Department of Computer Science and Engineering

Course: Database Systems Lab (CS254)

IceDB

An In-memory Key-Value Store

Team:

1. Sai Krishna Anand (181CO244)

2. Suhas K S (181CO253)

INDEX

1. Introduction
2. A Brief Study of NoSQL database systems
3. Existing implementations and Inspirations
4. Design:
    a. Preliminaries
    b. Architecture
    c. API
    d. Directory Structure
    e. Data Structures
5. Implementation:
    a. The Engine
    b. The Shell
6. Results
7. Conclusion
8. References

1. Introduction:

1.1. Abstract:

Relational Data Models and SQL Database Management Systems are tried and tested techniques to store and maintain data efficiently. Today's SQL Database Systems are the outcomes of decades of research put into refining every functionality of Relational Data Models.

At the core of a Relational Data model is the concept of a **table**, a rigid conceptual visualization of data storage. The **rigidity** of this concept provides a lot of advantages. These include structural **consistency**, versatility in data access, sophisticated ways to maintain data and overall an easier to use system due to the use of SQL. But this kind of rigidity brings its own set of problems, primarily **performance overhead** with respect to data access, problems with **horizontal scalability**, over-reliance on high end computing and networking hardware and over-reliance on **structured data**. The first and last points especially are huge problems for a wide variety of web applications and technologies in the present day. A lot of web applications rely heavily on semi-structured data and SQL databases are not the best option for storing such data.

Due to such limitations in the conventional Relational Model, the **NoSQL** concept of database management was introduced in 1998. Here the focus was more on speed than validity, on semi-structured data than perfectly structured data, on dynamic schema rather than a preset, structured schema. These database systems quickly rose in popularity, especially applications like Redis and MongoDB.

This project aims to study such database systems in detail, including their components and construction. It also aims to implement a **Key-Value store** NoSQL database system **IceDB,** using hash tables in C++.

1.2. Aims of the project:

1. A minimalist, functional in-memory Key-Value store database engine based on easy-to-use associative data structures. In the case of IceDB, we have based the engine on **Hash Tables**.
2. A command line interface for direct and instantaneous control over the databases stored on a machine.

2. A Brief Study of NoSQL database systems:

NoSQL database systems, by definition, are based on data storage and retrieval mechanisms antithetical to relational database systems, in the sense that they are non - relational. A database system can be called a 'relational database system' if:

- It presents the data to the user as **relations**, a tabular form of data representation.
- The sequence of columns and rows are not of importance.
- Relationships between entities are **complex**, and depend on the relations participating in the relationship.
- They are a very simple form of storing data - structured but not ordered.

On the other hand, NoSQL databases store data in a key - value format. Few properties of NoSQL databases are:

- It presents the data to the user as **collections,** which is a group of objects stored together. Each object may or may not have the same keys, i.e, the schema is **flexible**.
- NoSQL databases are scalable **horizontally**, which means that they can handle increased traffic simply by adding more servers to the database, while SQL systems are scalable only **vertically**.
- They do not have any complex relationships as the data is **unstructured**.
- Their access times are much lesser and hence are much more **efficient** compared to their SQL counterparts.

NoSQL databases have become very popular in the last 10 - 15 years, with the increase in demand and use of database systems. While the primary data structure of an SQL database is a table, NoSQL databases can be implemented using a wide variety of data structures like **hash - tables, documents, graphs**, etc.

The advent of big data and artificial intelligence in the 21st century has led to many companies using NoSQL databases, as data processing is simple, quick and efficient. For example, social media companies like **Instagram** and **Facebook** have millions of users everyday, and store an immense amount of data. They use NoSQL database systems as fetching data for multiple users can be achieved much faster. Further, social media companies store a lot of data about every single user and in most cases, there will be optional data fields to be filled out. If a relational database system were to maintain this system, there tends to be a lot of **NULL** values in the database, wasting a lot of space. Since it needs to store millions of users' information, wastage of space is a very big problem. NoSQL databases solve this problem by using a **key - value pair** for the users. Only the data filled out will go into the key - value stores and hence a lot of space is saved.

