BUILDING A DATABASE IN RUST

Submitted for the Degree of MEng in Computer Science, 2019

JOHN MCMENEMY

STUDENT ID: 201748244

Except where explicitly stated all the work in this report, including appendices, is my own and was carried out during my final year. It has not been submitted for assessment in any other context.

I agree to this material being made available in whole or in part to benefit the education of future students.

Project Aims & Objectives

Rust (https://www.rust-lang.org/en-US/) is a relatively new programming language developed by Mozilla. It is intended to allow low level programming in a "safe" way -- there should be none of the memory errors, undefined behaviour, and race conditions that often arise in other low level languages like C and C++.

Rust enables safe low level programming by making the notion of "lifetime" explicit in programs. A lifetime tracks the parts of a program that have access to a piece of memory, preventing errors such as "use after free" and accessing memory on stack frames that have been deallocated. Lifetimes also enable race free concurrency.

The objective of this project is to use Rust to implement a simple database server in order to gain experience in how Rust's features help or hinder safe systems programming.

Lastly, through this project I hope to learn more about how databases work: learn how they are implemented at a low level and understand why certain decisions are made for certain databases i.e. understand the design decisions behind many different databases.

Development Methodology

The development methodology works through simple steps or milestones to achieve as much progress in the time allocated to produce the project, the steps that I decided upon with the help of my supervisor are as follows:

1. Learn Rust Basics
    a. Learn about data structures
    b. Implement a command line String Manipulator
    c. Learn about Object Orientation
    d. Learn about Traits & Guidelines
    e. Learn about Concurrency Mechanisms
2. Learn about asynchronicity
    a. Learn about Rust's Tokio Library
    b. Implement a Simple Server
    c. Expand the server to accept multiple concurrent connections
    d. Tie-in with Concurrency Mechanisms by implementing shared state variables
3. Learn about database design & implement a database
    a. Study different databases, concepts and important keywords
    b. Implement a simple key-value storage system into the server made at 2.d.
4. Improve the database
    a. Add different operations to the database
    b. Add logging to files
    c. Add writing to disk
    d. Add crash recovery
5. Bonus elements
    a. Add SQL like operation chaining

The steps laid out above are but guidelines only. As I have already discovered, some of the steps can be done before others depending on what I find and how my development goes. Lastly, if the implementation of all of the above is completed successfully, I could take on the challenge of making a relational type of database.

Testing

Initially, testing of my implementation will be done by myself with occasional checks from my supervisor. At the later stages of the project I suspect I can get more thorough testing done by asking other users and more in-depth testing by my supervisor.

Project Progress

Part I: Learning Rust

My initial reflex was to find resources for learning Rust in the university library. I was successful since I found the book: “Beginning Rust” by Carlo Milanesi [1]. I got it as an online resource through the library’s online search engine. For the first couple of weeks, my focus was on reading that book. For my second meeting with my supervisor I had produced a simple String manipulation program. Firstly, the user is asked on the terminal what it is they are wanting to do among the following options:

- Remove a character from a String
- Turn the entire String into uppercase
- Change the case (uppercase to lowercase or vice versa) of a single character in the String
- Split the text by some character or sequence of characters

Then, the program would ask the user to input a String and depending on the option that was chosen, it would prompt the user for a character or an option to choose from e.g. when changing case, what the desired case is.
Lastly, the user would see the output of the operation and they would be asked if they wanted to save the result into a text file (I did this to experiment with file access) (Snippet 2).
I mainly made use of Iterators and corresponding functions that work on them e.g. map, fold, collect and closures to make my program (Snippet 1).

```
pub fn remove_character() {
    println!("Please type something to eliminate a character: ");
    let input_text = read_text();

    let mut input_char: char;
    loop {
        println!("Please type the character you wish to delete: ");
        input_char = get_char();
        match input_char {
            '\n' => continue,
            _ => break,
        }
    }

    let result = input_text
        .trim()
        .chars()
        .filter(|c| *c != input_char)
        .collect::<String>();

    println!("The resulting string is: '{}'", result);

    save_to_file(result);
}
```

