COLUMBIA UNIVERSITY
IN THE CITY OF NEW YORK

Micro DB

Presentation By:

Lulu (lz2761)

Chengrui(cz2664)

Shubham(sp3895)

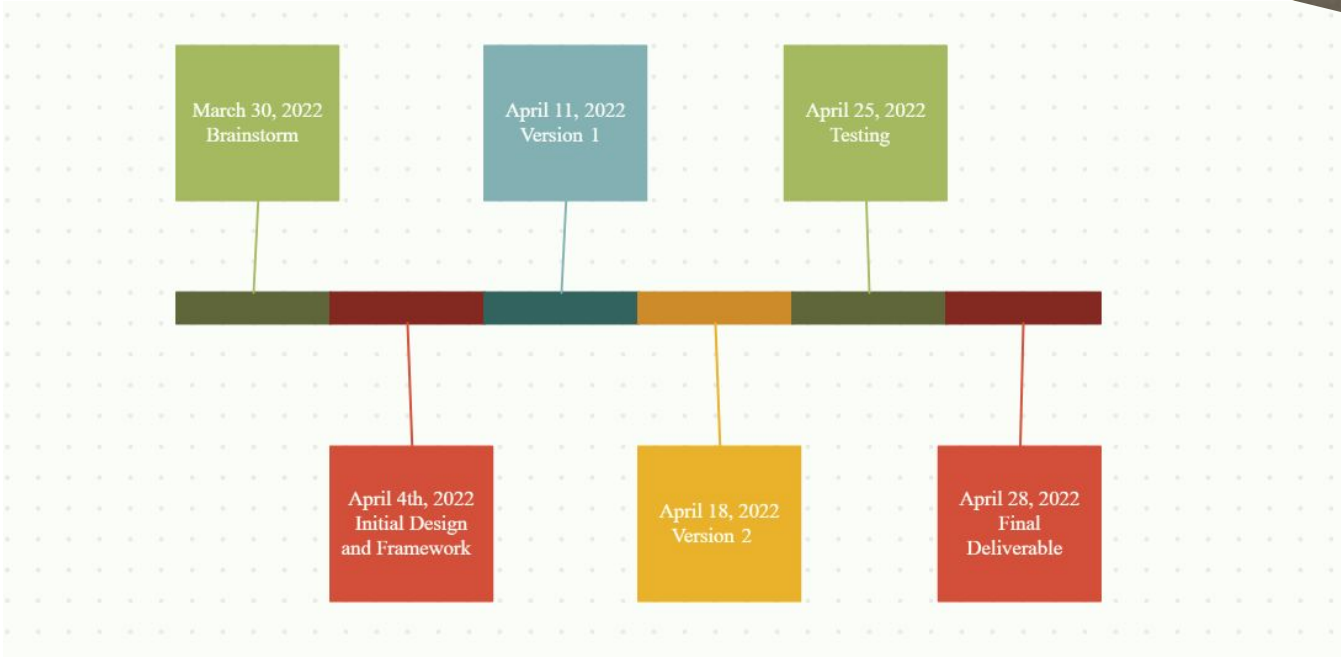
Acknowledgement

We want to convey our heartfelt gratitude to Prof Stroustrup as well as our TAs David and Luiz for providing us with a chance to work on this project and for helping us and guiding us along the way.

Abstract

- There are many C++ based SQL database frameworks like SQLAPI++, SOCI etc
- These need other dependent libraries to function properly.
- We were inspired by Redis and initially set out to built a replica using C++.
- Redis uses in-memory data storage but due to time constraints and scope of this project, we opted to provide similar functionalities by file-storage based implementation.
- **Our goal is to develop an easy to use self-contained library that can support key-value pair storage into filesystem.**
- The program uses idx and DAT files to store and retrieve data

Development Timeline



A little detail about each of these steps

Brainstorm: March 30th to 4th April : Narrowed down set of functionalities that the project will support based on the needs of users and their pain points.

