Survey of software architecture V0.0

Name of system: Berkeley DB
Reviewer: David Dietrich
Date: November 21, 2010
Author of software: Margo Seltzer and Keith Bostic
Author of book chapter: Margo Seltzer and Keith Bostic
Five star rating of book chapter: ****

Purpose of system:

Berkeley DB is a software library that provides fast, flexible, reliable and scalable data management. Originally Unix was a proprietary operating system of AT&T and Berkeley DB was created with the goal of replacing several proprietary in-memory libraries of Unix with improved, freely available ones. Berkeley DB has since evolved into the most widely used database toolkit in the world[1].

This first version of Berkeley DB remained the same for several years until 1996 when the authors were approached by Netscape to create a fully transactional version of their software. The outcome the first transactional version of Berkeley DB. This also led to formation of Sleepycat Software by Margo Seltzer and Keith Bostic which developed Berkeley DB until the company was purchased by Oracle.

Unlike relational databases such as Oracle DBMS, Berkeley DB does not require a separate server. All of the data is stored in memory, but can be read from or written to a file when required. The data is stored as Key-Value pairs in one of four access methods: B+ Tree, Hash Table, Record Numbers and Queue[4]. The B+ Tree and Hash Table methods are the most common and are the correct choice for almost all implementations. Until version 5 data was only accessible by using accessing it with the Key, or through a sequential read. In version 5 SQL support was added.

Despite being stored in memory databases in Berkeley DB are still accessible to different processes. This gives it a major benefit over the extremely popular embedded database SQLite[2]. The method of doing this is explained in more detail in the Concurrency section of the survey.

Berkeley DB has many differences from a relational database such as Oracle that make it attractive in certain situations. Because it is a single small application it can be used on systems with small amounts of memory (such as embedded systems) where large relational databases are unreasonable. It also provides extremely fast methods of data insertion and querying which make it possible to use with real time systems[3].

Oracle acquired Sleepycat Software in 2006 which led to Berkeley DB becoming one of Oracle's products. Thus it is not exactly open source in the traditional sense that it has public source control and is governed by the community. It does not have a public source code repository, however the code is freely available. There is also a method in place for submitting patches, but the source code is completely under the control of Oracle.

[1]Seltzer, M., Bostic, K., “The Architecture of Open-Source Applications: Berkeley DB”
[2]SQLite Website: http://www.sqlite.org/
[3]Oracle Corporation. 2010, ”Oracle Berkeley DB: Performance Metrics and Benchmarks”.
[4]Seltzer, M. et al, “Berkeley DB”. 1999.

Basic metrics

KLOC: 423 KLOC of C

Project start-up: 1984

Number of major releases: 5

Number of developers: Unknown. With the acquisition of Berkeley DB by Oracle it is now developed by an internal Oracle team. As the system does not have a public source control repository it is impossible to estimate the number of developers.

Size of user community or number of installations: Millions

Major stakeholders: Oracle, Google

Use of concurrency: Berkeley DB supports sharing databases between any number of running processes. This is implemented by having all data structures important to the functionality of the system be stored in shared memory. This does load to a caveat. Berkeley DB does not handle memory access through pointers, but instead does so by taking the base address of the shared memory and using offsets to access it.

