UNIVERSITÉ LIBRE DE BRUXELLES, UNIVERSITÉ D'EUROPE

Embedded Databases with Berkeley DB

Nazrin Najafzade
Ainhoa Zapirain Mariezcurrena

Embedded Databases with Berkeley DB

Nazrin Najafzade, Ainhoa Zapirain Mariezcurrena

December 14, 2018

Abstract

The objective of this report is to demonstrate our understanding of embedded databases with Berkeley DB. This report has two main subjects, embedded databases and the tool Berkeley DB. So, first we will briefly explain what an embedded database is, their use nowadays and how they work. After, we will deeply define the Berkeley DB and make a tutorial to show how to create a database and how to use the operations.

1 Introduction

To introduce the main objective of this report, first we need to talk about data management. The origin of this term starts with the evolution of data. With the progress of technology, data has been gaining more and more importance.

Nowadays, data management is integrated in the organizations, due to the fact that data is considered an inseparable part of a business. The objective of it, is to create an administrative process for the collected data. At first this data is considered raw data. So, to add some value, data mining processes are applied. As a result, data starts being useful for the business. However, this is the last step of the whole process of data management. This report will be focused in the step where the data is stored and managed.

To achieve that, a database system and a model are needed. In this case, we will use an embedded database with the Berkeley DB tool.

2 Embedded Databases

An embedded database is a database system which is integrated inside of a software application. In other words, embedded databases use inter-process communication, and not networking protocol such as TCP/IP sockets, to connect to a database management system.

2.1 Definition

There are two definitions of embedded databases [4]: simply embedded (Figure 1) and deeply embedded (Figure 2). Simply embedded, is the one mentioned above, the one which is integrated inside the application. However, a deeply embedded database is a simply embedded database that is inside an embedded sytem.

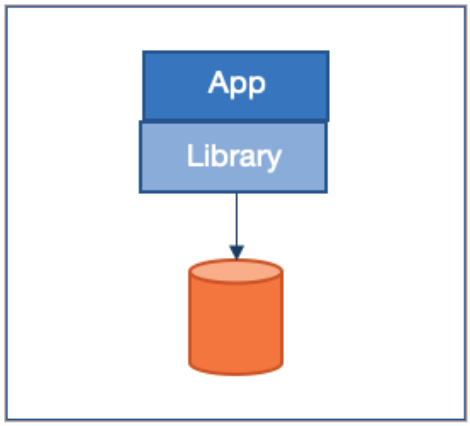

Figure 1: Schema of Simply Embedded database

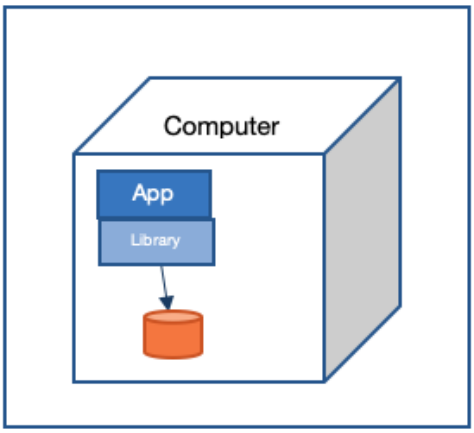

Figure 2: Schema of Deeply Embedded database

So, an embedded database as we can observe, is a database technology within an application. In other words, there is a library attached to the application which has all the operations needed for managing the embedded database.

2.2 Embedded Databases Nowadays

In the 80s and 90s, embedded databases were used primarily for line-of-business applications, for the internal unit of a business. Since the late 90s and early 2000s, embedded systems began to migrate to 32-bit architectures. Hence, embedded databases started to be considered commercially viable. In 2001, *eXtremmeDB* was launched as the first in-memory embedded database system.

Furthermore, *Raima* is a multinational company that has been present since the beginning. Nowadays, they offer a database manager for embedded databases among other tools. They offer two modules in their product: a library for the application and a transactional file server that manages the database.

Usually, this kind of database run in embedded environments, such as, mobile devices, tables, raspberry Pi, etc. This is due to the fact that this type of database is light weighted and easy to use. For instance, we can find embedded databases in statistics of games, industrial tools related to high tech, healthcare, retail, etc.