Functions that work on Iterators

Snippet 1: Function to remove a character from a String

```
pub fn save_to_file(string_to_save: String) {
    println!("Would you like to save the result to a file? type y/n");

    let choice = get_choice('y', 'n');

    match choice {
        'y' => {
            use std::io::Write;
            let mut file =
std::fs::File::create("result.txt").unwrap();
            file.write_all(string_to_save.as_bytes()).unwrap();
            println!("Done.");
        }
        'n' => (),
        _ => (),
    }
}
```

Saves to a text file

Snippet 2: Saving to the file system

Over the next couple of weeks, I finished reading the book. The last couple of chapters explained the concept of borrowing and Lifetimes. I was still confused after reading the book, so I looked for more information online. Among what I found, what stood out was a video on Youtube titled *Rust Tutorial - Lifetime Specifiers Explained* by the channel "BinaryAdventure" [2]. The video helped me understand better why the usage & implementation of Lifetimes is required. It also helped me understand the syntax better. Unfortunately, at this point I still felt it wasn't something I would understand properly unless I put it into practice myself.

I decided to put the research into Lifetimes on hold as I had agreed with my supervisor that I would aim to produce a simple server by our next meeting. So, I started learning about the I/O libraries in Rust and then came across “Tokio”. My supervisor had mentioned previously that I would probably have to make use of this library, so I decided to look into it. By following the tutorials in the Tokio documentation I made a simple “Echo” server (v1 on GitLab). This program, when you connect through a client like “Telnet” on Windows will immediately send back anything that is sent to it i.e. if you press “h” on your keyboard, you would receive “h” back immediately so your terminal would display “hh” (Snippet 3). This implementation of Echo did not satisfy me, I decided I wanted to implement a version where the sent data would only be returned upon a newline being received i.e. when the user presses the Enter key.

```
let (reader, writer) = socket.split();
let amount = io::copy(reader, writer);
let msg = amount.then(|result| {
    match result {
        Ok((amount, _, _)) => println!("Wrote {} bytes", amount),
        Err(e) => println!("Error: {}", e),
    }
    Ok(())
});

tokio::spawn(msg);
```

Getting input and output stream

Snippet 3: The part of the program that echoes back characters by using the copy function from the Tokio library

After a bit of research, I found out about the Tokio Codec library which is used to apply certain modifications to String data. The objects in the library work on the “Streams” (Input) and “Sinks” (Output) objects in Tokio’s I/O library. In the Codecs library one can find the “LinesCodec” object which splits data by using the newline (“\r\n” on Windows) character(s) (Snippet 4). Thanks to this I could now implement an Echo server (v2 on GitLab) that would only send back “h” if one pressed the Enter key. Therefore, unlike my previous version, now a user could type “hello” then the Enter key and get “hello” back.

```
let (lines_tx, lines_rx) = LinesCodec::new().framed(socket).split();

let responses = lines_rx.map(move |incomming_message| {
    return incomming_message;
});
```

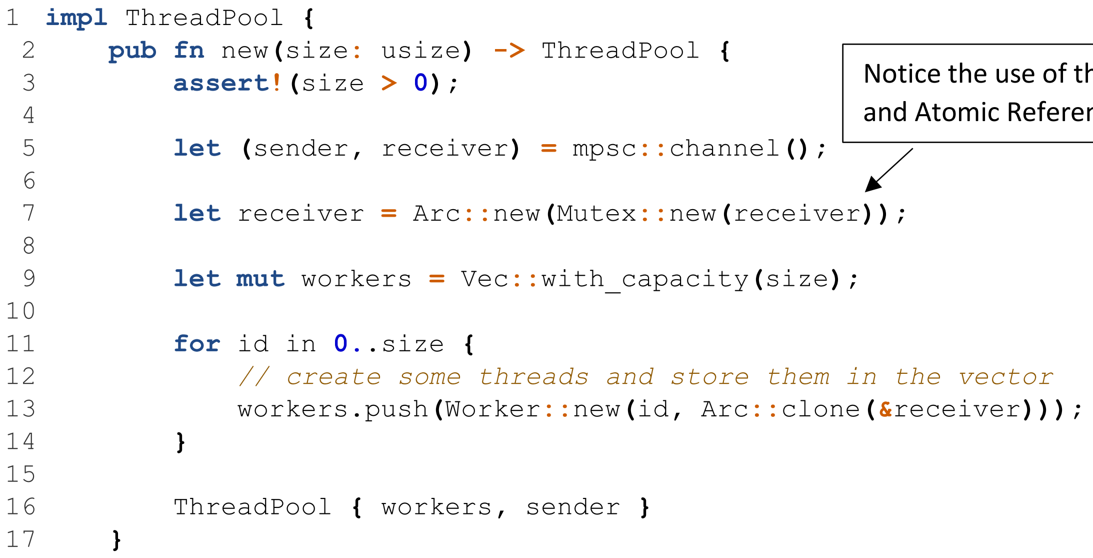
Snippet 4: The only changes to be made compared to the previous Snippet, notice the use of the LinesCodec object

