UNIVERSITÉ LIBRE DE BRUXELLES

Embedded databases & Berkeley

Shubham Batra, Nathan Liccardo

December 18, 2017

Contents

1 Introduction **4**
1.1 An introduction to data management . . . 4
1.2 Data access and data management . . . 4
1.3 Network Database . . . 5
1.4 Clients and servers . . . 5

2 Embedded databases **6**
2.1 Introduction . . . 6
2.2 Definition . . . 6
2.3 Properties of embedded databases . . . 6
2.3.1 Minimisation of footprint . . . 6
2.3.2 High availability . . . 6

3 Berkeley Database **7**
3.1 Introduction . . . 7
3.2 What is Berkeley Database ? . . . 7
3.2.1 Berkeley DB Library . . . 7
3.2.2 Berkeley DB Architecture . . . 7
3.3 Records & Requests . . . 8
3.4 Berkeley Database . . . 9
3.4.1 Creating Database . . . 9
3.4.2 Creating Table . . . 9
3.4.3 Linking Table to Data . . . 10
3.4.4 Inserting Data to Table . . . 11
3.4.5 Updating Data to Table . . . 11
3.4.6 Retrieving Data from the Table . . . 12
3.4.7 Example . . . 13
3.5 Data Access Services . . . 14
3.6 Data management services . . . 14
3.7 The Berkeley DB products . . . 14
3.8 Available access methods . . . 15
3.9 Berkeley DB subsystems . . . 15
3.10 Programming model . . . 16
3.11 Programmatic APIs . . . 16
3.12 Security . . . 17
3.13 Encryption . . . 17
3.14 Berkeley Database Customers . . . 18
3.14.1 Amazon.com . . . 18
3.14.2 Motorola Wireless network . . . 18
3.14.3 Cisco Systems Broadband Provisioning Register . . . 18
3.14.4 Openwave Mobile Messaging . . . 18
3.14.5 AT&T Network . . . 19
3.14.6 Bitcoin Database . . . 19

4 Conclusion **20**

1 Introduction

In this document, we are going to talk about embedded databases. The report will be separated into two main parts. We will firstly define and understand what is precisely an embedded database and why we are more and more speaking about this kind of databases today. Based on each previous definitions, we will then describe a real embedded database system named Berkeley. Those explanations would be accompanied by some of the most know embedded applications : Motorola,OpenWave & AT-T Network.

1.1 An introduction to data management

Cheap, powerful computing and networking have created countless new applications that could not have existed a decade ago. The advent of the World-Wide Web, and its influence in driving the Internet into homes and businesses, is one obvious example. Equally important, though, is the shift from large, general-purpose desktop and server computers toward smaller, special-purpose devices with built-in processing and communications services. As computer hardware has spread into virtually every corner of our lives, of course, software has followed. Software developers today are building applications not just for conventional desktop and server environments, but also for handheld computers, home appliances, networking hardware, cars and trucks, factory floor automation systems, cell phones, and more. Many computer systems must store and retrieve data to track history, record configuration settings, or manage access. Data management can be very simple. In some cases, just recording configuration in a flat text file is enough. More often, though, programs need to store and search a large amount of data, or structurally complex data. Database management systems are tools that programmers can use to do this work quickly and efficiently using off-the-shelf software. Data storage is a problem dating back to the earliest days of computing.

1.2 Data access and data management

Fundamentally, database systems provide two services. The first service is data access. Data access means adding new data to the database (inserting), finding data of interest (searching), changing data already stored (updating), and removing data from the database (deleting). All databases provide these services. How they work varies from category to category, and depends on the record structure that the database supports. Each record in a database is a collection of values. For example, the record for a Web site customer might include a name, email address, shipping address, and payment information. Records are usually stored in tables. Each table holds records of the same kind. The second service is data management. Data management is more complicated than data access. Providing good data management services is the hard part of building a database system. When you choose a database system to use in an application you build, making sure it supports the data management services you need is critical. Data management services include allowing multiple users to work on the database simultaneously (concurrency), allowing multiple records to be changed instantaneously (transactions), and surviving application and system crashes (recovery).

1.3 Network Database

The "network model" is a fairly old technique for managing and navigating application data. Network databases are designed to make pointer traversal very fast. Every record stored in a network database is allowed to contain pointers to other records. Reorganization is often necessary in databases, since adding and deleting records over time will consume space that cannot be reclaimed without reorganizing. Without periodic reorganization to compact network databases, they can end up with a considerable amount of wasted space.

1.4 Clients and servers

Database vendors have two choices for system architecture. They can build a server to which remote clients connect, and do all the database management inside the server. Alternatively, they can provide a module that links directly into the application, and does all database management locally. In either case, the application developer needs some way of communicating with the database (generally, an Application Programming Interface (API) that does work in the process or that communicates with a server to get work done).