2.3 Comparison with Client/Server Database Systems

- **Installation.** In terms of installation embedded databases are easier to install than client/server database systems. The end user does not get involved with any external database, straightforward starts working with the database. Whereas in client/server database the end user needs to connect with an external server to work with the database.

- **Structure.** Embedded databases and client/server database systems have different structures, as reflected in the Figure 1 and Figure 3. Client/Server databases have an application connected to a server where the database is, whereas in embedded databases the database is integrated in the application through a library. So, in embedded databases functionality becomes part of the application itself. As a consequence, the transmission of the data is faster.

- **Types of database.** Subsequently, we encounter that there exist the same type of database in both systems. Embedded databases can be either an in-memory database or persistent database. Additionally, the same happens with client/server database systems. As a result, both can implement in-memory and persistent databases.

- **Data transmission.** Considering that embedded databases are integrated in the application, the data is not transferred across network, it remains inside the environment. Therefore, the transmission of data is straightforward. However, in client/server database systems, the data transfers across the network. This means that the application will have to wait for a response before transmitting data. Indeed, embedded databases are controlled by the application in which is integrated.

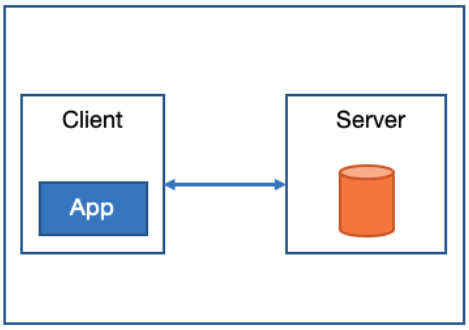

Figure 3: Schema of a client/server database system

3 Berkeley DB

Berkeley DB is a library for embedded databases, which is available for multiple programming languages such as C, C++, Java, Perl, Tcl, Python, and PHP [2]. It can run in UNIX, LINUX, Windows and a number of embedded real-time operating systems, running in 32-bit and 64-bit systems. This library its linked into an application.

3.1 Definition [2]

Before dipping into the explanation of Berkeley DB, we should know what Berkeley DB is not.

Firstly, Berkeley DB is not a relational database. There are no SQL queries, all data is accessed through the Berkeley DB API. Additionally, Berkeley DB is not aware of the schema of the database whereas relational databases are.

Secondly, Berkeley DB is not an object-oriented database. It is written in C and can be used with Java, C++ and other programming languages mentioned above. Furthermore, Berkeley DB does not know which methods are defined in the objects of the user. This is the reason why the programmer of the applications needs to tell Berkeley DB which record retrieve.

Thirdly, Berkeley DB is not a network database. As we know, in a network database the main model is a hierarchical model, where every record has one or more parent records and many child records. In other words, relationships between records are in a network-style and records of tile are linked to one another. However, in Berkeley DB records are referred only by key, not by addresses.

Lastly, Berkeley DB is not a database server. As mentioned above, Berkeley DB is a library, so it is stored within the same direction as the application. However, Berkeley DB can be integrated in a server without disturbing the other databases.

3.2 Operating Mode [2]

The operating mode of Berkeley DB is the following, an application will be the one calling to the operations that offers this library. The CRUD operations are only available for the end user. The application will need some method to manage the data to convey with these operations. There are two ways of executing these operations: in multiple processes or multiple threads in a single process. The library is responsible of the low-level services, such as locking, transaction logging, shared buffer management, memory management and so on. The end user is not aware of these services. As a consequence, the end user can use the application without knowing that a database exists.

The database is able to grow and manage up to 256 TB in size. In other words, its scalable, it can handle the growth of data and users. Depending on the machine that is working on, Berkeley DB can take advantage of GBs of memory and TBs of disk.

3.3 Architecture

Berkeley DB uses five major subsystems: cache, data store, locking, logging and recovering. Using these, Berkeley DB provides ACID properties.

