



Design and Implementation of Elastic Key Value Data Store

Hareesh B. N., PG Student, CSE, AIET, Moodbidri

Venkatesh, Senior Associate Professor, CSE, AIET, Moodbidri

Abstract: Berkeley DB is an open source embedded database library that provides scalable, high-performance, transaction-protected data management services to applications. Berkeley DB provides a simple function-call API for data access and management. By "open source," we mean that Berkeley DB is distributed under a license that conforms to the Open Source Definition. This license guarantees that Berkeley DB is freely available for use and redistribution in other open source products. Sleepycat Software sells commercial licenses for redistribution in proprietary applications, but in all cases the complete source code for Berkeley DB is freely available for download and use. Berkeley DB is embedded because it links directly into the application. It runs in the same address space as the application. 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. 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 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. Once Berkeley DB is linked into the application, the end user generally does not know that there's a database present at all. Berkeley DB is scalable in a number of respects. The database library itself is quite compact (under 300 kilobytes of text space on common architectures), but it can manage databases up to 256 terabytes in size. It also supports high concurrency, with thousands of users operating on the same database at the same time. Berkeley DB is small enough to run in tightly constrained embedded systems, but can take advantage of gigabytes of memory and terabytes of disk on high-end server machines.

Keywords: HBASE, Berkeley DB, Elastic key value store

I. INTRODUCTION

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, and more. While these operating environments are diverse, the problems that software engineers must solve in them are often strikingly similar. Most systems must deal with the outside world, whether that means communicating with users or controlling machinery. As a result, most need some sort of I/O system. Even a simple, single-function system generally needs to handle multiple tasks, and so needs some kind of operating system to schedule and manage control threads. Also, 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. Of course, database management systems have been around for a long time. Data storage is a problem dating back to the earliest days of computing. Software developers can choose from hundreds of good, commercially-available database systems. The problem is selecting the one that best solves the problems that their applications face. Figure 1: HBase architecture [3]. See Figure 2 for a high-level comparison with the architecture of HBase-BD.

What is Berkeley DB?

So far, we've discussed database systems in general terms. It's time now to consider Berkeley DB in particular and see how it fits into the framework we have introduced. The key question is? what kinds of applications should use Berkeley DB? Berkeley DB is an open source embedded database library that provides scalable, high-performance, transaction-protected data management services to applications. Berkeley DB provides a simple function-call API for data access and management. By "open source," we mean that Berkeley DB is distributed under a license that conforms to the Open Source Definition. This license guarantees that Berkeley DB is freely available for use and redistribution in other open source products. Sleepycat Software sells commercial licenses for redistribution in proprietary applications, but in all cases the complete



source code for Berkeley DB is freely available for download and use.

Berkeley DB is embedded because it links directly into the application. It runs in the same address space as the application. 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. 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 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. Once Berkeley DB is linked into the application, the end user generally does not know that there's a database present at all. Berkeley DB is scalable in a number of respects. The database library itself is quite compact (under 300 kilobytes of text space on common architectures), but it can manage databases up to 256 terabytes in size. It also supports high concurrency, with thousands of users operating on the same database at the same time. Berkeley DB is small enough to run in tightly constrained embedded systems, but can take advantage of gigabytes of memory and terabytes of disk on high-end server machines.

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.

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. Hash tables allow users to ask, "Does this key exist?" or to fetch a record with a known key. Hash

tables do not allow users to ask for records with keys that are close to a known key.

Btrees are better for range-based searches, as when the application needs to find all records with keys between some starting and ending value. Btrees also do a better job of exploiting locality of reference. If the application is likely to touch keys near each other at the same time, the Btrees work well. The tree structure keeps keys that are close together near one another in storage, so fetching nearby values usually doesn't require a disk access. 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. In a record number table, the record number is the key for the record. Berkeley DB will generate these record numbers automatically. Queues are well-suited for applications that create records, and then must deal with those records in creation order. 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.

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. The library provides strict ACID transaction semantics, by default. However, applications are allowed to relax the isolation guarantees the database system makes.

Multiple operations can be grouped into a single transaction, and can be committed or rolled back atomically. Berkeley DB uses a technique called two-phase locking to be sure that concurrent transactions are isolated from one another, and a technique called write-ahead 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. The database is guaranteed to be consistent and all committed changes are guaranteed to be present when recovery completes. Some applications need fast, single-user, non-transactional Btree data storage. In that case, the application can disable the locking and transaction systems, and will not incur the overhead of locking or logging. If an application needs to support multiple concurrent users, but doesn't need transactions, it can turn on locking without transactions. Applications that need concurrent, transaction-protected database access can enable all of the subsystems.