2 Embedded databases

2.1 Introduction

Our smartphones are the most striking example of the use of embedded databases. But, the exact notions of embedded systems and databases are often misunderstood and confused. The most common mistake is the confusion between Real-time systems and Embedded systems. In fact, they are absolutely not the same but are usually used together.

2.2 Definition

Definition. *An embedded database is a DBMS which is directly linked with an application. The database is completely hidden for the application's end-user and requires little maintenance.*

In other words, the main property of every embedded database is the hidden aspect. This type of architecture will built the DBMS into the application rather than provides external management. For this reason, in many cases every database interventions are directly managed from the application.

Nowadays, we also often use the term of embedded database to designated embedded applications on mobile devices. This definition is, however, not complete. In fact, real-time and embedded applications are only a small subset of embedded database systems. Embedded databases are not related to embedded platforms and can run on mobile devices but also on laptop and desktop. The only condition to be embedded is that the database is directly linked into the application.

2.3 Properties of embedded databases

In traditional entreprise, the main objectives are mainly the flexibility, scalability, functionality, etc. While we are talking about embedded systems (and databases), we are focusing on resources and processor usage which are relatively not important for traditional entreprise due to relatively cheap price of hardware.

2.3.1 Minimisation of footprint

A footprint is commonly used to designated the amount of space a particular unit of hardware or software occupies. Embedded systems and databases focus on the use of smallest footprint. As example, a typical footprint size is within the range of kilobytes to megabytes.

2.3.2 High availability

In embedded databases, we commonly do not have access to system administrator during the execution and after the system is deployed. In fact, we only have access to the database administration during the initial configuration and the ongoing maintenance would be realised by the program itself. The high availability added the considerations that the database would continue to function normally even when the hardware or the network failures within the system.

3 Berkeley Database

3.1 Introduction

So far, we have discussed database systems in general terms. It is time now to consider Berkeley DB in particular and see how it fits into the framework. The key question for this part will be the following : what kinds of applications should use Berkeley DB ?

3.2 What is Berkeley Database ?

Berkeley DB is an embedded database library that provides scalable, high performance, transaction-protected data management services to applications. It provides also a simple function-call API for data access and management. Berkeley DB is embedded because it links directly into the application and runs in the same address space as it.

3.2.1 Berkeley DB Library

As a result, no inter-process communication, either over the network or between processes on the same machine, is required for database operations. Berkeley DB provides a simple function-call API for a number of programming languages, including C, C++, Java, Perl, Tcl, Python, and PHP and all database operations happen inside the library. Multiple processes, or multiple threads in a single process, can all use the database at the same time as each uses the Berkeley DB library. Low-level services like locking, transaction logging, shared buffer management, memory management, and so on are all handled transparently by the library. The Berkeley DB library is extremely portable. It runs under almost all UNIX and Linux variants, Windows, and a number of embedded real-time operating systems. It runs on both 32-bit and 64-bit systems. It has been deployed on high-end Internet servers, desktop machines, and on palmtop computers, set-top boxes, in network switches, and elsewhere.

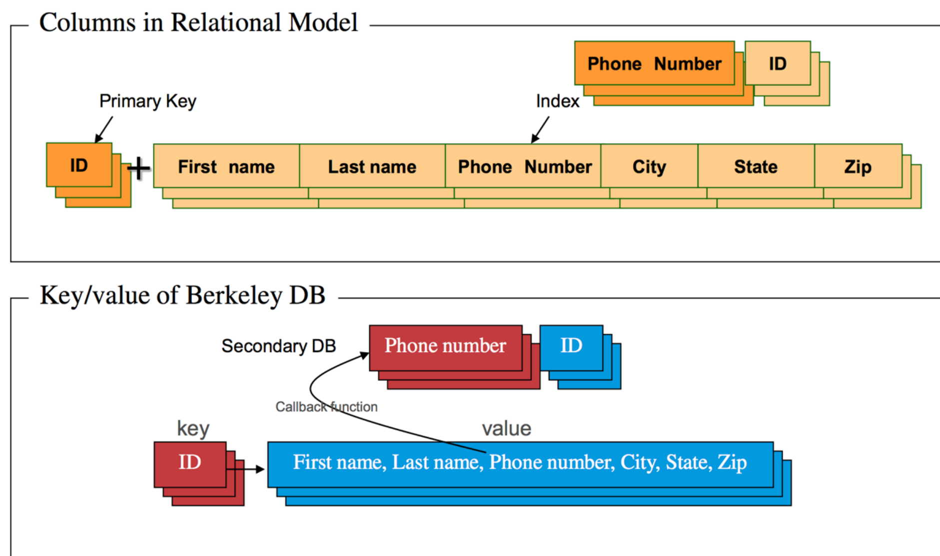
3.2.2 Berkeley DB Architecture

Once the databse is linked into the application, the end user generally does not know that there is a database present at all. Each of Berkeley's database files can contain up to 256 terabytes of data, assuming the underlying filesystem is capable of supporting files of that size. Note that applications often use multiple database files. This means that the amount of data your Berkeley DB application can manage is really limited only by the constraints imposed by your operating system, filesystem, and physical hardware. It also supports high concurrency, allowing thousands of users to operate on the same database files at the same time.