- **Cache.** All Berkeley DB databases are just a set of pages. For instance, consider having the following pages: Page 0, page 1, page 2, , Page 1000, , up to TBs of data. (Figure 4)

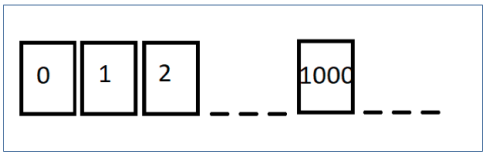

Figure 4: All the pages have the same size and they never change once the database is created

Now, consider a database called "Capitals" and inside of it there is a record. In this record, the key is "Brussels" and the value is "Is the capital of Belgium". Suppose that this record is stored on the 585th page and its offset is 134. To access this record the procedure will be the following: Open the file, seek to search 1000 pages size, read each page into the cache and then offset at 134 and read the key in the data.

To make it more efficient hash tables can be used, where each page is hashed by its page number, from 0 to 1000. Once created the hash table, we just need to find the correct bucket for the page.

- **Locking.** A locking system is a page-level read/write lock. Considering the previous example, now instead of reading a record we are going to insert it in the database. For that, we need a read lock from the locking system. If page 0 has not write locks on it, then the read lock is granted and we continue. If there is a write lock, the request of the read lock is put in a queue and the thread of control is forced to block until there is no write locks.

  Berkeley DB has its own deadlock system. Whenever certain locks are requested, Berkeley DB will go through all the current locks and lock requests. With these locks, it will build a matrix to determine which is the deadlock (Figure 5), a lock will be a deadlock when it cannot be obtained within a time. If there is a deadlock, the transactions that are involved on it will be sacrificed. After, all locks will be released and the transactions will be forced to rollback. As a result, all the transactions in the deadlock can now move forward. This locking system performs in the record level.

	Holder								
Requester	No-Lock	Read	Write	Wait	iWrite	iRead	iRW	uRead	wasWrite
No-Lock									
Read			✓		✓		✓		✓
Write		✓	✓	✓	✓	✓	✓	✓	✓
Wait									
iWrite		✓	✓					✓	✓
iRead			✓						✓
iRW		✓	✓					✓	✓
uRead			✓		✓		✓		
iwasWrite		✓	✓		✓	✓	✓		✓

Conflict Matrix

Figure 5: Conflict matrix [5]

  However, you can create your own lock modes and a conflict matrix to suit the purposes of your application. The only requirement is that the number of columns and rows must be the same.

- **Data Store.** Berkeley DB offers the following different methods for organizing the data.

  - ☐ **Hash Tables.** Supports key/value pairs and only page-level locking. It goes well with large scale data and predictable search and update.
  - ☐ **BTree.** Supports key/value pairs and only page-level locking. Allows to retrieve data with keys between a start and end value. This is not possible in hash tables.
  - ☐ **Record-Number-based Storage.** Supports variable-length values. The key of the database will have the value of the number of the record.
  - ☐ **Queues.** Supports fixed-length values and record-level locking. Deals with the data in the order that they are inserted.

- **Logging.** Many databases use a journal file until the time when the transaction is resolved. In other words, when this journal file, logs and database pages are flushed to disk. Once everything is resolved, the journal is deleted.

  Berkeley DB writes all logs into a single large journal. There is not such concept of one file for each database, all the transaction logs and the records are stored in these logs. It uses an ever-increasing number of fixed length log files (Figure 6). By default, when a log file reach to 10MB of capacity, Berkeley DB will create another log file. These log files only store the appends, the inserts, deletes and updates of the records are added in the end of the current file that we are in.

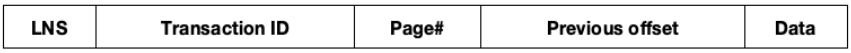

Figure 6: Structure of the log files

  - **LNS.** The log sequence number. Consists on the number of the log file and the offset into the log file, where this log entry will be stored.
  - **Transaction ID.** The unique ID that the transaction uses to identify. This is used for the case that we fall back, at least we know what logs are from which transaction.
  - **Page Number.** All operations are logged and performed at the page level. If a record is on multiple pages, each insert is broken up into a different operation and each operation gets its own log.
  - **Previous Offset.** The offset of the previous record.
  - **Data.** The data.