Many NoSQL databases compromise consistency in favour of **data availability, partition tolerance** and **performance**. Most NoSQL databases also lack true ACID transactions, although a few of them have made them central to their design. Instead, they provide a concept of "**eventual consistency**", in which a database's changes are propagated to all nodes eventually(in milliseconds), so queries for data might not return updated data immediately or might result in reading inconsistent data, a problem known as **stale reads**. Some other problems of NoSQL databases is that they provide the use of **low-level query languages**(instead of something like SQL, which can easily provide joins), **lack of standardized interfaces**, and huge previous investments in existing relational databases.

However, all these factors do not make one type of database system better than the other. Choosing between NoSQL and RDBMS depends on various factors, such as the **application data model,** the **consumer community**, and **performance requirements**. Many applications require a structured form of data, so as to create **custom reports** and **data relationships**. Using RDBMS, data retrieval can be made very specific and high level languages like SQL use abstraction to make the understanding of the database design very simple. On the other hand, NoSQL database systems are used very often in applications requiring **performance and fast synchronization** and such applications generally tend to store a large amount of data. Here, importance is not given on ease of querying, but on how fast the data can be brought.

Some examples of the companies using **RDBMS** are **Microsoft, Dell, Cognizant,** etc as these companies require and perform a lot of data analysis, for which SQL databases are ideal. Examples of companies using **NoSQL** database systems are **Amazon, Instagram, J P Morgan, Capgemini**, etc, as these companies need fast processing of data very often. Hence, selection of the database system should be done after carefully analysing the requirements of the application.

3. Existing Implementations and Inspirations :

IceDB is essentially a key - value store, written in C++. It uses a very basic implementation of an **in-memory hash-table**, with a very primitive hash function. The collisions in the hash table are handled using **chaining**, which implies that every bucket of the table is essentially a linked list, and all items with same hashed indexes are stored in that respective bucket.

Key - value stores are the simplest type of NoSQL databases, simply because of it’s direct mapping data structure. Examples of such NoSQL databases are Amazon DynamoDB, Redis, Riak, etc. Such database systems are mainly used for their speed - reading , writing or updating is very fast as long as you have the key. However, querying the entire database is slow. IceDB follows the same principle and stores the database in a flat-file system.

NoSQL database systems like **MongoDB** use a **document style data structure**, where each key has a value of data structure “document”. This document can contain key-value pairs, key-array pairs and also documents itself(nested documents). Such databases are very flexible and are **schemaless**, and hence can store any type of data needed in the document. They also have good query times based on indexing and hence are widely used in applications.

Other types of NoSQL databases are **Column stores(Cassandra)** and **Graph stores(OrientDB)**. Column stores are the fastest type of NoSQL DB’s and are used widely in big data analysis and applications where speed is a critical component.
Graph stores focus on relationships between the data keys, where each key value pair is a node and can handle specific queries very quickly.

IceDB is in its primitive stages of development and does not boast many of the main features of the above mentioned database systems. However here are some of its properties :

- It uses **Object Oriented Programming** to abstract storage details and access paths of the database, and is a **schemaless** database.
- Since C++ in general is a fast programming language, hash tables have very **high access speeds** as in our implementation.
- Very simple implementation of memory management and data storage.

4. Design :

4.1. Preliminaries :

IceDB is essentially a Linux application. It requires the following dependencies to be present for build purposes :

- autoconf (>= 2.69)
- automake (>= 1.15)
- Libtool (>= 2.4.6)
- pkg-config
- libreadline-dev (Development files for GNU readline)

More detailed build and run instructions can be found in section 6 **Results**.

4.2. Architecture :

Here we briefly describe the high-level architecture used as a road map for building the application. The main aspect to consider while designing such an application is modularity, in the logical design as well as the actual code. Hence, we will divide the system into self contained modules, each containing several sub-modules. The main systems to implement in the application is:

- **Interface**: An API. This also includes Parametrization, for handling options and arguments.
- **Data storage:** The interface used to access the memory where data is stored.
- **Data Structure**: As discussed before, hash tables are used here.
- **Iteration**: Implemented using C++ iterator objects. Can also be considered to be a part of Data Storage. Can be used to display the entire database.
- **Error Management**: Error handling mechanisms.
- **Memory Management**: For handling low level memory management operations like managing the flat - file system used here.