Berkeley DB generally outperforms relational and object-oriented database systems in embedded applications for a couple of reasons. First, because the library runs in the same address space, no inter-process communication is required for database operations. The cost of communicating between processes on a single machine, or among machines on a network, is much higher than the cost of making a function call. Second, because Berkeley DB uses a simple function-call interface for all operations, there is no query language to parse, and no execution plan to produce.

3.3 Records & Requests

Berkeley DB never operates on the value part of a record. Values are simply payload, to be stored with keys and reliably delivered back to the application on demand. Both keys and values can be arbitrary byte strings, either fixed-length or variable-length. As a result, programmers can put native programming language data structures into the database without converting them to a foreign record format first. Storage and retrieval are very simple, but the application needs to know what the structure of a key and a value is in advance. It cannot ask Berkeley DB, because Berkeley DB doesn't know. This is an important feature of this kind of system, and one worth considering more carefully. On the one hand, Berkeley DB cannot provide the programmer with any information on the contents or structure of the values that it stores. The application must understand the keys and values that it uses. On the other hand, there is literally no limit to the data types that can be store in a Berkeley DB database. The application never needs to convert its own program data into the data types that Berkeley DB supports. Berkeley DB is able to operate on any data type the application uses, no matter how complex. Because both keys and values can be up to four gigabytes in length, a single record can store images, audio streams, or other large data values. Large values are not treated specially in Berkeley DB. They are simply broken into page-sized chunks, and reassembled on demand when the application needs them. Unlike some other database systems, Berkeley DB offers no special support for binary large objects (BLOBs). As a first example, the following figures shows the difference between a classical SQL system and a Berkeley Key/value record. In contrast to most other database systems, Berkeley DB provides relatively simple data access services. Records in Berkeley DB are (key, value) pairs and supports only a few logical operations.



As Berkeley DB is an embedded database, it provides all the SQL request differently. All of them (Create, Insert, Update, ...) will be directly integrated into the code. A complete example of how it works is available on the official Oracle's web page. [1]

[1] http://www.oracle.com/technetwork/articles/seltzer-berkeleydb-sql-086752.html

3.4 Berkeley Database

3.4.1 Creating Database

In SQL we create the database as follows; CREATE DATABASE personnel

Instead in Berkeley DB, environment is created we place the application data. Throughout the code we refer to the environment via an environment handle i.e. DB_ENV.

```
DB_ENV *dbenv;
int ret;

/* Create the handle. */
DB_ASSERT(db_env_create(&dbenv, 0) == 0);

/*
 * If you wanted to configure the environment, you would do that here.
 * Configuraition might include things like setting a cache size,
 * specifying error handling functions, specifying (different)
 * directories in which to place your log and/or data files, setting
 * parameters to describe how many locks you'd need, etc.
 */

/* Now, open the handle. */
DB_ASSERT(dbenv->open(dbenv, "my_databases/personnel",
        DB_CREATE | DB_INIT_LOCK | DB_INIT_MPOOL | DB_INIT_TXN | DB_THREAD, 0644);
```

3.4.2 Creating Table

Now that we have created the database, now we can create some tables. In Berkeley DB, tables are referenced by handles of type DB *. For each table in the application, typically open one handle and then use that handle in one or more threads. IN SQL the database is responsible for implementing and interpreting the schema of the data, we create the table as follows; CREATE TABLE employee

In Berkeley DB, this interpretation is left up to the application. In creating the employee table, Berkeley DB will only know about the primary key and not about the different fields in the database. Firstly the database handle is created to represent the table ;

```
DB *dbp;
DB_ENV *dbenv;


/* Let's assume we've used the code from above to set dbenv. */
ASSERT(db_create(&dbp, dbenv, 0) == 0);

/*
 * Like with the environment, tables can also be configured. You
 * can specify things like comparison functions, page-size, etc.
 * That would all go here.
 */

/* Now, we'll actually open/create the primary table. */

ASSERT(dbp->open(dbp, NULL, "employee.db", NULL, DB_BTREE,
        DB_AUTO_COMMIT | DB_CREATE | DB_THREAD, 0644) == 0).
```

This call creates the table, using a B-tree as the primary index structure. The table will be materialised in the directory my_databases/personnel with the name employee.db. That file will contain only a single table and will have file system permissions as specified by the final parameter (0644). The flags that we've specified create the table in a transaction, allowing future transactional operations (DB_AUTO_COMMIT); allow creation of the table if it doesn't exist (DB_CREATE); and specify that the resulting handle can be used by multiple threads of control simultaneously (DB_THREAD).