I was still feeling a bit confused about how Tokio worked, especially when it came to its asynchronous logic. So, in the hopes to get more knowledgeable about Tokio, I came across a video of a lecture from “RustFest” done in Zurich in 2017 [3]. I feel like this video helped me learn about the thoughts that were behind the development of Tokio and why it was designed the way it is i.e. what issues arose when asynchronicity was implemented into other languages and how Tokio could be developed while taking those issues into account. For example, when passing “Future” objects in between threads when dealing with concurrency the concept of ownership had to be dealt with.
I feel I now understand a bit more about the inner workings of Tokio, which I hope will help me when using the library for myself.

At my next meeting with my supervisor I presented to him the progress I had made and the programs I had developed. We agreed that the next step for me to take would be for me to implement a server that keeps the same state between sockets i.e. clients. For example, that one connected client can increase the value of a variable and that another client connected at the same time can read the value in this variable and modify it. Clearly, the main challenge of this program is dealing with race conditions.

After some Googling, I came across a project tutorial [4] on the Rust Language book website where one builds and HTTP server starting from a simple single-threaded one and building upon it to get a more a more complex multi-threaded one. This interested me and thought I could learn enough from it that I could then transfer the knowledge to my Tokio Echo server.

When I completed the tutorial, I had implemented a simple HTTP server that was multi-threaded, I learned a lot about implementing one's own Thread Pool (Snippet 5) and working with Workers that could receive jobs. I also learned about sharing resources within threads by using Atomic Reference Counters (Arc) and Mutexes to be able to lock a variable before modifying it so that race conditions cannot occur. This last piece of knowledge was especially valuable since this is what I could apply to my Rust Echo server.



```
impl ThreadPool {
    pub fn new(size: usize) -> ThreadPool {
        assert!(size > 0);

        let (sender, receiver) = mpsc::channel();

        let receiver = Arc::new(Mutex::new(receiver));

        let mut workers = Vec::with_capacity(size);

        for id in 0..size {
            // create some threads and store them in the vector
            workers.push(Worker::new(id, Arc::clone(&receiver)));
        }

        ThreadPool { workers, sender }
    }
```

Notice the use of the Mutex and Atomic Reference Counter (Arc)

Snippet 5: The ThreadPool I implemented to learn about Atomic Reference Counters and Mutexes

Using the knowledge I gained, I implemented a counter variable into my "Echo" server (called "counter server" on GitLab) using the Arc and Mutex Objects so that when multiple clients connected over a system like "telnet" one could modify the value and another could read it and the changes the former made were reflected in the latter's terminal (Snippets 6 & 7). There were three possible actions a connected client could take: read the variable, increment the variable and decrement the variable by typing "read", "increment" and "decrement" respectively (Snippet 8).

```
fn main() {
    let addr = "127.0.0.1:6142".parse().unwrap();
    let listener = TcpListener::bind(&addr).unwrap();

    let counter = Arc::new(Mutex::new(0));

    ...
}
```

Snippet 6: Adding the counter variable wrapped in a Mutex

```
        .for_each(move |socket| {
            let counter = Arc::clone(&counter);
```

Snippet 7: Cloning the Mutex for each connection socket so as to have and keep track of multiple "owners" for one Rust variable

```
                "increment" => {
                    let mut value = counter.lock().unwrap();
                    *value += 1;
                    return format!("After being incremented, the counter reads: {}\n", *value);
                }
```

Snippet 8: Getting the counter's lock so as to modify it when the connected client requests so, in this case incrementing the variable. Locking the variable is of paramount importance as this prevents concurrent modifications

Therefore, I managed to implement a Shared-State variable into my Rust server which would no longer be an “Echo” server but simply a server where a variable can be manipulated by multiple clients. Technically, I have now got my first working database, one where an integer value is stored in volatile (RAM) memory that is only stored while the program is running.