Initial Design and Framework: April 4th to April 11th :Decided the hashing function and storage architecture

Version 1: April 4th to April 11th : Developed the insert, find, delete and create database functionality

Version 2: April 11th to April 18th: Added support to insert values into database on file uploads. Added support for query functionality

Testing: April 18th to April 25th: Added performance test application and tested with datasets from IMDB

Final Deliverables: April 25th to April 28th : Created tutorial, demo and sample docs.

Commands we support

1. INSERT : to add a key-value pair into the database
2. FIND : to get value of any key passed
3. DELETE : to delete a key-value pair from the database
4. QUERY : to search through database based on condition of key or value at particular index
5. IMPORT : import data from a csv or tsv file into database

Control Flow for each can be referred in the design document here.

Moreover, we added a test application in the project which also acts as a tutorial sample on how to use the library. The test application captures time taken for each of the above command and also does a sanity test of the database by testing random string inserts.

FileList

It's a simple directory structure and each file is named to be self-explanatory of what it does

File Name	Purpose
CMakeLists.txt	CMake file to build the project
Manual	Instruction Manual
data.tsv	Test dataset from IMDB
db.cpp	Function definitions
db.h	Function declarations
exception.h	Exception declarations
idx.h	Index file declarations
main.cpp	Application file which provides command line interface to test
test.cpp	Test application
tutorial.cpp	Tutorial file

Design Approach

Hash Function

```
unsigned int DB::hash(const char* key) {
    unsigned int seed = 1313131;
    unsigned int h = 0;
    while (*key) {
        h = h*seed + (*key++);
    }
    return ((h & 0x7FFFFFFF) % HASH_SIZE);
}
```

BKDR hash function (comes from Brian Kernighan and Dennis Ritchie):

Hash value of string s = seed^(n-1)* s[n-1] + seed^(n-2)*s[n-1] + …… + seed^0*s[0]

```
#define HASH_SIZE  20000003
```

Design Approach

DB class

```
class DB {
public:
    DB(string fileName);
    unsigned int hash(const char* key);
    bool open();
    bool close();
    bool insert_file(string file_name, int num);
    vector<string> find(const char* key);
    bool del(char* key);
    bool insert(char* key, vector<string> val);
    vector<string> query(int k, string val);
    void clear();
private:
    Idx* find_key(const char*key);
    unsigned int last_idx_off;
    unsigned int last_dat_off;
    string fileName;
    string idxName, datName;
    FILE* fp1;
    FILE* fp2;
};
```