With these systems in mind, we can come up with a list of modules required for the implementation and their dependencies:

- **Core:** This is the module containing the interface. This will be the pivot for every other module, the central hub for the application. For the sake of compactness, it will also contain the Memory management system.
- **Data Storage**: This will handle the data storage system and the iteration system.
- **Error Management**: This is concerned with error handling.

Note : Mutexes, lock management and multithreading aren't implemented yet so the application can be safely run only locally and does not handle multiple users. This is the basic road map. In the future sections, each of these modules will be explained in detail, along with plans for implementation. Apart from this, a shell is built which utilises the API to enable full command-line control of the database.

4.3. API Design :

4.3.1. Core Class : The core class is IceDB. It would contain the methods that are part of the API. These methods and the functioning of the API in general is explained in this section.

4.3.2. API Components : There are totally 8 primary command-line functions used to interact with the database : (Note that all commands and data are case - sensitive)

➔ **`Open(<database_name>)` :** This is to open a database and map it to the calling object. The creates a database if it has not been created or opens it if it already exists. Internally, if a database already exists, it reads the data and the metadata file associated with that database and loads the data into a hashtable using `ReadDB` from the class `hashClass`.

➔ `Close(<database_name>)` **:** This is to close the database active in the calling object. Note that this will not drop or delete that database. This is the only way to save changes on the database permanently since all changes made to a database are **in-memory**. This function writes them to storage.

➔ `Get(<key>)` **:** This is the **read** operation for the database. It fetches the value stored corresponding to the passed key.

For example, suppose we have an existing database '**db**' with some data.
The following commands can be used to read data related to some key '**java**' :

```
Open("db");
std::map<std::string, std::string> *value = Get("java");
Close("db");
```

➔ `Set(<key>, "<key-value pair 1>, .... <key-value pair n>")`**:** This is the **write** operation for the database. It creates a new entry in the database with the main key and a number of key - value pairs, each separated by a comma.

If the main key is already present, it cannot be used again, due to which the user will receive a prompt to choose a new key.

The key-value pairs are written in the format - **<key:value> -** a colon separating the key and value.
Example :

```
Set("Sai Krishna", "Age: 20, Branch: CSE, College: NITK");
```

The main key is 'Sai Krishna' , followed by the key-value pairs - **<Age:20>, <Branch:CSE> and <College:NITK>**.

➔ `Delete(<key>)`**:** This function deletes the given key from the table. If the given key does not exist in the table, no action takes place. Note that all the data stored under that key is deleted.

Example :

```
Delete("Sai Krishna");
```

This command deletes all the data stored under the key **‘Sai Krishna’.**

➔ `Update(<key>, "<key-value pair 1>, .... <key-value pair n>")` **:** This is also a **write** operation for the database. It creates a new value with the given key - value pairs in the table if the key is already not present in the table, else changes the values of the fields given as the key-value pairs.
For example,

```
Set("Sai Krishna", "Age: 20, Branch: CSE, College: NITK");
...
Update("Sai Krishna", "Age : 19, Course: DBMS");
```

This sequence of functions changes the Age field of the key **‘Sai Krishna’,** creates a **‘Course’** field, and leaves the other fields unaffected.

➔ `PrintAll()`**:** This function **displays the entire database**. The display is done bucket - wise, to show the clarity of storage.

➔ `PrintKeyBucket(<key>)` **:** This function is similar to the previous one, but it does not print the entire database. It first checks if the given key is present in the table, and if yes, it prints it’s **entire bucket only.**
For example :

```
PrintKeyBucket("Sai Krishna");
```

Would print the data of all the elements present in the same bucket as the key **‘Sai Krishna’.**

This is the **basic API design**. The application can be developed with the focus of making it easier to add new options to allow for further development, beyond the course.