For now, the knowledge I gained on thread pools and workers is not required for the Tokio server since the Asynchronous logic coded into the Tokio library takes care of handling multiple clients (connections).

Part II: Learning about Database Design

Now that I felt more confident with my Rust skills, I felt it was time to start thinking about how I’m going to build a Database using Rust.

I know very little about database design, therefore that is what I decided to research next.

Firstly, I came across a lecture from the 2018 “FOSDEM” event [5] which according to their website at https://fosdem.org/ is a “[…] free event for software developers to meet, share ideas and collaborate.” It is held in Brussels.

The lecture, given by Siddon Tang talked about using Rust to Build a Distributed Transactional Key-Value Database. Unfortunately, even though this lecture gave me knowledge about what tools and libraries are available to get a database up and running in Rust, it was too high-level in terms of its thinking. It seems the lecture is aimed more at what a business could do rather than what I’m looking for which is to get into the low-level technical details of database design so I can build one myself from scratch. Thankfully it did offer a clue as to what I could research next: it was mentioned that for the database’s key-value storage engine, the Rust wrapper library for “RocksDB” could be used. I thought that maybe I could see how this library is implemented so that I can gain more of the low-level knowledge I’m looking for.

Before looking into RocksDB, I stumbled upon a blog series by "Emmanuel Goossaert" [6] where he documents his journey into developing a key-value storage system using C++ and HashTables, I find this very interesting because what he is doing is basically what I'm trying to do but with Rust (and also I would probably use BTrees instead of HashTables since that is what Rust supports well).

I've also been thinking about how I can design a simple key-value store building upon the Tokio server I have developed. Basically, I think I can build a Rust module where I can encapsulate all the database management functions and the actual BTree on there and use my Tokio server implementation to get commands from a user (e.g. "Set Name 'John') parse them, and call the proper functions from my module to store the desired information into the BTree. At the moment this is a memory only implementation, when I achieve this, I would think of implementing disk writing.

After meeting with my supervisor I decided that the couple of weeks before the project report submission was due my work would consist of the following two elements: firstly, doing research behind key-value databases (which I explained in the related work part of the report) and secondly, to try to implement the design I had come up with before: a BTree database in Rust using what I have built previously. He mentioned that it might not be possible to have a separate "database" module in Rust that hold the actual database object, this is more like Java thinking so he suggested that first I implement the database object and the functions all in the one file, in a procedural programming type of way.

Related Work

Most of my related work focused on researching database design and studying different key-value storage implementations. I decided to research key-value storage databases specifically because my initial milestone for this project is to implement a simple Key-Value storage database in Rust.

RocksDB

I decided to start by looking into "RocksDB" as I had heard about this database before while watching a seminar on a database implementation in Rust, as I mentioned previously in my report.

RocksDB [7] is a database maintained by Facebook and it is based on another database called "LevelDB" with the aim of being specially tailored for fast storage media, specifically Flash media. It aims to stand out for server workloads that include high-random reads and high-update reads (i.e. overwriting).
When it comes to the architecture (design) of it, the developers base their database on 3 foundational objects: the "memtable" which is a data structure that is in-memory (RAM), the "logfile" which keeps track of changes done to the memtable and is always written to permanent storage (Hard Disk or Solid State Drives) and the "sstfile", which is the one that hold the database structure in permanent storage.
When a change is made to the DB through some of the operations provided by the DB such as Get, Put, or Delete, the changes are made to the memtable and written to the logfile. Once the memtable fills up (because the OS may not be able to prove more RAM to the DB), the memtable gets "flushed" to the sstfile i.e. all of the changes made to the database are written to permanent storage and then the memtable get cleared of data. The logfile is then removed so a new one can be created for the newly cleared out memtable. The data is stored in sorted order according to an Iterator definition.
The above is the basic architecture I want to follow for my key-value database, that's why reading about RocksDB's architecture helped concretize my ideas.

The database also implements checksums to prevent against corrupted data.