3.4.3 Linking Table to Data

Let's consider what would happen if we wanted both a primary index on the employee id and a secondary index on the last name. In SQL we must have used the following ; CREATE INDEX lname ON employee (last_name)

In Berkeley DB, secondary indexes look like tables. We can then associate tables to make one a secondary index of the other. Let's assume that the application is going to use a C structure to contain the tuples in the employee table. Structure can be shown as below:

```
typedef struct _emp_data {
        char    lname[20];
        char    fname[15];
        float   salary;
        char    street[20];
        char    city[15];
        char    state[2];
        int     zip;
} emp_data;
```

And let's say that the employee ID is a simple integer: typedef int emp_key; In Berkeley DB, when we manipulate keys or data items, we can use a structure called a DBT. A DBT encapsulates an opaque byte-string, representing it as a pointer and a length. The pointer is referenced by the data field of a DBT, and the length is stored in the size field of the DBT. If we wanted to manipulate the key/data pair representing an employee, we would use one DBT for the emp_key and another for the emp_data.

```
DBT     key_dbt, data_dbt;
emp_key ekey;
emp_data        edata;

memset(&key_dbt, 0, sizeof(key_dbt));
memset(&data_dbt, 0, sizeof(data_dbt));

/*
 * Now make the key and data DBT's reference the key and data
 * variables.
 */
key_dbt.data = &ekey;
key_dbt.size = sizeof(ekey);

data_dbt.data = &edata;
data_dbt.size = sizeof(edata);
```

3.4.4 Inserting Data to Table

Now for inserting data to the table in Berkeley database, Let's assume that we have table opened from last time and we have a database handle dbp that references the employee table. Now we insert a new employee Mickey Mouse with key, salary and address ;

```
DB *dbp;
DBT key_dbt, data_dbt;
emp_data edata;
emp_key ekey;

/* Put the value into the employee key. */
ekey = 00010002;

/* Initialize an emp_data structure. */
strcpy(edata.lname, "Mouse");
strcpy(edata.fname, "Mickey");
edata.salary = 1000000.00;
strcpy(edata.street, "Main Street");
strcpy(edata.city, "Disney Land");
strcpy(edata.state, "CA");
edata.zip = 98765;

/* Initialize DBTs */
memset(&key_dbt, 0, sizeof(key_dbt));
memset(&data_dbt, 0, sizeof(data_dbt));

/* Now, assign key and data values to DBTs. */
key->data = &ekey;
key->size = sizeof(ekey);
data->data = &edata;
data->size = sizeof(edata);

/* Finally, put the data into the database. */
ASSERT(dbp->put(dbp, NULL, &key_dbt, &data_dbt, DB_AUTO_COMMIT) == 0);
```

3.4.5 Updating Data to Table

Data can be updated in Berkeley DB if we know exactly which bytes of a data item is wished to replace and use the update command to do it.In order to do this, we need to introduce the notion of a cursor. A cursor represents a position in a table. It lets iterate over the table and maintain the notion of a current item that you can then manipulate.

Creating a cursor in Berkeley DB is easy?it's a method off of a database handle:

```
DBC *dbc;
    DB      *dbp;
     ASSERT(dbp->cursor(dbp, NULL, 0) == 0);
```

Next we can change the salary (handle the "SET salary=2000000" part of the clause)

```
/* Change the salary. */
       edata = data_dbt->data;
       edata.salary = 2000000;
```

Finally, apply the UPDATE portion of the SQL statement:

```
dbc->c_put(dbc, &key_dbt, &data_dbt, DB_CURRENT)
```

```
/* We don't want the data, we just want to position the cursor. */
memset(&data_dbt, 0, sizeof(data_dbt));
data_dbt->flags = DB_DBT_PARTIAL;
data_dbt->dlen = 0;

/* Position the cursor on Mickey's record */
dbc->c_get(dbc, &key_dbt, &data_dbt, DB_SET);

/*
 * Now, prepare for a partial put. Note that the DBT has already
 * been initialized for partial operations.  We need to specify
 * where in the data item we wish to place the new bytes and
 * how many bytes we'd like to replace.
 */
salary = 2000000.00;

/* The DBT contains just the salary information. */
data_dbt->data = &salary;
data_dbt->size = sizeof(salary);

/*
 * dlen and doff tell Berkeley DB where to place this information
 * in the record.  dlen indicates how many bytes we are replacing --
 * in this case we're replacing the length of the salary field in
 * the structure (sizeof(emp_data.salary)).  doff indicates where
 * in the data record we will place these new bytes -- we need to
 * compute the offset of the salary field.
 */
data_dbt->dlen = sizeof(emp_data.salary);
data_dbt->doff = ((char *)&edata.salary - (char *)&edata);

/* Now, put the record back with the new data. */
dbc->c_put(dbc, &key_dbt, &data_dbt, DB_CURRENT);
```