  However, sometimes happens that there are logs no longer in use. So, these files are occupying space, for the sake of recovering space JE provides a cleaner. Thus, this cleaner reads these files that are no longer in use, and recovers the records which are still in use and writes in the in end of the current log file. Finally, deletes the file that it has inspect.

- **Recovery System.** In Berkeley DB, the checkpoints work as follows. First, all pages are flushed to disk and a checkpoint log entry is created, where the LSN of that log is guaranteed to be larger than every single log entry of a committed transaction. Then the checkpoint log is inserted in the buffer and the buffer is flushed. If a crash happens, there is no need to read all of the logs from the beginning, it is enough to go to the last checkpoint. This guarantees that everything beyond that is on the disk.

For example, consider that we have inserted the previous example and committed the transaction, but we have not done a checkpoint yet. Then the application crashes. Berkeley DB recovers as follows: When you open the database after the crash, before performing any other operations, the database looks into the log files and finds the last log. Then it reads and finds the page number, in which was last operated, and reads that page into memory. After, it checks the LSN on that page. If the number matches or is greater than the last log, then it is the page that was flushed before the crash. As a result, proceeds to undo the operations. If this log number is less, mean that it was never flushed after being altered. So, the log is just ignored.

However, there is an option of creating a backup of the database. Consist on copying all the log files from the lowest numbered log file until the highest numbered log file. As a result, the database can be restored from this backup. Whenever there is a need to recover something, it is enough by copying the log files from the backup file to the environment in which the database is.

3.4 Tutorial [3]

For the tutorial, we will be implementing a simple bookshop manager. We will be storing the next data.

Customer	Book	Books Sold
ID: Integer Name: String Age: Integer Gender: String	ISBN: String Title: String Author: String	CustID: Integer BookNumber: Integer Date: String

Table 1: Tables of the database

But first, we need to define some terms of the Berkeley DB. Considering we are more familiar with RBDMS, we will find the equivalent of these terms in RBDMS. For that we have done the next table.

Berkeley DB	SQL Server
Environment	Database
Secondary Database	Secondary Index
Database	Table
Key/Data pair	Tuple/row
Key	Primary Index

Table 2: Equivalent terms

So, we know that Berkeley DB offers two different APIs: DPL and Base API. Therefore, we will explain how to implement them. However, both of them need an environment where we store our databases. Thus, we will explain how to create an environment first and after we will deep in with the implementation of the two APIs.

- **Create environment.**This environment will create multiple threads inside the in-memory cache for each database opened. These threads will be opened once per process.

```
private Environment env;
...
// Configuration of the environment
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setTransactional(true);
envConfig.setAllowCreate(true);
// Creation of the environment
this.env = new Environment(new File(homeDirectory), envConfig);
```

- **Close environment.**When an environment is closed all the resources are cleared and all the threads are stopped. However, this must be done when there is no other handler referencing to this environment. If there exist any handler, we need to close them first.

```
this.env.close();
```

- **Implementation of Persistence Layer.** Suited for applications that want to store and manage Java class objects.

  1. **Open Entity Stores.** Once we have the environment, we need to create the storage for the persistent entity objects.

```
private EntityStore store;
...
public void setup() throws DatabaseException {
     ...  Open the environment ...
     // Configuration of the entity store
     StoreConfig storeConfig = new StoredConfig();
     storeConfig.setAllowCreate(true);
```

```
        this.store = new EntityStore(this.envmt, "EntityStore", storedConfig);
    }
```

2. **Persistent Entity Objects.** Storage for the data. We will use annotations. Create a persistent entity object for each table that you want to store in your database.

```
    // Declares a class with a primary index and multiple indices
    @Entity
    public class Book {
        // Primary key of the entity.  Only used once per entity.
        @PrimaryKey
        private String ISBN;

        // Secondary key of the entity.  Can be used multiple times per
    entity
        @SecondaryKey
        private String title;
        @SecondaryKey
        private String author;

        ...  Create GET/SET methods for each attribute ...
    }
```