4.4. Directory Structure :

- **Preliminaries :** IceDB uses a flat-file storage mechanism, i.e, the entire database is stored in a single file. While this brings a set of disadvantages, it is the simplest way to store databases. Better storage techniques could be used in the future iterations of IceDB.
- **Directory Structure :** The working directory will be referred to as 'root' from here on. The absolute path of the root is stored in a global variable inside the master class. By default, the path to the root is `/var/lib/icedb/`.
  Whenever the **open** method is called, a new directory is opened with the name as the name of the database(ex: Database1) along with two files inside it:
  - ➔ **Database1.iced** : This file contains the data of the database, which is basically the hash table written from the main memory to the file.
  - ➔ **Database1.icem** : This file contains the metadata for the database.

- **Metadata :** The metadata contains the **name, last modified date and time, and number of key value pairs**. In the future iterations, with implementation of better data organization, database specific metadata could be used to a much larger extent.

4.5. Data Structures :

The data structure used for storage in IceDB is a hash table.

The hash-table takes in a value provided by the user (a string for example) and applies the **hash function** on it to get an index in the table. Every element of the table would be a **struct** containing the **key** entered by the user and all **other information** related to that value. It will also contain a pointer to the next element in the same index for the purpose of **chaining,** in case of collisions.

Chaining is the process of making a linked list at each hash-table element since the hash function may map multiple values to the same index. The idea is to have a hash function in which these linked lists are of minimal size, i.e, **minimal collisions**. The main functionality of our hash-table is the following :

- **Get/Search** : This function will take the value given by the user and transform it into an index using the hash function and look directly into the linked list at that index. This greatly **increases speed** as even with large data, only a small bucket of data has to be looked through, whose time complexity can be considered negligible.
- **Set** : This function is used to insert a new value in the hash table or update the data existing in the hash table.
- **Delete** : This function is used to delete the data related to a particular value in the hash table.

These 3 functions comprise the main functionality of the key value store. It is a very simple model as the main goal here is to increase its **efficiency and speed** rather than the ease of use.

Proper selection of a hash function is of high importance. For now, the hash function just calculates the sum of the ASCII values of the key. This sum modulus (size of the hash table) gives the index of the element. Further iterations of IceDB will consist of **better hash functions** so as to maximize efficiency.

5. The Implementation :

5.1. The Engine:

The main advantage of using C++ for the database engine was the use of Object Oriented Programming.

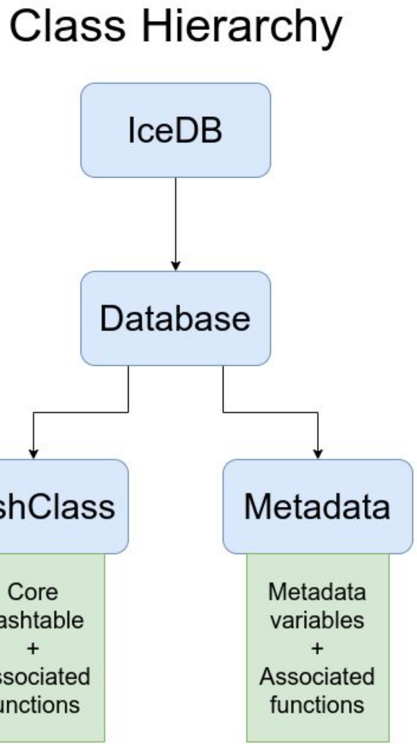
The main overlying class is the **IceDB** class, at the highest level, and the raw hash table data structure at the lowest level.

There are 4 main classes at play - the **IceDB** class, the **Database** class, the **Metadata** class, and the **hash** class. The Database class is the subclass of the IceDB class and each Database object has 2 subclasses - the metadata class and the hash class. This layered design ensures that the low level implementation of the hash table is **abstracted** from the user. The user can never directly interact with the hash table.

The **`IceDB`** class contains the main **API** functions which interact with the hash table and service the user's requests. The hash table is represented by an object of the class **`hashClass`**. The `IceDB` class uses an object of the **`Database`** class to **encapsulate** the **`hashClass`** and **`Metadata`** class, which has its own functions to read, write and display the metadata. At the lowest level, the hash table handles the read, write and delete functions as normal hash table functions.