3.4.6 Retrieving Data from the Table

Till now we know how to add data to the table, now it's time to retrieve the data from the table. In SQL, looking up values by its primary key;

```
SELECT * FROM employees  WHERE id=0010002
```

In Berkeley database we have seen that we can do that with the help of the cursor as follows;

```
DBT             key_dbt, data_dbt;
emp_data        *edata;
emp_key ekey;

/* We'd like to look up Mickey's key. */
emp_key = 0010002;
memset(&key_dbt, 0, sizeof(key_dbt));
key_dbt.data = &emp_key;
key_dbt.size = sizeof(emp_key);

/*
 * We want the data returned, so we don't need to initialize the
 * employee data data structure.
 */
memset(&data_dbt, 0, sizeof(data_dbt));

/* Now, set the cursor to the record with the key emp_key. */
dbc->c_get(dbc, &key_dbt, &data_dbt, DB_SET);
```

In the above code we have used cursor operation because we also wanted to update the record,

But we can also just retrieve the data without using the cursor. We just need to use the get method off of the dbp handle;

```
DBT             key_dbt, data_dbt;
emp_data        *edata;
emp_key ekey;

/* We'd like to look up Mickey's key. */
emp_key = 0010002;
memset(&key_dbt, 0, sizeof(key_dbt));
key_dbt.data = &emp_key;
key_dbt.size = sizeof(emp_key);

/*
 * We want the data returned, so we don't need to initialize the
 * employee data data structure.
 */
memset(&data_dbt, 0, sizeof(data_dbt));

/* Now, use the dbp method. */
dbp->get(dbp, NULL, &key_dbt, &data_dbt, 0);
```

3.4.7 Example

```
int
display(database)
        char *database;
{
        DB *dbp;
        DBC *dbcp;
        DBT key, data;
        int close_db, close_dbc, ret;

        close_db = close_dbc = 0;

        /* Open the database. */
        if ((ret = db_create(&dbp, NULL, 0)) != 0) {
                fprintf(stderr,
                    "%s: db_create: %s\n", progname, db_strerror(ret));
                return (1);
        }

        /* Turn on additional error output. */
        dbp->set_errfile(dbp, stderr);
        dbp->set_errpfx(dbp, progname);

        /* Open the database. */
        if ((ret =
            dbp->open(dbp, NULL, database, NULL, DB_UNKNOWN, DB_RDONLY, 0)) != 0) {
                dbp->err(dbp, ret, "%s: DB->open", database);
                goto err;
        }

        /* Acquire a cursor for the database. */
        if ((ret = dbp->cursor(dbp, NULL, &dbcp, 0)) != 0) {
                dbp->err(dbp, ret, "DB->cursor");
                goto err;
        }

        /* Initialize the key/data return pair. */
        memset(&key, 0, sizeof(key));
        memset(&data, 0, sizeof(data));

        /* Walk through the database and print out the key/data pairs. */
        while ((ret = dbcp->c_get(dbcp, &key, &data, DB_NEXT)) == 0)
                printf("%.*s : %.*s\n",
                    (int)key.size, (char *)key.data,
                    (int)data.size, (char *)data.data);
        if (ret != DB_NOTFOUND) {
                dbp->err(dbp, ret, "DBcursor->get");
                goto err;
        }

err:    if (close_dbc && (ret = dbcp->c_close(dbcp)) != 0)
                dbp->err(dbp, ret, "DBcursor->close");
        if (close_db && (ret = dbp->close(dbp, 0)) != 0)
                fprintf(stderr,
                    "%s: DB->close: %s\n", progname, db_strerror(ret));
        return (0);
}
```

3.5 Data Access Services

Berkeley DB applications can choose the storage structure that best suits the application. Berkeley DB supports hash tables, Btrees, simple record-number-based storage, and persistent queues. Programmers can create tables using any of these storage structures, and can mix operations on different kinds of tables in a single application. Hash tables are generally good for very large databases that need predictable search and update times for random-access records. Btrees are better for range-based searches, as when the application needs to find all records with keys between some starting and ending value. Record-number-based storage is natural for applications that need to store and fetch records, but that do not have a simple way to generate keys of their own. A good example is on-line purchasing systems. Orders can enter the system at any time, but should generally be filled in the order in which they were placed.

3.6 Data management services

Berkeley DB offers important data management services, including concurrency, transactions, and recovery. All of these services work on all of the storage structures. Many users can work on the same database concurrently. Berkeley DB handles locking transparently, ensuring that two users working on the same record do not interfere with one another. Berkeley DB uses a technique called two-phase locking to be sure that concurrent transactions are isolated from one another, and a technique called writeahead logging to guarantee that committed changes survive application, system, or hardware failures. When an application starts up, it can ask Berkeley DB to run recovery. Recovery restores the database to a clean state, with all committed changes present, even after a crash.