Design Approach

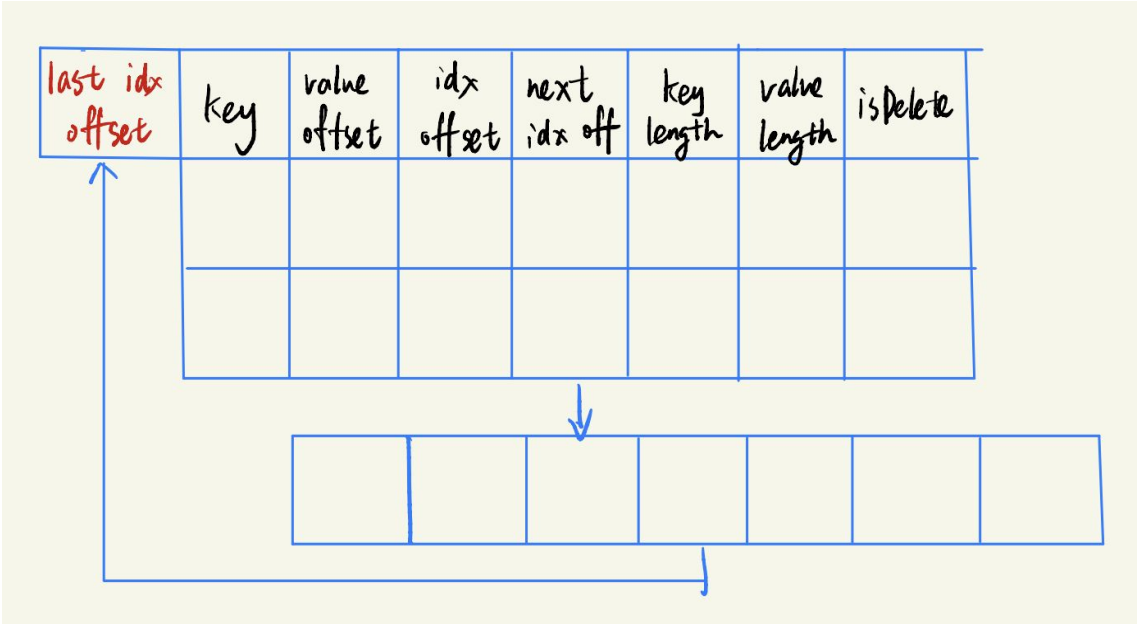
Index struct

Solve hash value conflicts

value:support single string/multiple strings(<=10)

```
struct Idx {
    char key[KEYSIZE_MAX];
    unsigned int value_off;
    unsigned int off;
    unsigned int off_next;
    int len_key;
    int len_value[10];
    bool isDelete;
};
```

How we organize the index file



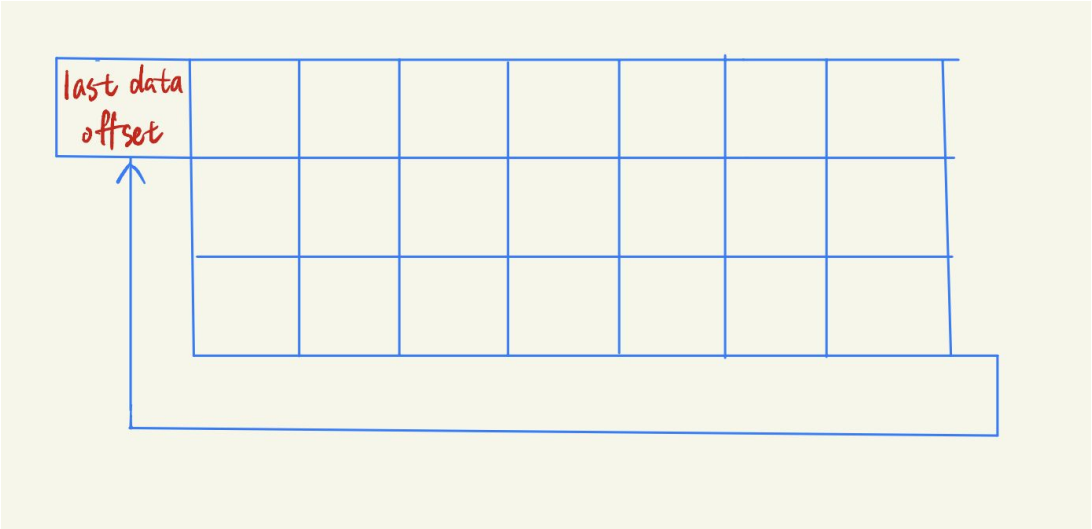
When key value conflicts happens, append the new idx struct to the end of the file.

Set the next_idx_off of the last conflict key's idx struct as the offset of the new idx struct. Thus, the hash table is organised as a Linked list.

When a key's idx struct is deleted, we do not truly delete it. Only set "isDelete" to be true. We can directly insert a new idx struct to the position where "isDelete" ==

How we organize the data file

The value data of each key is stored sequentially in the dat file.



Exception Handling

We wrote a exception class to handle some potential problems. We take some cases in consideration that there may be some incorrect input like strings or characters when the system wants us to input a number.