Implementation language: C

Supporting software: None

High level architecture

Diagram of software architecture

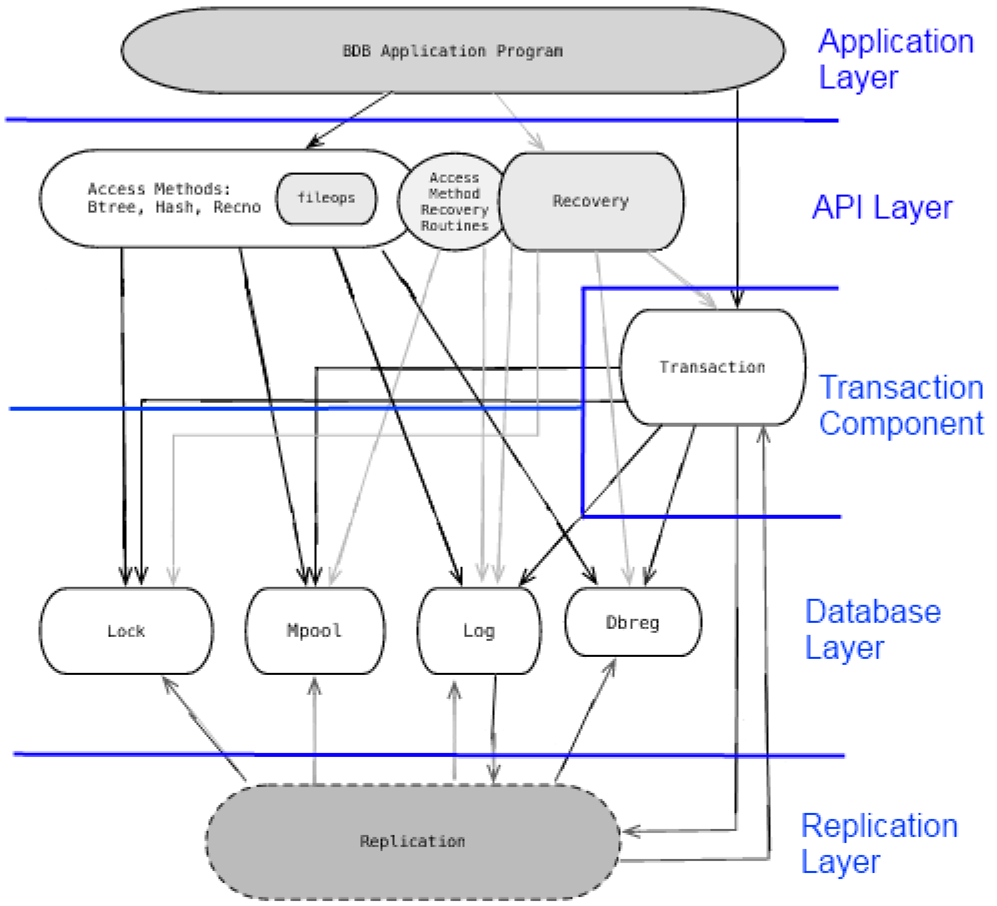

Figure 1: Concrete Architecture of Berkeley DB 5.0.21

Every box in this diagram is a separate component of the Berkeley DB Architecture. The boxes each represent different components of the system.

The Recovery components are coloured in light-grey and are not required for the system to be operational, they provide additional functionality. The rest of the boxes represent those systems that are necessary to the operation of Berkeley DB.

The dark grey box with the dotted border is the Replication component. And lastly the box on the top layer represents the application calling into Berkeley through the API.

The grey arrows represent those calls to and from the Recovery subsystem. And the rest of the arrows represent normal calls within the system.

I have also placed borders between each layer for the sake of clarity. I unfortunately did not have room for multiple diagrams to clarify this. A further explanation of this layering occurs in the Architectural Style section below.

High level scenarios

As a high level scenario imagine an application wishes to open a database and write a piece of data.

