code can be compiled and run as

```
javac bdb_traverse.java
java bdb_traverse
```

and the C code can be compiled and run as

```
gcc  bdb_traverse.c -o bdb_traverse -ldb
./bdb_ traverse
```

Here is a sample program for range search in Java and in C, which can be compiled and run as the previous programs.

More information

See the Berkeley DB documentation.

Last modified: Tuesday, 10 September 2019, 6:20 PM

◄ Introduction to Python bsddb3 | Jump to... | Mini-Project 1 Submission ►