Or we input the wrong number which is not valid for it to work. Like it requires us to input numbers between 1 ~ 4, but we input 8.

```
Enter 1 to insert, enter 2 to search, enter 3 to delete, enter 4 to query, enter -1 to return to the previous menu
sd
Please input a valid type

Please input a valid number between -1 and 2:
```

Exception Handling

This is the exception class we defined.

```cpp
class Myexcept
{
protected:
    string message;

public:
    Myexcept(string_view str = "There may be a problem") : message{ str} {}
    virtual ~Myexcept() = default;
    virtual string_view what() const {
        return message;
    }
};
```

```cpp
class InvalidNumber:public Myexcept
{
public:
    InvalidNumber(string_view str = "Please input a valid number") : Myexcept{str} {
        cout << "Please input a valid number" << '\n' << endl;
    }
};
```

```cpp
class InvalidType:public Myexcept
{
public:
    InvalidType(string_view str = "Please input a valid type") : Myexcept{str} {
        cout << "Please input a valid type" << '\n' << endl;
        cout << "Please input a valid number between -1 and 2:" << '\n' << endl;
    }
};
```

These are the subclasses of Myexcept for some specific cases.

Sample

For Creating database file : On running the executable file, it asks the user for a database name, if a dat and idx file with same name is present in the directory it will connect to that database else it will create new set of dat and idx file.

```
C:\Users\91956\microdb_2804\redisproj\microdb.exe
Welcome to MicroDB!
Enter the name of db. if you want to delete a db, please enter 'delete-' + name of db:
demotest
An empty DB will be built!
Db has been open.
```

Sample

Delete database: User would enter the name of the database to be deleted. if the database is found, then the idx file and dat file of this database would be deleted. Otherwise, the program would warn that the database is not found and return to the menu.

```
Enter the name of db. if you want to delete a db, please enter 'delete-' + name of db:
delete-demotest
Db has been found.
clear successfully!
Enter 1 to continue the program. Enter -1 to exit the program
```

Sample

Insert: This command adds a key-value pair to the database file(Notice: in the current system, input of more than 10 values is not allowed. We can insert as many as we want, but it could only read 10 values.)

```
------------------------insert operation-----------------------
insert key :
key1
insert value (use comma to split value):
This,is,a,sample,input,1,2,3
Insert operation complete.
Enter 1 to insert, enter 2 to search, enter 3 to delete, enter 4 to query, enter -1 to return to the previous menu
```

Sample

Find: This function searches through the database for the specific key value and returns the value vector

```
-------------------------find operation-----------------------
enter key :
key1
the 0th  value is: This
the 1th  value is: is
the 2th  value is: a
the 3th  value is: sample
the 4th  value is: input
the 5th  value is: 1
the 6th  value is: 2
the 7th  value is: 3
Find operation complete.
Enter 1 to insert, enter 2 to search, enter 3 to delete, enter 4 to query, enter -1 to return to the previous menu
```

Sample

Query: This function returns all keys that fulfill the query requirement. Specifically, it iterators through all keys in the database and check if the kth value of each key equals to what the user query.

```
-------------------------query operation-----------------------
enter the value :
sample
enter the position of value :
3
The following keys are what you want :
1. key1
The query operation is complete.
Enter 1 to insert, enter 2 to search, enter 3 to delete, enter 4 to query, enter -1 to return to the previous menu
```