3. **Insert and Retrieve data.** For accessing and retrieving data, we will use indexes. So, first of all we create a data accessor class, where we declare the indexes that we are going to use to retrieve and insert data.

```
    public class BookDA {
        // Index accessors
        PrimaryIndex<String, Book> pIdx;
        SecondaryIndex<String, Book> sIdx1;
        SecondaryIndex<String, Book> sIdx2;
        // Open de indexes
        public DataAccessor(EntityStore store) throws DatabaseException
    {
            // Primary key
            this.pIdx = store.getPrimaryIndex(String.class, Book.class);
```

```
        // Secondary key
        this.sIdx1 = store.getSecondaryIndex(pIdx, String.class, "title");
        this.sIdx2 = store.getSecondaryIndex(pIdx, String.class, "author");
    }
}
```

Once we have the data accessor class, we can create the insert and retrieve classes.

☐ **Insert data.**

```
// Open the environment and the entity store
// Open the data accessor -> to store persistent objects
// The variable store is the EntityStore
this.da = new DataAccessor(this.store);



// Instantiate as much entities as you want to insert
Book book1 = new Book();
Book book2 = new Book();

...  Other instances ...

// Assign and insert values
book1.setISBN("Book 1");
book1.setTitle("The Three Musketeers");
book1.setAuthor("Alexandre Dumas");
da.pIdx.put(book1);

...  Now with the other instances ...

// Close the environment and the entity store
```

☐ **Retrieve data.**

```
// Open the environment and the entity store
// Open the data accessor -> to store persistent objects
// The variable store is the EntityStore
this.da = new DataAccessor(this.store);
```

```
// Instantiate as much entities that you want to retrieve

Book book1 = this.da.pIdx.get("Book 1");
...  Other instances ...

// Showing the values from the scree
System.out.println("Book 1:  "+ book1.getISBN());
System.out.println("Book 1:  "+ book1.getTitle());
System.out.println("Book 1:  "+ book1.getAuthor());
```

– **Queries.** For this example, imagine we want the ISBN of a specific book. So, we will construct the query with the help of the indices.

```
// Create indices
PrimaryIndex<String, Book> ISBNPidx = this.store.getPrimaryIndex(
                                          String.class, Book.class);
SecondaryIndex<String, String, Book> titleSidx =
    this.store.getSecondaryIndex(ISBNPidx, String.class, "title");
SecondaryIndex<String, String, Book> authorSidx =
    this.store.getSecondaryIndex(ISBNPidx, String.class, "author");

// Creates an equality between two or more secondary keys
EntityJoin <String, Book> join = new EntityJoin(ISBNPidx);
join.addCondition(titleSidx, "The Three Musketeers");
join.addCondition(authorSidx, "Alexandre Dumas");
ForwardCursor<Book> joinCursor = join.entities();
Iterator<Book> i = joinCursor.iterator();

try {
    for (Book book:  joinCursor) {
        System.out.println(book.getISBN());
    }
} finally {
    joinCursor.close();
}
```

4. **Update data.** First retrieve the data and after update with the new value.

```
// The variable da is the instance of the DataAccessor class
Book book1 = this.da.pIdx.get("Book 1");
book1.setAuthor("Alexandre Dumas, Gatien de Courtilz de Sandras")
this.da.pIdx.put(book1);
```

5. **Delete data.** The easiest way to delete data from an object is to delete its primary index.

```
this.da.pIdx.delete("Book 1");
```

6. **Close Entity Storage.** Closes the EntityStorage instance.

```
this.store.close();
```

- **Implementation with Base API.** Suited for applications from Berkeley DB API. Allows you to configure more the database, is in a lower level than the persistence layer.

1. **Opening Databases.** Once we have the environment, we create a database for each table.

```
private Database bookDB = null;
...
// Configuration of the databases
DatabaseConfig dbConfig = new DatabaseConfig();
dbConfig.setTransactional(true);
dbConfig.setAllowCreate(true);
this.bookDB = this.env.openDatabase(null, "book", dbConfig);
```

As we can see here we have created a transactional database, this means that whenever the transaction is committed these changes will become durable. There is another mode of handling the writes, deferred write databases (`setDeferredWrite()`). This type will write when it receives a sync signal (`DatabaseConfig.sync()`), this will make the written data durable.