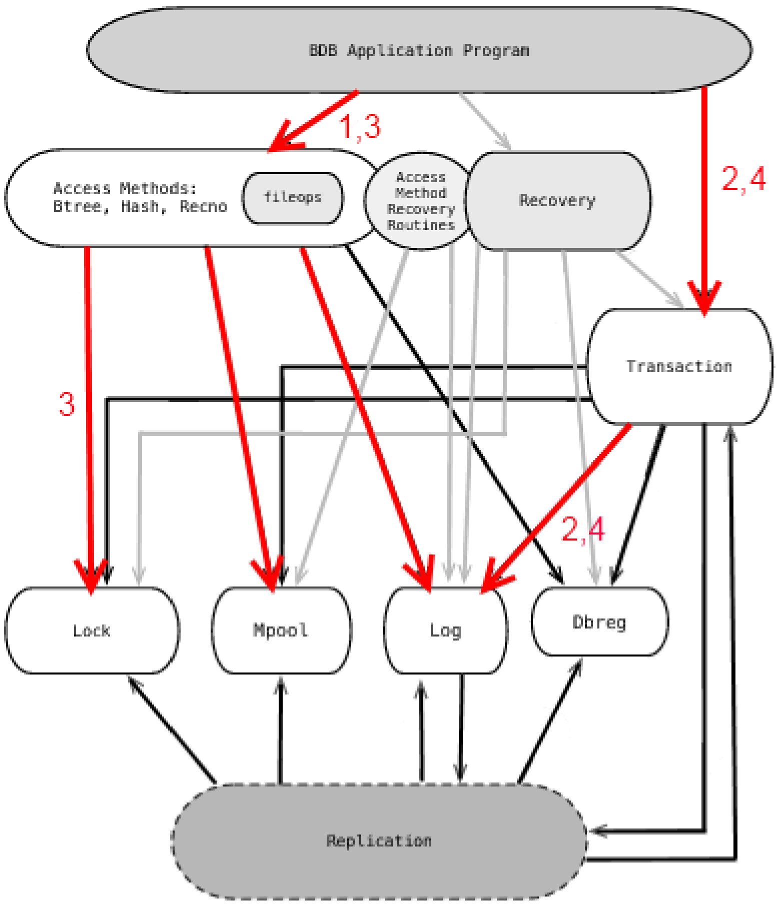

Figure 2: Writing to a database

The first step that the calling application takes is opening the database (2.1). This calls the open API call on the Access Methods component. After this the application creates a new transaction (2.2) by calling the Transaction module, at the same time the transaction module writes to the Log file that it is starting a new transaction. This then returns control to the calling application.
The calling application would then put a new (key,value) in the database (2.3). This is done by calling the Access Methods module. This module then calls to the Lock module to get a lock on the page of data where it is writing the data to. Then it calls the Log component (this is done before writing the data because Berkeley DB is an ACID database which uses Write-ahead logging to avoid loss of data[1]). Lastly the access methods component will write the (key,value) pair to the database by using the Mpool component. The Lock is then released and control is returned to the calling application.

The application can then close the transaction (2.4). This will write to the Log that the transaction is closed.

[1]Gray, J., Reuter, A., “Transaction Processing: concepts and techniques”. 1993.

Data structures or algorithms:

A unique data structure used is the Memory Pool. This is represented in the architecture diagram by the Mpool component. This component was created to hide the fact that main memory is a limited resource, and it therefore moves pages in and out of memory when the database is larger than memory. In this way, a piece of memory is only loaded when needed.

A consequence of this is that instead of using pointers, the data structures used to implement the database (such as B+ Trees) must use page numbers and offsets instead of pointers to access data in memory (I also talk about this in the Concurrency section of the survey).

The other data structures and algorithms used throughout Berkeley DB are no different than those used in most other database systems.

Control flow and/or data key to the architecture if any:

The control flow in the architecture is very strictly layered. The Transaction component is the only component that violates the layering. What is strange is that layers are tightly coupled, but there is low cohesion within the layers. I think this is due to the fact that C does not actually have a way to define interfaces. This may also just be an artifact of Berkeley DB's history, before the idea of low coupling was popular. And it may be too difficult now to refactor the design to use a facade between the layers.

Another very possible option is that each component can just be drawn as part of a layered architecture and that there no formal notion of a layered architecture in the system.

Architectural style:

The architecture of Berkeley DB is closest to a layered architecture. However, as you can see in the architecture diagram above the layering is not completely well defined. The transaction component sits outside of the layers and interacts with them without regard for the layering of the system. The rest of the components follow this layered architecture.

Each component is accessed via an API. The authors did this on purpose to enforce the layering and maintain generalization. Another benefit of this structuring through APIs is that an external application could choose to just use a certain component of Berkeley DB in it's functionality. For instance, an application could choose to only use the Locking component to handle locks, and avoid re-inventing the wheel when they requiring locking of objects in their application.

Major evolutionary changes:

The architecture has not had any major changes since version 2.0. The layered architecture (including the external Transaction component) have remained the same. The interactions between existing components has become more complicated as new functionality has been added. In addition, the Replication layer has been added to implement the shared storage that was discussed in the Use of Concurrency section.

Performance bottlenecks:
As with all databases, disk I/O is the major bottleneck. By using the in-memory caching abilities of Berkeley DB this can be partially avoided, but not entirely. The best way to avoid this bottleneck is to attempt to use batch processing as much as possible. For instance, if you know that you will need to write a lot of data to the database at some point, wait until a period of low activity and write the data in one transaction.

Real time: None

Notation for architecture: The original architecture for Berkeley DB was described using a box and arrow diagram. This has not changed, the current architecture in the book chapter was also described in this way.

Methodology: Not clear. Originally the development methodology appears to have been iterative. However I am unaware of the development methodology used at Oracle.

Appendix:
Kruchten's eight context attributes applied to Brown/Wilson systems

1. **Size:** M
2. **Criticality:** Hi
3. **Age of system:** XL
4. **Rate of change:** Lo
5. **Business model:** Open-Source
6. **Stable architecture:** Lo
7. **Team distribution:** Lo
8. **Governance:** UnK