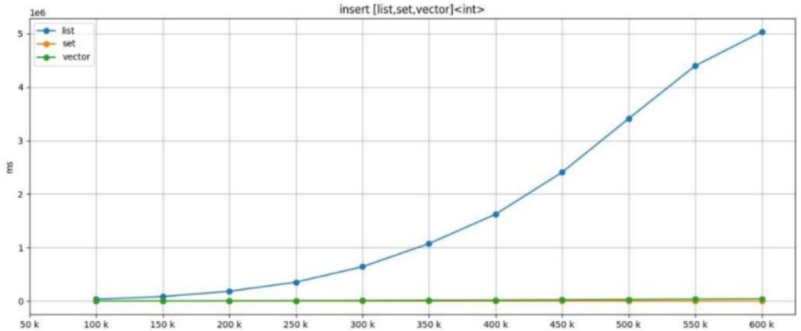
Sample

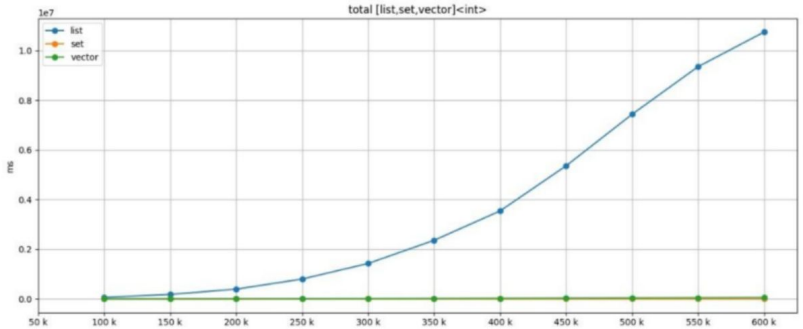
Delete: This function deletes a key-value data.. The program would first check if the key-value pair exists. If exists, it would delete it. Otherwise, it would warn the user that the key is not found.

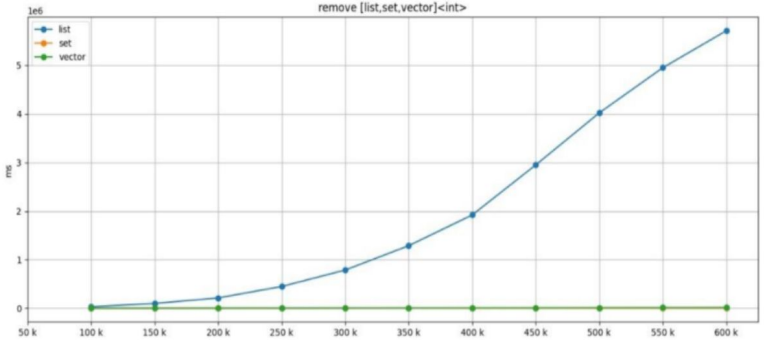
```
-------------------------delete operation-----------------------
enter key :
key1
delete operation complete.
Enter 1 to insert, enter 2 to search, enter 3 to delete, enter 4 to query, enter -1 to return to the previous menu
```

Performance Measurement

Inserting performance:





