BUILDING A DATABASE IN RUST

Submitted for the Degree of B.Sc. (Hons) in Computer Science, 2020

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.

Contents

Introduction

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.

Project Specification

The goal is to make a server-run key-value database. A series of user stories describe how this should be done on a high level.

As a user I:

- Would like to be able to connect to the database through the internet.
- Would like to be able to store values and assign them keys by typing commands such as "set 'Name' as 'John'".
- Would like to be able to delete and modify those values.
- Would like to be able to get a list of values and keys in the database.
- Would like the database to be shut down and rebooted without data being lost.
- Would like the database to have minimal if any data loss in the event of a crash.
- Would like other users to be able to access the database at the same time as myself.
- Would like to be able to access the database through the web i.e. through an API.

Outcome Overview

By the end of the project, most of the original specifications were met. A key-value database was coded. It runs on a server and one can set, remove and get values (among other commands). Nested keys are supported, much like the way a hard-drive storage directory works e.g. a command to set a user's admin level to "1" could be "set Users/PersonName/AdminLevel 1;". Overwriting values can be done by using the set command on the same key. All the keys of the database or a sub-directory can be obtained by using the get command e.g. "get;" for the keys at the base database level or "get Users" for the keys under the Users directory.

The database also has crash recovery mechanisms by the use of a log text file. This is also the way the database maintains data when it is offline.

The database supports concurrent user usage by queueing commands from each user.

The only feature from the original specification that was not implemented was the feature that allowed a user or program to connect over the web using the HTTP protocol.

Testing and benchmarking were performed using a mix of manual and automated testing.

Aside from the original specification, many features can still be added to the database. There are also some bugs and code nuances that could be improved. This is laid out at the end of this report under the section "Summary and Conclusions".

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 were followed to produce the program 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. Extra Features

       a. Add operation chaining (Transaction support)

Development was done in a fashion similar to the Agile Development methodology by implementing each step in 2-week blocks (which in Agile are called Sprints). Aside from the odd set-back, the plan was followed quite accurately.

Testing Strategy

Initially, the database was tested manually using the Windows Telnet client. Later on, automated tests were designed using a mix of Rust, by building a testing client that connects to the server, and Bash command line scripts. For details on this, please reference the [Testing and Benchmarking](#) section.

Report Structure

Following the introduction material, the report first accounts the research undertaken during the development of the project (Chapter 1). Then, a detailed explanation of the learning process for the Rust programming language can be found (Chapter 2). Following this, the report expands upon the specifics of developing the database, using code snippets when appropriate to explain in detail decisions made during development.

Thereafter, a detailed account of the testing and benchmarking process is given, showing results and explaining design decisions using screenshots to illustrate. (Chapter 3)

Lastly, a summary of the report is given as well as an explanation on future work that can be undertaken followed by conclusions that can be made from this experience.

The report is structured in a chronological basis, except for the related work section. A narrative inspired by a chronological style of writing has been implemented.

Chapter 1: Related Work

Most of the 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 was to implement a simple Key-Value storage database in Rust.

1.1. 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, which I mention further on in my report on

RocksDB [1] 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 provide 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.

1.2.   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." [2] 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. [3] 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.

1.3.   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 [4] 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) [5] . 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. [6]

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.

1.4.   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 [7] 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.

Chapter 2: Detailed Design & Implementation

In this chapter, one can find the detailed journey that was taken to achieve development of the database. First, one will learn about the small projects developed to learn the Rust programming language. Then, one can expect an explanation of the learning process undertaken to design the database. Lastly, a walkthrough of parts of the technical implementation.

2.1. 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 [8]. 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).

```rust
1  pub fn remove_character() {
2      println!("Please type something to eliminate a character: ");
3      let input_text = read_text();
4
5      let mut input_char: char;
6      loop {
7          println!("Please type the character you wish to delete: ");
8          input_char = get_char();
9          match input_char {
10             '\n' => continue,
11             _ => break,
12         }
13     }
14
15     let result = input_text
16         .trim()
17         .chars()
18         .filter(|c| *c != input_char)
19         .collect::<String>();
20
21     println!("The resulting string is: '{}'", result);
22
23     save_to_file(result);
24 }
```