The database that we are creating here by default is durable, but there is an option of making it temporal.

- ☐ **Temporary.** Useful for an application that wants temporal databases. They work best when there is a large in-memory cache. Uses the deferred write mode.

  ```
  DatabaseConfig.setTemporary(true);
  ```

2. **Create the databases.** We create a class for each table that we want to create.

```
public class Book implements Serializable {
     private String isbn;
     private String title;
     private String author;

     // SET Methods
     public void setISBN(String data) {
          this.isbn = data;
     }
     public void setTitle(String data) {
          this.title = data;
     }
     public void setAuthor(String data) {
          this.author = data;
     }

     // GET Methods
     public String getISBN(String data) {
          return this.isbn;
     }
     public String getTitle(String data) {
          return this.title;
     }
     public String getAuthor(String data) {
          return this.author;
     }
}
```

3. **Insert and Retrieve Data.** For accessing the data we will get the help of `DatabaseEntry` object. This object encapsulates the key and the data of the database.

   - **Insert Data.** Because our database is serialized, it need to be binded to store the objects with serialized object format. So, after setting the data we will bind it and insert it into the database.

```
// Key of the database
String key = "theBooks";
Book bookToInsert = new Book();

bookToInsert.setISBN("Book 1");
bookToInsert.setTitle("The Three Musketeers");
bookToInsert.setAuthor("Alexandre Dumas");


// Open database for your class information
Database classDB = this.env.openDatabase(null, "classDB", dbConfig);

// Instance of class catalog
StoredClassCatalog classCatalog = new StoredClassCatalog(classDB);
// Bind the database
EntryBinding dataBinding = new
     SerialBinding(this.env.getClassCatalog(), Book.class);

// Create the database entries
DatabaseEntry theKey = new DatabaseEntry(key.getBytes("UTF-8"));
DatabaseEntry theData = new DatabaseEntry();
dataBinding.objectToEntry(bookToInsert, theData);

// Insert data
this.bookDB.put(null, theKey, theData);
```

   - **Retrieve Data.**

```
// Key of the database
String key = "theBooks";

// Open database for your class information
Database classDB = this.env.openDatabase(null, "classDB", dbConfig);

// Instance of class catalog
```

```
StoredClassCatalog classCatalog = new StoredClassCatalog(classDB);

// Bind the database
EntryBinding dataBinding = new
     SerialBinding(this.env.getClassCatalog(), Book.class);

// Create the database entries
DatabaseEntry theKey = new DatabaseEntry(key.getBytes("UTF-8"));
DatabaseEntry theData = new DatabaseEntry();

// Retrieve data
this.bookDB.get(null, theKey, theData, LockMode.DEFAULT);
Book retrieveBook = (Book) dataBinding.entryToObjects(theData);
```

4. **Close Database.** Closes the databases. But before closing the database, we need to make sure that there are no cursors using the database that we want to close. If there exist and we have not handle it JE will warn us and close them.

   ```
   bookDB.close();
   ```

   The implementation described above, is a vague idea of an actual database. Actually there are plenty of features that we have not mentioned. We wanted to make an introduction to this API. But if you want to learn about this API check *Oracle Berkeley DB Java Edition* [3].

Now the big question is which implementation to use. This decision depends on what is the objective of your application. If you have a database schema that is static the best option is to use the persistence layer, but if the schema is dynamic the best approach is the Base API, even though you can use the persistence layer.

4 Use cases

Embedded databases have two evident advantages: fast access to the stored data and the low latency as a result of being in the same machine as the application. However, where can we use these databases?

- **Storing sessions.** We can use the embedded database to store user sessions. The key would be an identifier for one user, for example, a random generated string. The data can be anything that we require. This is especially useful for web applications that receive millions or billions of requests daily, and accessing a database for user sessions is not feasible.
- **White-listing.** Consists in a list that contains some identifiers that are have some privileges. For instance, we can have a MAC address white-list to control who is allowed to enter in the network.
- **Inside a server.** An embedded database inside a server. What we can do is when a user makes a lot of call to the same table, we can store this data in a embedded database inside the server, like that the reading of the data is faster.