Lastly, on the database’s wiki I read they “provide different types of ACID guarantees” and that they support “optimistic” & “pessimistic” transactions. I had heard of ACID before, but I still don’t know what it is, and I had never heard of pessimistic & optimistic transactions”. For those reasons I decided to find out about them next.

ACID & Database Transactions

According to an article on *Lifewire* “The ACID model of database design is one of the oldest and most important concepts of database theory.” [8] It is a set of four properties that database systems must try to meet as these four properties, when met by a database, indicate that the database is reliable. It is probably the most popular database paradigm. I have also now learned that a Transaction is an operation or sequence of operations on the database that satisfy the ACID properties. [9] These four properties are the following:

- **Atomicity**: This means that if a transaction consists of multiple operations on the database, the Database Management System (DBMS) has to have measures in place to guarantee that either all the operations of the transaction complete or if one fails then all of the operations fail. e.g. if a transaction consists of a read, write and deletion, if the writing operation fails for some reason (like a hardware or software failure) then the deletion operation must not happen.
- **Consistency:** This means that each transaction on the database must always comply with the rules of the database. For example, in an SQL database where there is a column defined with the datatype “Date” then the DBMS must not allow a value like “John” to be stored in it, otherwise this would break the rules of the database. If for some reason, a transaction happens that violates the rules of the database, the DBMS must have measures in place to roll back the database to a previous state where the rules have not been violated.
- **Isolation:** Isolation is very important in the context of concurrent modifications and databases that have multiple users, as isolation means that if two different transactions have to take place, say by two different users, they must happen without interfering with each other. One of the ways to achieve this is to use a transaction queue where only one transaction can happen at the same time. If two transactions happen concurrently, they must not modify the same value. In my Rust project I have achieved this by using Atomic Reference Counters (Arcs) and Mutexes where a value grants a lock to the thread that wants to modify it and other threads must wait to acquire the lock before they can do so themselves.
- **Durability:** Lastly, durability simply means that information must not be lost by the database. For example, by using backups and “write-ahead logging” which writes transactions to a log before they are actually committed to the database. Since writing to a log is very quick, if there is a problem like a hardware failure during the actual transaction then the transaction is not lost once the database is restored as it was written to the log.

The “write-ahead-logging” method also ensures atomicity since if some of the operations in the transaction have taken place, but not all before the failure, then through scanning the log, the DBMS can tell what was left to be performed of the transaction.

Log Structured Database Design

On the main website of RocksDB (rocksdb.org) they mention that “RocksDB uses a log structured database engine”.

I decided to look into what that is. I came across a blog post by Nick Johnson [10] where he described this system in the context of databases (because this system can also be used for filesystem applications, that's how it originated in the 1980s).

Log structured design is a way to store data where the data is never overwritten in the disk, it is always appended to then end of the previous piece of data. At the end of the database storage file, which can be thought of as a log (hence the name) an index node keeps track of the most up-to-date values. Every time a transaction is complete, the index is updated. Some of the advantages of this method are:

- Cleaning up unused disk space is quite easy when you break the storage up into chunks: once a chunk has very little values in it or none at all, you can move those values to another chunk and mark that section of the disk as being free.
- Concurrent transactions can be more easily handled. In a read operation for example, the DBMS can access the last index and not worry about data being modified as once it has read the index it holds a "snapshot" of the database at the time that will not change since in this system, existing data is never modified. I learned that this is called Multiversion Concurrency Control (MVCC) [11] . During a writing operation, a way to check that data that a transaction wants to modify has not being modified by another transaction is by looking at the most up to date index before modification and checking that the index node still points to the data that we want to modify, if it does, then the data has not been modified by another transaction otherwise we just do the whole read operation again to get the most up-to-date data again. All of this without having to using write locks. This latter methodology is called Optimistic concurrency control. [12]

I have learned that many databases employ this design or aspects of it, among them: RocksDB, CouchDB, PostgreSQL, Apache Cassandra, Datomic…

Note that these are not only key-value databases but also relational, this shows the universal utility of this design method.

BTrees

I decided to do a little bit of research on BTrees as my memory wasn't fresh from when this was mentioned in university classes. I now understand that BTrees are self-balancing tree data structures that try to minimize tree depth, therefore BTrees are very wide trees. The main benefit of BTrees is that disk access times are minimized as much as possible [14] and since disk access time is significantly slower compared to main memory access time, this means that operating data on BTrees is significantly quicker in comparison to other Tree data structures like Binary Search Trees.