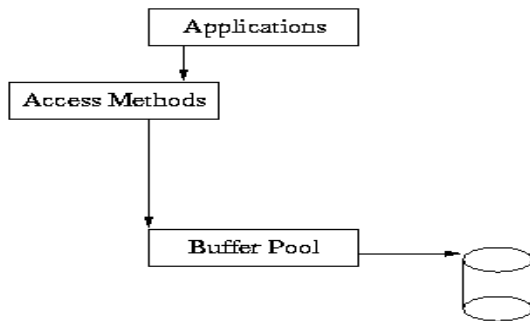
II. GENERAL SPECIFICATIONS

Application code that uses only the Berkeley DB access methods might appear as follows:



```
switch (ret = dbp->put(dbp, NULL, &key, &data, 0)) {
case 0:
    printf("db: %s: key stored.\n", (char *)key.data);
    break;
default:
    dbp->err(dbp, ret, "dbp->put");
    exit (1);
}
```

The underlying Berkeley DB architecture that supports this is:



Small

As you can see from this diagram, the application makes calls into the access methods, and the access methods use the underlying shared memory buffer cache to hold recently used file pages in main memory.

When applications require recoverability, their calls to the Access Methods must be wrapped in calls to the transaction subsystem. The application must inform Berkeley DB where to begin and end transactions, and must be prepared for the possibility that an operation may fail at any particular time, causing the transaction to abort.

An example of transaction-protected code might appear as follows:

```
for (fail = 0;;) {
/* Begin the transaction. */
if ((ret =dbenv->txn_begin(dbenv, NULL, &tid, 0))!= 0) {
    dbenv->err(dbenv, ret, "dbenv->txn_begin");
    exit (1);
}
/* Store the key. */
switch (ret = dbp->put(dbp, tid, &key, &data, 0)) {
case 0:
    /* Success: commit the change. */
    printf("db: %s: key stored.\n", (char *)key.data);
    if ((ret = tid->commit(tid, 0)) != 0) {
        dbenv->err(dbenv, ret, "DB_TXN->commit");
        exit (1);
    }
    return (0);
case DB_LOCK_DEADLOCK:
default:
    /* Failure: retry the operation. */
    if ((t_ret = tid->abort(tid)) != 0) {
        dbenv->err(dbenv, t_ret, "DB_TXN->abort");
        exit (1);
    }
}
```

```

}
if (++fail == MAXIMUM_RETRY)
    return (ret);
    continue;
}
}
```

In this example, the same operation is being done as before; however, it is wrapped in transaction calls. The transaction is started with `DB_ENV->txn_begin` and finished with `DB_TXN->commit`. If the operation fails due to a deadlock, the transaction is aborted using `DB_TXN->abort`, after which the operation may be retried.

There are actually five major subsystems in Berkeley DB, as follows:

Access Methods

The access 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. See `DB->open` and `DB->cursor` for more information.

Memory Pool

The Memory Pool 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

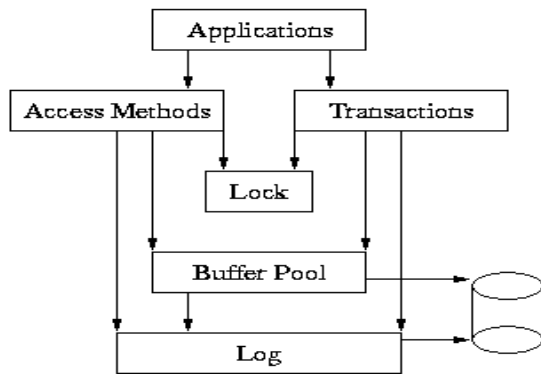
The Transaction subsystem allows a group of database changes to be treated as an atomic unit so that either all of the changes are done, or none of the changes are done. The transaction subsystem implements the Berkeley DB transaction model. This module is useful outside of the Berkeley DB package for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

Locking

The Locking 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

The Logging 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.



Here is a more complete picture of the Berkeley DB library:

Large

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.

The underlying subsystems can be used independently by applications. For example, the Memory Pool subsystem can be used apart from the rest of Berkeley DB by applications simply wanting a shared memory buffer pool, or the Locking subsystem may be called directly by applications that are doing their own locking outside of Berkeley DB. However, this usage is not common, and most applications will either use only the access methods subsystem, or the access methods subsystem wrapped in calls to the Berkeley DB transaction interfaces.