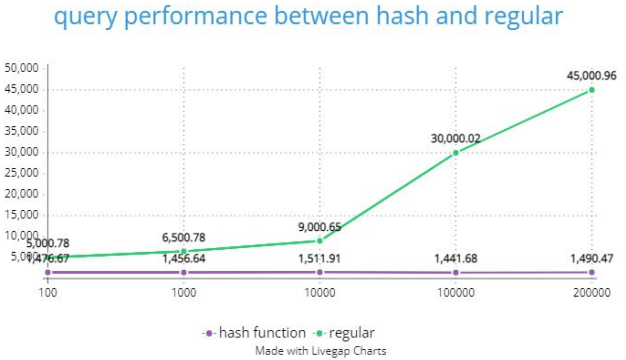


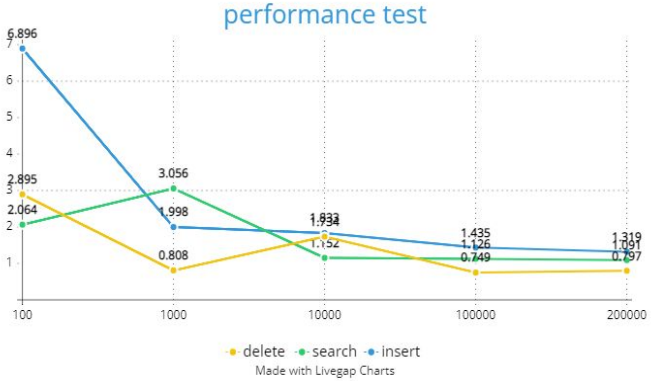
Performance Measurement

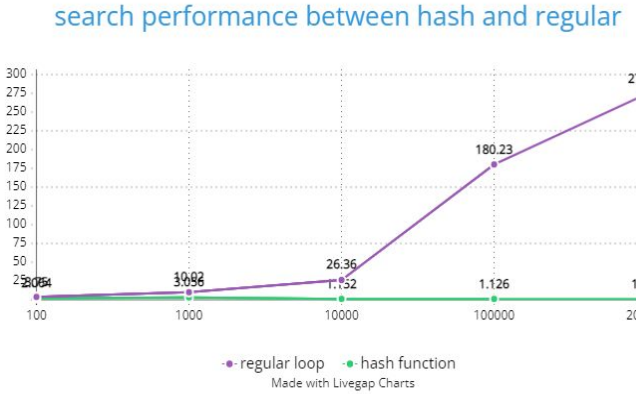
Function	Operations	Measure
Insert	6.7	millisecond/Insert
Find	0.8	millisecond/Read
Query	1.85337	Seconds for 100000 rows in Database
Delete	1.1	millisecond/Delete
Import csv	56,139	Inserts/Second

Performance Measurement

Performance between hash function
and normal reading function:







Correctness Measurement

We defined the max num of key string and single data value, and we randomly generated some value according to the given length.

```
unordered_map<char*, vector<string>> random_generate() {
    srand( Seed: time( Time: NULL));
    unordered_map<char*, vector<string>> m;
    for (int i = 0; i < 5; i++) {
        vector<string> temp;
        char*key = randomString( max: 5);
        while(m.contains( x: key)){
            char*key = randomString( max: 5);
        }
        for(auto j = 0; j < rand() % VALUENUM_MAX + 1; j++){
            string tempstr = randomString( max: VALUESIZE_MAX - 1);
            temp.push_back(tempstr);
            if(temp.size() > 10) break;
        }
        m.insert( x: unordered_map<char*, vector<string>>::value_type( x: key, y: temp));
    }
    return m;
}
```

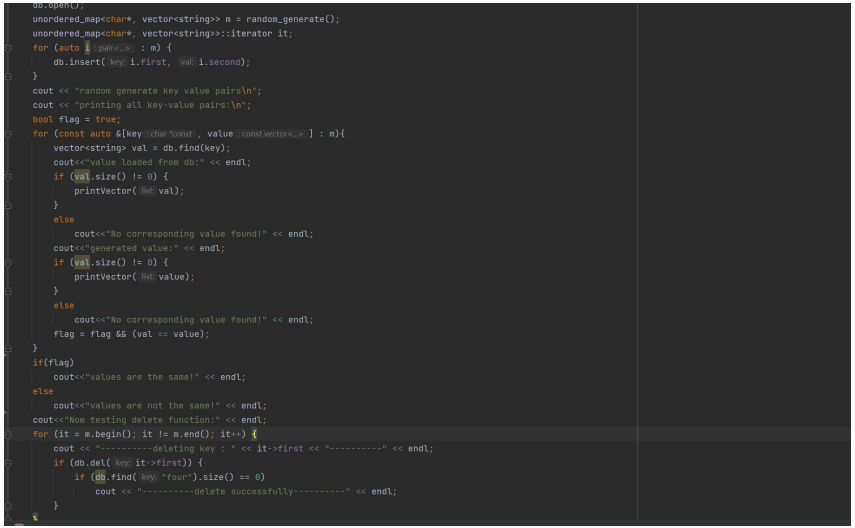
Correctness Measurement

We stored the data we generated into a map and insert them into the database.

```
db.open();
unordered_map<char*, vector<string>> m = random_generate();
unordered_map<char*, vector<string>>::iterator it;
for (auto i : pair<…> : m) {
    db.insert( key: i.first, val: i.second);
}
cout << "random generate key value pairs\n";
cout << "printing all key-value pairs:\n";
bool flag = true;
for (const auto &[key : char *const , value : const vector<…> ] : m){
```