Functions that work on Iterators

Snippet 1-1: Function to remove a character from a String

```rust
1  pub fn save_to_file(string_to_save: String) {
2      println!("Would you like to save the result to a file? type y/n");
3
4      let choice = get_choice('y', 'n');
5
6      match choice {
7          'y' => {
8              use std::io::Write;
9              let mut file =
10 std::fs::File::create("result.txt").unwrap();
11             file.write_all(string_to_save.as_bytes()).unwrap();
12             println!("Done.");
13         }
14         'n' => (),
15         _ => (),
16     }
17 }
```

Saves to a text file

Snippet 1-2: 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" [9]. 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.

```
1    let (reader, writer) = socket.split();          Getting input
2    let amount = io::copy(reader, writer);           and output stream
3    let msg = amount.then(|result| {
4        match result {
5            Ok((amount, _, _)) => println!("Wrote {} bytes",
6 amount),
7            Err(e) => println!("Error: {}", e),
8        }
9        Ok(())
10   });
11
     tokio::spawn(msg);
```

Snippet 1-3: 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.

```
1    let (lines_tx, lines_rx) =
2  LinesCodec::new().framed(socket).split();
3
4        let responses = lines_rx.map(move |incomming_message| {
5            return incomming_message;
6        });
```

Snippet 1-4: 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 [10]. 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 [11] 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.

```rust
1  impl ThreadPool {
2      pub fn new(size: usize) -> ThreadPool {
3          assert!(size > 0);
4
5          let (sender, receiver) = mpsc::channel();
6
7          let receiver = Arc::new(Mutex::new(receiver));
8
9          let mut workers = Vec::with_capacity(size);
10
11         for id in 0..size {
12             // create some threads and store them in the vector
13             workers.push(Worker::new(id, Arc::clone(&receiver)));
14         }
15
16         ThreadPool { workers, sender }
17     }
```

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

Snippet 1-5: The Thread Pool 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).

```rust
1 fn main() {
2     let addr = "127.0.0.1:6142".parse().unwrap();
3     let listener = TcpListener::bind(&addr).unwrap();
4
5     let counter = Arc::new(Mutex::new(0));
6
7     ...
8 }
```

Snippet 0-6: Adding the counter variable wrapped in a Mutex

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

Snippet 1-7: Cloning the Mutex for each connection socket so as to have and keep track of multiple "owners" for one Rust variable

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

Snippet 1-8: 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).

2.2. 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 [12] 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" [13] where he documents his journey into developing a key-value storage system using C++ and Hash Tables, 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 Hash Tables 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 is explained on Chapter 1: Related Work) 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.

2.3. Implementing functions for database access

```
Command::Remove(parameters) => {
    let key = parameters.unwrap(); //Can safely do this as fisrt match is Error
    if key.len() == 1 {
        let mut db = database_arc.lock().unwrap();
        let result = (*db).remove(&(key[0])).unwrap(); // Error check for n/a val
        match result {
            Data::Value(val) => {
                return format!("Removed value: {}\n", val);
            }
            Data::Map(_) => {
                return format!("Removed directory under: {}\n", &(key[0]));
            }
```

Snippet 2.1: Non-modularised way of accessing the database

After coming back from the holidays, I started working on my database again. So far, I had a basic in-memory only database that could receive commands and store Strings (i.e. Text). At this point, the server was the one reading and writing directly to the database by getting the database lock, without passing through any intermediaries (see snippet above). I did this to quickly test (similarly to an artist sketching) if what I wanted to code, indeed worked. Since it did, I decided it was time to put the code that interacted with the database into its own functions.

2.4. Creating database interaction functions

While I was working on modularising my database access functions, all was going well until I started to work on the "setvalue" function. This caused a big setback for me.

I was modifying it so that nested BTreeMaps (the data structure that forms the core of the key-value database) could be added to the database. My idea for implementing this was to receive a list of the nested BTreeMaps and loop through them. I would have a pointer that would keep track of the current BTreeMap. If the map existed, then I would change the pointer to it. Otherwise, I would create a new BTreeMap in the data structure that the pointer is currently pointing to (i.e. the parent map) and then I would change the pointer to the newly created map. Unfortunately, for about a week I was unable to figure out how to get the code to work as I was running into multiple Rust-specific issues regarding variable mutability and borrowing. I didn't realise until I solved it, that one would have to borrow the reference to the "main" database as mutable and do the same procedure when getting the nested Maps from the database.