Thus, the best fit for embedded databases are in-memory databases. Regarding Berkeley DB as a chosen embedded database tool, we will define some specific use cases [1].

- **Storage for identities.** As mentioned above, there are some web-services that need fast login. They can store username, password and the behaviour that had in the web page. In this way we avoid all the queries for each operation that need to be made to verify the user.
- **Storage for messages and their metadata.** Messaging systems need to track the senders and recipients messages all along the process. Track the length, the creation time, the subject and other metadata of each message. Therefore, Berkeley DB it will be hosted in the email server saving all this data.
- **Storage for switching and routing.** These systems are connected between them. To transmit information between them, they need to know where is their location and where is the destination. This type of transactions need to be fast, for this reason these systems rely on Berkeley DB.
- **Storage of the sharing information between companies and their partners.** Companies need to transfer information between their partners. This transmission needs to be exported from the financial system of the company and after sent to the partner. Berkeley DB will store, as a cache, the logs of this information, for auditing and compliance purposes. This database will be set on the edge of the network, to capture the traffic.

This are some of the uses of Berkeley DB. But most of the time it has the same task, to work as a cache.

5 Conclusions

5.1 Berkeley DB vs RDBMS

During the process of this report, we have seen the flaws and the strong points of Berkeley DB. However, in some cases we have seen that is better to use a relational system instead of a embedded system.

- **Language.** Berkeley DB is a library, in our case is written in Java. Usually these type of databases are deployed by software developers. So, when you are using Berkeley DB you need to know how to program, in our case how to program in Java. Therefore, it implies you to know at least the basics of Java. Therefore, we think that relational systems are more intuitive because of SQL.

- **Queries.** Berkeley DB works better with static queries, because the developers need to predict what the applications will want. In comparison, relational systems are good with dynamic queries, being best at responding to new questions.

- **Model.** As mentioned above Berkeley DB stores records that consist in a key and data. So, this library is able to search for these keys in the database, without knowing the inner structure of this keys and data. Differently, relational systems know how is the relational model, so it formats the record in a way that it is useful.

- **Servers.** Berkeley DB does not need any server to work, it is a library embedded to the application that uses the database. Contrarily, relational systems need a server to host the database, with which the application is connected.

In this table we can observe the comparison between an specific tool of RBDMS.

Name	**Microsoft SQL Server**	**Oracle Berkeley DB**
Description	Microsoft's relational DBMS	Widely used in-process key-value store
Primary database model	Relational DBMS	Key-value store Native XML DBMS
License	commercial	Open Source
Implementation language	C++	C, Java, C++ (depending on the Berkeley DB edition)
Server operating systems	Linux Windows	AIX, Android, FreeBSD, iOS, Linux OS X, Solaris, VxWorks, Windows
Data scheme	Yes	schema-free
Typing	Yes	No
APIs and other access methods	OLE DB, Tabular Data Stream (TDS), ADO.NET, JDBC, ODBC	
Supported programming languages	C#, C++, Delphi Go, Java, JavaScript (Node.js), PHP, Python R, Ruby,Visual Basic	.Net , C, C#, C++, Java, Perl, Python,Tcl
Server-side scripts	Transact SQL, .NET languages, R, Python and (with SQL Server 2019) Java	No
Foreign keys	yes	No
Transaction concepts	ACID	ACID

Table 3: Table of the comparison

5.2 Our experience

To conclude with the report we need to add that this technology may not be that useful these days. However we think that it is an interesting implementation. From our experience, it is really easy to deploy a simple application, and that there are plenty of ways of implementing it. Additionally, while creating the application we have seen that making queries is really specific, meaning that the developer needs to know what will ask the application to retrieve the data.

References

[1] Oracle. *A Comparison of Oracle Berkeley DB and Relational Database Management Systems*. November 2006.

[2] Oracle. *Berkeley DB Programmer's Reference Guide*. October 2010.

[3] Oracle. *Oracle Berkeley DB Java Edition*. October 2017.

[4] RaimaDB. *What's an Embedded Database*. Youtube, 2013.

[5] Margo Seltzer and Keith Bostic. *Berkeley DB*.