HBase-BDB is the result of re-engineering HBase to replace its LSM-Tree implementation over an HDFS backend with the use of a collection of Berkeley Database (BDB) Java Edition (JE) [11] storage managers over local file systems. This design leverages the log structured [12] B+ tree implementation at the core of BDB-JE. Since data at each node are stored and organized as a full log there is no need for a distinct WAL, eliminating one cause of write amplification in HDFS. Additionally, the aggressive storage reorganization needed in HBase-HDFS to improve random read performance is not needed in HBase-BDB. BDB-JE still needs to reorganize its storage layout to reclaim space; however this is a far less aggressive operation compared to HBase HDFS. Furthermore, HBase-BDB maintains elasticity properties (which HBase-HDFS achieves through the use of HDFS) by re-implementing two key operations (split, move) with minimum service disruption. The use of local file systems allows any node to reorganize its data without affecting any other node or any of its replicas. In the following sections we describe more details on the design of HBase- BDB. Figure 2b depicts the structure of the BDB B+ tree, which consists of Internal Nodes (IN), Bottom Internal Nodes (BIN), and Leaf Nodes (LN) which hold the (key, offset-to disk for value) pair. A single instance of BDB can manage multiple databases (a BDB database maps to an HBase region in HBase-BDB) writing everything to a logical (per database) log, which is the only on-disk structure. A log is implemented as a number of physical files of configurable size. Below we present the schema mapping of HBase in HBase-BDB and its basic operations.

III. FIGURES

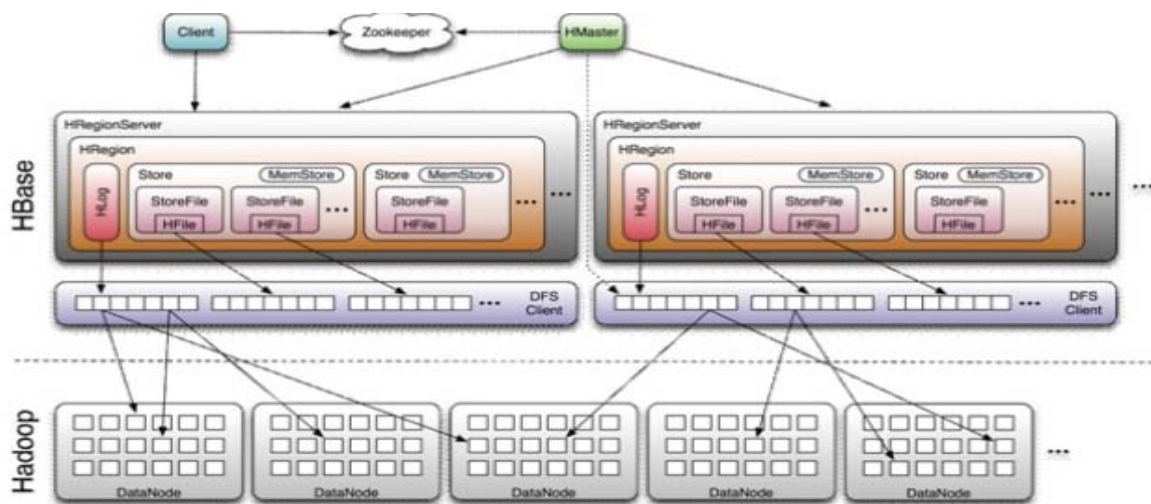


Figure 1: HBase architecture [3]. See Figure 2 for a high-level comparison with the architecture of HBase-BD

IV. CONCLUSION

HBase-BDB, a distributed key-value store that shares HBase's data model and data distribution mechanisms but departs from it in the use of a log structured B+-tree

indexed storage back end over locally attached files systems. With the use of a log structured key value store combined with novel elasticity mechanisms HBase-BDB is able to improve over HBase in random read and write YCSB workloads by 30% and 85% respectively. HBase-



BDB lags behind HBase only in random scans; these however are only a small share of overall operations in popular workloads such as e-mail, SMS, Chat, etc. Support for elasticity in HBase-BDB is shown to be effective in TPCC experiments yielding similar availability and performance impact to what is achievable with HBase-HDFS.

REFERENCES

- [1] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end arguments in system design," *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 4, pp. 277–288, 1984.
- [2] M. Welsh and D. Culler, "Virtualization considered harmful: OS design directions for well-conditioned services," in *Proceedings of the 8th Workshop on Hot Topics in Operating Systems*, 2001, pp. 139–144.
- [3] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of HDFS under HBase: a Facebook messages case study," in *Proceedings of USENIX Conference on File and Storage Technologies*, 2014, pp. 199–212.
- [4] P. Shetty, R. P. Spillane, R. Malpani, B. Andrews, J. Seyster, and E. Zadok, "Building workload-independent storage with VT-trees." In *Proceedings of USENIX Conference on File and Storage Technologies*, 2013, pp. 17–30.
- [5] O. Rodeh, J. Bacik, and C. Mason, "BTRFS: The Linux B-tree filesystem," *ACM Transactions on Storage*, vol. 9, no. 3, p. 9, 2013
- [6] H. T. Vo, S. Wang, D. Agrawal, G. Chen, and B. C. Ooi, "LogBase: a scalable log-structured database system in the cloud," *Proceedings of the VLDB Endowment*, vol. 5, no. 10, pp. 1004–1015, 2012.