All commands executed between an **open** and **close** database comprises a **session**. Everytime the **open** command is used, the `Open()` API function calls the `ReadDB()` function of `hashClass`. This function reads the database line by line and creates an in memory hash table. For large amounts of data, this step may take a little extra time. But the **fast servicing** of the requests made in the session overshadows this startup time. At the end of the session (**close** command), the `Close()` API function is called, which in turn calls the `WriteDB()` function of the hash class. It stores every element in a single line, encrypts it, and writes it into the file. This way, it writes the whole table into a single file.

This is the overall class structure and organization:


Class Hierarchy
IceDB
Database
hashClass
Metadata
Core hashtable + Associated functions
Metadata variables + Associated functions

The encryption used is the **Vigenere cipher,** which is a very basic cryptographic cipher, but used to show how the files can be made secure. Further versions of IceDB can include stronger encryption strategies such as **AES** or **RSA,** which are much more commonly used and current industry standard. We have used the Vigenere cipher for simplicity and because it's code is easy to implement. AES encryption and decryption functions require a large amount of cryptographic experience to code and hence, we leave it as part of future scope.

Essentially, the whole file is rewritten every time, something later versions of IceDB can improve.

5.2. The Shell:

The command line API of IceDB is handled by a shell, again written in C++. The `ice` shell is an interactive interface to IceDB. The shell can be used to query and update data by means of the commands discussed earlier. The build instructions are essentially to build and run the ice shell, after which all database commands are executed. The shell performs 3 important functions :

- Read the command and store it in a string buffer.
- Tokenize / parse the command to split it up into the actual command, main key, key - value pairs, etc.
- Process the command using the IceDB object assigned to that database and deliver the output back on the command line interface.

These are the list of commands in the shell:

➔ **`open <database_name>`** **:** This is to open a database and map it to the calling object. The database creates a database if it has not been created or opens it if it already exists. Note that if a database is open, we can only open another database after closing the currently opened one.

➔ **`close <database_name>`:** This is to close the database active in the calling object. Note that this will not drop or delete that database. The API keeps track of the opened database and hence, an error will be thrown if the database open is not the same as the one in the close command.

➔ **`get <key>`:** This is the **read** operation for the database. It fetches and displays the value stored corresponding to the passed key.

For example, suppose we have an existing database ‘**db**’ with some data.
The following commands can be used to read data related to some key ‘**java**’ :

```
open db
get java
close db
```

➔ `set <key>, <key-value pair 1>, .... <key-value pair n>` **:** This is the **write** operation for the database. It creates a new entry in the database with the main key and a number of key - value pairs, each separated by a comma.

If the main key is already present, it cannot be used again, due to which the user will receive a prompt to choose a new key.

The key-value pairs are written in the format - **<key:value> -** a colon separating the key and value.

Example :

```
set Sai Krishna, Age: 20, Branch: CSE, College: NITK
```

The main key is ‘Sai Krishna’ , followed by the key-value pairs - **<Age:20>, <Branch:CSE> and <College:NITK>**.

➔ `delete <key>`**:** This command deletes the given key from the table. If the given key does not exist in the table, no action takes place. Note that all the data stored under that key is deleted.

Example :

```
delete Sai Krishna
```

This command deletes all the data stored under the key **‘Sai Krishna’.**

➔ `update <key>, <key-value pair 1>, .... <key-value pair n>` **:** This is also a **write** operation for the database. It creates a new value with the given key - value pairs in the table if the key is already not present in the table, else changes the values of the fields given as the key-value pairs.

For example,

```
set Sai Krishna, Age: 20, Branch: CSE, College: NITK
...
update Sai Krishna, Age : 19, Course: DBMS
```

This sequence of commands changes the Age field of the key **‘Sai Krishna’,** creates a **‘Course’** field, and leaves the other fields unaffected.

➔ `display`**:** This command **displays the entire database**. The display is done bucket - wise, to show the clarity of storage.