3.7 The Berkeley DB products

- **Berkeley DB Data Store:** The product is an embeddable, high-performance data store. This product supports multiple concurrent threads of control, including multiple processes and multiple threads of control within a process.

- **Berkeley DB Concurrent Data Store:** The product adds multiple-reader, single writer capabilities to the Berkeley DB Data Store product. This product provides built-in concurrency and locking feature.

- **Berkeley DB Transactional Data Store:** It is intended for applications that require industrial-strength database services, including excellent performance under high-concurrency workloads of read and write operations, the ability to commit or roll back multiple changes to the database at a single instant, and the guarantee that in the event of a catastrophic system or hardware failure, all committed database changes are preserved.

- **Berkeley DB High Availability:** The product adds support for data replication. A single master system handles all updates, and distributes these updates to multiple replicas. If the master system fails for any reason, one of the replicas takes over as the new master system, and distributes updates to the remaining replicas.

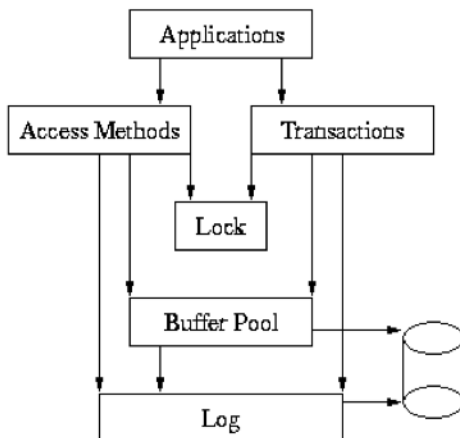
3.8 Available access methods

- **Btree :** This access method is an implementation of a sorted, balanced tree structure. Searches, insertions, and deletions in the tree all take $O(log_b N)$ time.

- **Hash :** This access method data structure is an implementation of Extended Linear Hashing.

- **Queue:** This access method stores fixed-length records with logical record numbers as keys. It is designed for fast inserts at the tail and has a special cursor consume operation that deletes and returns a record from the head of the queue. The Queue access method uses record level locking.

- **Recno :** The Recno access method stores both fixed and variable-length records with logical record numbers as keys, optionally backed by a flat text (byte stream) file.



In this model, the application makes calls to the access methods and to the Transaction subsystem. The access methods and Transaction subsystems in turn make calls into the Memory Pool, Locking and Logging subsystems on behalf of the application.

3.9 Berkeley DB subsystems

- **Access Methods :** This methods subsystem provides general-purpose support for creating and accessing database files formatted as Btrees, Hashed files, and Fixed- and Variable-length records. These modules are useful in the absence of transactions for applications that need fast formatted file support.

- **Memory Pool :** This subsystem is the general-purpose shared memory buffer pool used by Berkeley DB. This is the shared memory cache that allows multiple processes and threads within processes to share access to databases. This module is useful outside of the Berkeley DB package for processes that require portable, page-oriented, cached, shared file access.

- **Transaction :** This subsystem allows a group of database changes to be treated as an atomic unit so that either all of the changes are done, or none of the changes are done. This module is useful outside of the Berkeley DB package for processes that want to transaction-protect their own data modifications.

- **Locking :** This subsystem is the general-purpose lock manager used by Berkeley DB. This module is useful outside of the Berkeley DB package for processes that require a portable, fast, configurable lock manager.

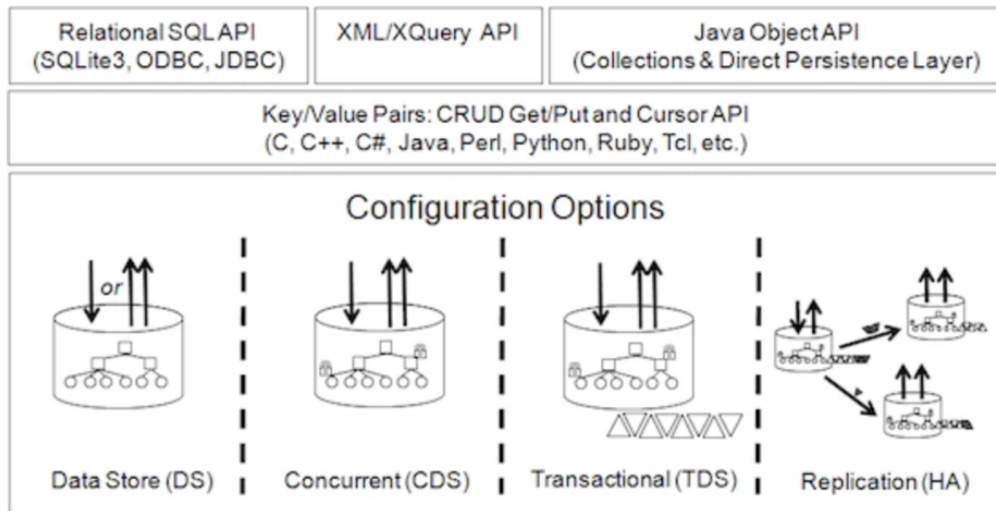
- **Logging :** This subsystem is the write-ahead logging used to support the Berkeley DB transaction model. It is largely specific to the Berkeley DB package, and unlikely to be useful elsewhere except as a supporting module for the Berkeley DB transaction subsystem.