Design

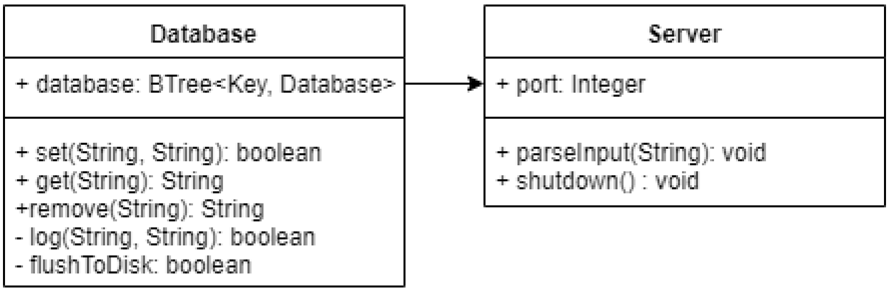

Database
+ database: BTree<Key, Database>
+ set(String, String): boolean
+ get(String): String
+remove(String): String
- log(String, String): boolean
- flushToDisk: boolean
Server
+ port: Integer
+ parseInput(String): void
+ shutdown() : void

Figure 1: Key Value Database Design

The above diagram illustrates the first major Milestone which would be to achieve a Key-Storage database with a server. Initially it would be memory only but then it would expand to have a log and it would actually write to disk. Afterwards, maybe a crash recovery functionality could be implemented.

References

1. Milanesi, C. *Beginning Rust: From Novice to Professional.* Apress. 2018. https://books.google.co.uk/books?id=LM5SDwAAQBAJ
2. BinaryAdventure. *Rust Tutorial - Lifetime Specifiers Explained*. YouTube. 2018. Retrieved on the 17th of October 2019. https://www.youtube.com/watch?v=QoEX-Vu-R6k
3. Rust. *RustFest Zürich 2017 - Tokio: How we hit 88mph by Alex Crichton*. YouTube. 2017. Retrieved on the 23th of October 2019. https://www.youtube.com/watch?v=4QZ0-vIIFug
4. Klabnik S. Nichols C. The Rust Programming Language. Rust Lang Org. 2018. Retrieved on the 25th of October 2019. https://doc.rust-lang.org/book/ch20-00-final-project-a-web-server.html
5. Siddon T. TiKV - building a distributed key-value store with Rust. FOSDEM. 2018. Retrieved on the 30th of October 2019. https://archive.fosdem.org/2018/schedule/event/rust_distributed_kv_store/
6. Goossaert. E. *Implementing a Key-Value Store*. codeCapsule. 2017. Retrieved on the 1st of November 2019. http://codecapsule.com/2012/11/07/ikvs-implementing-a-key-value-store-table-of-contents/
7. Borthakur D. et al. *RocksDB Basics.* GitHub. 2019. Retrieved on the 13th of November 2019. https://github.com/facebook/rocksdb/wiki/RocksDB-Basics
8. Chapple M. *The ACID Database Model.* Lifewire. 2019. Retrieved on the 13th of November 2019. https://www.lifewire.com/the-acid-model-1019731
9. *ACID*. Wikipedia. 2019. Retrieved on the 13th of November 2019. https://en.wikipedia.org/wiki/ACID
10. Johnson N. *Damn Cool Algorithms: Log structured storage.* NotDot. 2009. Retrieved on the 13th of November 2019. http://blog.notdot.net/2009/12/Damn-Cool-Algorithms-Log-structured-storage
11. *Multiversion concurrency control.* Wikipedia. 2019. Retrieved on the 13th of November 2019. https://en.wikipedia.org/wiki/Multiversion_concurrency_control
12. *Optimistic concurrency control.* Wikipedia. 2019. Retrieved on the 13th of November 2019. https://en.wikipedia.org/wiki/Optimistic_concurrency_control
13. *Introduction of B-Tree.* GeeksforGeeks. 2013. Retrieved on the 18th of November 2019. https://www.geeksforgeeks.org/introduction-of-b-tree-2/