➔ `display <key>` **:** This command is similar to the previous one, but it does not print the entire database. It first checks if the given key is present in the table, and if yes, it prints it’s **entire bucket only.**

For example :

```
display Sai Krishna
```

Would print the data of all the elements present in the same bucket as the key **‘Sai Krishna’.**

6. Results:

The final application is run and demonstrated through a video https://drive.google.com/drive/folders/1P_AUwbJxrp2YwhtpUkQwf16M2OzX9TW5.

To use the application:

1. Install the dependencies required:

```
$ sudo apt install autoconf
$ sudo apt install automake
$ sudo apt install libtool
$ sudo apt install pkg-config
$ sudo apt install libreadline-dev
```

2. Clone the github repository https://github.com/suhasks123/IceDB.git.
   a. If encryption support is not desired, clone the **master** branch of the repository as:

   ```
   git clone https://github.com/suhasks123/IceDB.git
   ```

   b. If encryption support is desired, clone the **encryption-support** branch of the repository:

   ```
   git clone -b encryption-support https://github.com/suhasks123/IceDB.git
   ```

3. In the root directory of the repository, run these commands to build IceDB:

```
$ autoreconf -i
$ ./configure
$ make
```

4. If the build is successful, there should be a file `libice.la` in the root of the repository, and two binary files `ice` and `ice_example` in `bin/` directory.

5. `libice.la` is the shared library containing the IceDB engine. The binary `bin/ice` is the IceDB shell that can be run directly.

6. Run the shell as `./ice` when the current directory is `bin`.

7. The `libice.la` library can be added to any other project for use as a general database engine for other projects. The `ice_example` binary is an example application that uses the library. The shell, `ice`, uses the library as well.

8. The binary `ice` needs to be run as root, so make sure to use sudo when running `ice`. The command will be `sudo ./ice` once in the bin/ directory.

7. Conclusion:

This project was started as an effort to understand database management on a systems level and to study the features, advantages and use cases for NoSQL database management systems. Based on the results, we have managed to reach this goal.

We started by studying the theory behind NoSQL database systems, why they were created, their advantages and how they are different with respect to relational database systems.

We then moved on to examining existing key-value stores like BerkeleyDB, Kyoto Cabinet, LevelDB, Redis and MongoDB. We studied their architectures and mechanisms to learn best practices and expected features for a generic key value store Key-Value store.

The planning phase started and we designed key components of the system like the modules, API, storage mechanism, data structure, etc. Much of this planning data can be found in the repo wiki:

- https://github.com/suhasks123/IceDB/wiki - For accessing on github
- https://github.com/suhasks123/IceDB.wiki.git - For cloning

We then started the implementation. The classes, their hierarchies, relationships and interactions were designed. The API code was written. The core data structure was implemented and integrated with the API.

Moving forward, we wanted to build an application that uses the engine so that it would be practically usable and also serve its purpose in demonstration of the engine. That turned out to be the ice shell.

The shell is supposed to mirror an external application that would have the engine integrated with it. It would also provide a functional way to interact with and manipulate data through the IceDB engine.

While this was a fulfilling experience, it is a project that we have spent countless hours on. For that reason, it would be better if the project was not exclusive to the DBMS course it was a part of. We want to expand its versatility and its use cases moving forward so that it is more of a fully realized application rather than just a project we started to learn about NoSQL databases. Thus we would continue to work on the project even after the completion of the course.

8. References:

- ➔ http://codecapsule.com/2012/11/07/ikvs-part-1-what-are-key-value-stores-and-why-implement-one/
- ➔ http://codecapsule.com/2013/05/13/implementing-a-key-value-store-part-5-hash-table-implementations/
- ➔ http://codecapsule.com/2013/04/03/implementing-a-key-value-store-part-4-api-design/
- ➔ https://www.pelock.com/products/string-encrypt#:~:text=String%20Encryption%20%26%20File%20Encryption%20for%20Developers,for%20any%20supported%20programming%20language.
- ➔ https://docs.mongodb.com/
- ➔ https://github.com/google/leveldb
- ➔ https://www.mongodb.com/nosql-explained