3.10 Programming model

Berkeley DB is a database library, in which the library is linked into the address space of the application using it. The code using Berkeley DB may be a standalone application or it may be a server providing functionality to many clients via inter-process or remote-process communication. In the standalone application model, one or more applications link the Berkeley DB library directly into their address spaces. This model provides significantly faster access to the database functionality, but implies trust among all threads of control sharing the database environment because they will have the ability to read, write and potentially corrupt each other's data. In the client-server model, developers write a database server application that accepts requests via some form of IPC/RPC, and issues calls to the Berkeley DB interfaces based on those requests. In this model, the database server is the only application linking the Berkeley DB library into its address space.

3.11 Programmatic APIs

The Berkeley DB subsystems can be accessed through interfaces from multiple languages. The standard library interface is ANSI C. Applications can also use Berkeley DB via C++ or Java, as well as from scripting languages. Environments can be shared among applications written by using any of these APIs. For example, you might have a local server written in C or C++, a script for an administrator written in Perl or Tcl, and a Web-based user interface written in Java – all sharing a single database environment.



Oracle Berkeley DB fits where you need it regardless of programming language, hardware platform, or storage media. Berkeley DB APIs are available in almost all programming languages including ANSI-C, C++, Java, Csharp, Perl, Python, Ruby and Erlang. There is a pure-Java version of the Berkeley DB library designed for products that must run entirely within a Java Virtual Machine (JVM). Oracle Berkeley DB is tested and certified to compile and run on all modern operating systems including Solaris, Windows, Linux, Android, Mac OS/X, BSD, iPhone OS, VxWorks, and QNX.

3.12 Security

The following are security issues that should be considered when writing Berkeley DB applications:

- **Database environment permissions:** The directory used as the Berkeley DB database environment should have its permissions set to ensure that files in the environment are not accessible to users without appropriate permissions.
- **File permissions:** By default, Berkeley DB always creates files readable and writable by the owner and the group. The group ownership of created files is based on the system and directory defaults, and is not further specified by Berkeley DB.
- **Temporary backing files:** If an unnamed database is created and the cache is too small to hold the database in memory, Berkeley DB will create a temporary physical file to enable it to page the database to disk as needed.

3.13 Encryption

Berkeley DB supports encryption using the Rijndael/AES (also known as the Advanced Encryption Standard and Federal Information Processing Standard (FIPS)) algorithm for encryption or decryption. The algorithm is configured to use a 128-bit key. Berkeley DB uses a 16-byte initialization vector generated using the Mersenne Twister. All encrypted information is additionally check summed using the SHA1 Secure Hash Algorithm, using

a 160-bit message digest. The encryption support provided with Berkeley DB is intended to protect applications from an attacker obtaining physical access to the media on which a Berkeley DB database is stored, or an attacker compromising a system on which Berkeley DB is running but who is unable to read system or process memory on that system. The encryption support provided with Berkeley DB will not protect applications from attackers able to read system memory on the system where Berkeley DB is running. The only encrypted parts of a database environment are its databases and its log files. Specifically, the Shared memory regions supporting the database environment are not encrypted. For this reason, it may be possible for an attacker to read some or all of an encrypted database by reading the on-disk files that back these shared memory regions. While all user data is encrypted, parts of the databases and log files in an encrypted environment are maintained in an unencrypted state. Specifically, log record headers are not encrypted, only the actual log records. Additionally, database internal page header fields are not encrypted. Log records distributed by replication master to replicated clients are transmitted to the clients in unencrypted form. If encryption is desired in a replicated application, the use of a secure transport is strongly suggested.

3.14 Berkeley Database Customers

3.14.1 Amazon.com

Amazon's website needed to be highly responsive, massively scalable, and always-on in order to give customers a highly personalized shopping experience. Amazon selected Berkeley DB to serve as a high speed cache in front of their massive products and offerings database built on Oracle Database. When customers view Amazon's pages, all the product information, pricing, recommendations, user reviews, etc. are retrieved from Berkeley DB running as a cache in front of an Oracle Database backend.