Correctness Measurement

Then we read the value from both the database and the unordered_map, we compared them, then we delete them and see whether they were existed in the database.



```
db.open();
unordered_map<char*, vector<string>> m = random_generate();
unordered_map<char*, vector<string>>::iterator it;
for (auto i :pair<...> : m) {
    db.insert( key: i.first, val: i.second);
}
cout << "random generate key value pairs\n";
cout << "printing all key-value pairs:\n";
bool flag = true;
for (const auto &[key :char*const , value :const vector<...> ] : m){
    vector<string> val = db.find(key);
    cout<<"value loaded from db:" << endl;
    if (val.size() != 0) {
        printVector( lst: val);
    }
    else
        cout<<"No corresponding value found!" << endl;
    cout<<"generated value:" << endl;
    if (val.size() != 0) {
        printVector( lst: value);
    }
    else
        cout<<"No corresponding value found!" << endl;
    flag = flag && (val == value);
}
if(flag)
    cout<<"values are the same!" << endl;
else
    cout<<"values are not the same!" << endl;
cout<<"Now testing delete function:" << endl;
for (it = m.begin(); it != m.end(); it++) {
    cout << "----------deleting key : " << it->first << "----------" << endl;
    if (db.del( key: it->first)) {
        if (db.find( key: "four").size() == 0)
            cout << "----------delete successfully----------" << endl;
    }
}
```

Testing Mechanism

- We tested with two mechanism
  - First, through command line interface
    - User can perform operations on any database file through this interface
    - Sample log from command line interface Link
  - Second, through test_performance application
    - Test app benchmarks all operations for different size of input file and prints log
    - Test checks correctness of the database by generating random string and testing read/write values based on it.
    - Sample test_performance log file Link.
- Databases Used for testing:
  - https://datasets.imdbws.com/
    - Title.ratings.tsv.gz
    - title.akas.tsv.gz

Problems faced:

An ideal size of hash table: time cost v.s. space cost

If the hash size is very large:

the idx file would be very big and takes lots of space.

If the hash size is very small:

It would suffer from key value conflict issue. We need to go through the linked list for conflicting keys in find and delete function, thus very time consuming.

Problems faced:

Fix the value length or not?

For single value, it is better to fix the value length.

So that we can implement update function without wasting much space.

Fix the value length or not?

This is our original update function.

```
bool DB::replace(char*key, char*value) {
    //cout << "replace function\n";
    Idx* Idx_find = find_key(key);
    if (Idx_find == NULL) {
        return false;
    }
    else {
        unsigned int n = Idx_find->value_off;
        Idx_find->len_value = strlen( Str: value);
        Idx_find->value_off = last_dat_off;
        last_dat_off += (Idx_find->len_value + 1) * sizeof(char);
        fseek( File: fp2, Offset: sizeof(int) + Idx_find->value_off, Origin: 0);
        fwrite( Str: value, Size: sizeof(char), Count: Idx_find->len_value + 1, File: fp2);//replace val
        fseek( File: fp1, Offset: sizeof(int) + sizeof(Idx)*Idx_find->off, Origin: 0);
        fwrite( Str: Idx_find, Size: sizeof(Idx), Count: 1, File: fp1);//renew index file
        fflush( File: fp1);
        fflush( File: fp2);
        return true;
    }
}
```

```
insert operation
insert key:
key1
insert value:
value1
Insert operation complete
Enter 1 for insert operation, enter 2 for search operation, enter
ion, enter 5 to print the entire database, enter -1 to return to
3
replace operation
insert key:
key1
insert value:
value2
Replace operation complete
Enter 1 for insert operation, enter 2 for search operation, enter
ion, enter 5 to print the entire database, enter -1 to return to
2
find operation
insert key:
key1
value is: value2
Find operation complete
Enter 1 for insert operation, enter 2 for search operation, enter
ion, enter 5 to print the entire database, enter -1 to return to
```

Original output

Problems faced:

Fix the value length or not?

For multiple values, it is not easy to handle.

Now we only restrict the total number of values for each key to be less than 10.

If we fix the value length, it would be a waste of space for short length value..

If we do not fix the value length, we have trouble implementing update function. Because the data is stored in sequence. If the new value we try to update is longer than the previous one, it would cover the data stored next to it.