3.14.2 Motorola Wireless network

Motorola needed carrier-class performance, scalability and reliability. In order to meet 99.999% uptime requirements, the data manager needed to have the ability to automatically restart after a failure with very fast start-up and shutdown speed. Motorola selected Berkeley DB to store configuration data such as radio ID and user information for the WNG. Berkeley DB met Motorola's requirements for high performance and ability to operate unattended and recover quickly from system crashes.

3.14.3 Cisco Systems Broadband Provisioning Register

Cisco's needed to manage up to 5 million networked devices and 150 configuration change transactions/second. BPR needed an embedded data manager which was fast, scalable, reliable and cost-effective. Berkeley DB was used to replace an object-oriented database in BPR. The result was faster, more reliable and saved Cisco $50,000/CPU.

3.14.4 Openwave Mobile Messaging

Openwave customers demanded a high performance, low TCO solution. Openwave uses Berkeley DB for the carrier-class message store and LDAP directory that are the basis for

all of its messaging products, including Email Mx, MMSC and IP Voicemail. By eliminating the operator's database licensing costs, extra hardware costs and DBA costs, the Berkeley DB based solution provided lower acquisition and operating costs for Ovenware's customers.

3.14.5 AT&T Network

AT&T's software for broadband networks originally used an object-oriented database, which was too slow, complex and expensive. Key data manager requirements were : Speed to handle the flood of requests following restoration of power after an electrical outage and Strict transactional integrity to ensure the accuracy of communication with billing and customer care systems. Berkeley DB was chosen for use as the backing store for all provisioning operations. Without sacrificing transactional integrity, Berkeley DB provided very fast performance, resulting in significant hardware savings for AT&T's customers.

3.14.6 Bitcoin Database

Bitcoin is a form of digital currency, created and held electronically. No one controls it. Bitcoins aren't printed, like dollars or euros, they're produced by people, and increasingly businesses, running computers all around the world, using software that solves mathematical problems. Till Late 2013 Bitcoin used Berkeley's database, but due to few issues Bitcoin migrated all its data to Level DB. Bitcoin is a distributed network. All transactions are stored on a public ledger called block chain. The block chain entries are secured with cryptography. A hash is generated for each block that also contains the previous block hash, so, when a block is confirmed, with the hash, it cannot be altered. All nodes of the network (miners and wallets), have a copy of the block chain, which is always updated. So, there is no database and no administrator to care about the database.
In particular:

1. BDB is much slower for the usage (large atomic batch writes, small random reads).

2. There were reports of database corruption as well with BDB, at a time when it was used far less intensively than LevelDB is now.

3. BDB is very painful to upgrade. It was designed for setups where a database upgrade only happened with professional supervision. In particular, the write log files created for durability were sometimes not readable by later versions. This is the reason why Bitcoin Core releases have for years stuck with BDB 4.8 for the wallet.

4. BDB has many resource limits that need configuration, where inappropriately chosen values may cause network-wide failures.

While database corruptions are reported relatively frequently these days, my belief is that it's mostly hardware failures or driver issues. Bitcoin Core tends to stress disks, memory and CPUs far more than most software, which makes otherwise invisible issues pop up.

4 Conclusion

Berkeley DB offers a unique collection of features, targeted squarely at software developers who need simple, reliable database management services in their applications. Good design and implementation and careful engineering throughout make the software better than many other systems.

References

[1] Oracle Embedded Databases, Oracle, http://www.oracle.com/us/technologies/embedded-datasheet-166556.pdf.

[2] Embedded Database Overview, Dan Outcalt & Gabe Stanek, http://www.nocoug.org/download/2008-05/Embedded_Database_Overview_UG_SFO.ppt

[3] The Case for Embedded Databases, Asif Ali, https://medium.com/@azifali/the-case-for-embedded-databases-5b7d5b57e736

[4] Challenges in Embedded Database System Administration, Margo Seltzer, Michael Olson, https://www.eecs.harvard.edu/margo/papers/embedded99/paper.html

[5] Embedded database, Wikipedia, https://en.wikipedia.org/wiki/Embedded_database

[6] Embedded Databases, Chris Crupp, http://embedded-databases.com

[7] Oracle Berkeley Database 9.2, Oracle, http://www.oracle.com/technetwork/products/berkeleydb/berkeley-db-datasheet-132390.pdf

[8] Oracle Berkeley Database Products, Oracle, http://www.oracle.com/technetwork/products/berkeleydb/learnmore/berkeley-db-family-datasheet-132751.pdf

[9] Oracle Berkeley DB 12c, Oracle, http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

[10] Orcale Berkeley Documentation, Oracle, https://docs.oracle.com/cd/E17076_05/html/index.html

[11] Guide to Oracle Berkeley DB for SQL Developers, Oracle, http://www.oracle.com/technetwork/articles/seltzer-berkeleydb-sql-086752.html