Encrypt and decrypt functions and class

```
class Encryption {
    int key;

    // File name to be encrypt
    char c;

public:
    void encrypt(std::string filename);
    void decrypt(std::string filename);
    void encrytFunction(std::string filename);
};
```

Problems faced:

Encryption and decryption functions:

At first we designed an algorithm based on MD5 and RSA and tried to realize the encryption and decryption functions of the dat files. And we made it. We successfully used password to realize that. However, even though we could decrypt the content of the file which was totally same as the previous content, yet we couldn't successfully search the value in the database, which may generate some garbled text when we search the key.

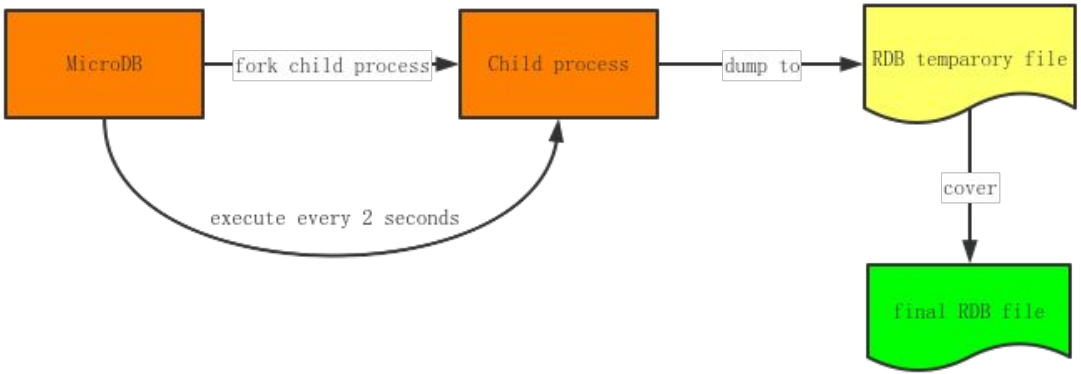
Further tasks:

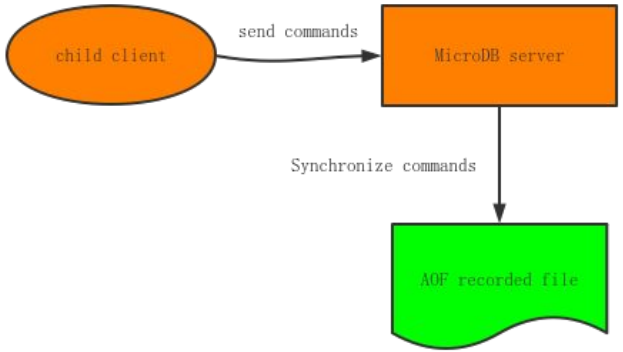
This implementation is file system based, in future we would like to add function like in-memory based implementation similar to Redis in design, we would need to add functions to maintain concurrency control and persistence.

Further tasks:

Now our product is just a simple version of the NoSQL database. There are still a lot of things for us to do:

1. About persistence now we only store all the data into the file on our local disk. In the future we will try to do some more persistence functions.





Further tasks:

Now our product is just a simple version of the NoSQL database. There are still a lot of things for us to do:

2. No we only have the version on our local computer. If we want to make our database more functional, we will introduced the distribution lock to this database to serve the scenario of huge amount of users.

3. Current implementation assumes that all keys and values vector are string, we want to try to move it to template based implementation so that the user can type of data while creating the database. This would provide better speed of operations when the user needs to use simple data types like int for values.

4. Convert to module(C++20) for easier integration with new applications

Further tasks:

Now our product is just a simple version of the NoSQL database. There are still a lot of things for us to do:

4. We also plan to add more functions on servers like master-slave synchronization, transactions and sentinel mechanism to make it more efficient and comprehensive.

5. When memory is low, we plan to free memory by evict some key according to LRU.

6. Convert to module(C++20) for easier integration with new applications

Thank you

Opening the floor to Q&A