But mutable references have one big restriction: you can have only one mutable reference to a particular piece of data in a particular scope. This code will fail:

Filename: src/main.rs

```rust
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```
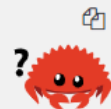
Here's the error:

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:14
  |
4 |     let r1 = &mut s;
  |              ------ first mutable borrow occurs here
5 |     let r2 = &mut s;
  |              ^^^^^^ second mutable borrow occurs here
6 |
7 |     println!("{}, {}", r1, r2);
  |                        -- first borrow later used here
```

This restriction allows for mutation but in a very controlled fashion. It's something that new Rustaceans struggle with, because most languages let you mutate whenever you'd like.

The benefit of having this restriction is that Rust can prevent data races at compile time. A *data race* is similar to a race condition and happens when these three behaviors occur:

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There's no mechanism being used to synchronize access to the data.

Screenshot 2.1: Online Rust book explaining mutability in the context of data races

At one point, I was trying to borrow as mutable the same object twice. This is not allowed in Rust as it's part of its data race prevention issues (see screenshot above, taken from the Rust online book [14].) As with most mistakes, in retrospect the answer seems simple, but it took me a lot of research, trial and error to come across a solution. As I was about to seek help from my supervisor, I stumbled upon an example of the "Entry" API for Maps in Rust (Screenshot 2.2) which is a design pattern made specifically for conditions like mine i.e. if a data entry exists in the map (Occupied) do something, if it doesn't (Vacant) do something else.

Thus, I finally solved my issue while learning a lot about pointers, borrowing and mutability in Rust. I met with my supervisor and moved on to the next stage of development.

Enum std::collections::btree_map::Entry

```
[-]
    pub enum Entry<'a, K, V>
    where
        K: 'a,
        V: 'a,
    {
        Vacant(VacantEntry<'a, K, V>),
        Occupied(OccupiedEntry<'a, K, V>),
    }
```

[-] A view into a single entry in a map, which may either be vacant or occupied.

This enum is constructed from the entry method on BTreeMap.
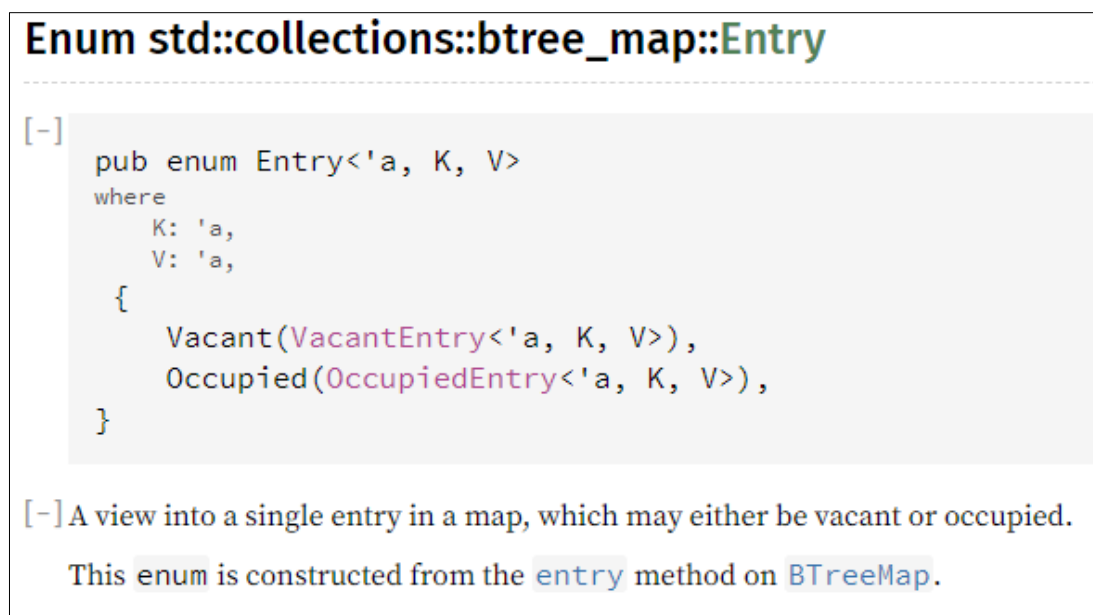
Screenshot 2.2: Entry API in Rust

2.5. Adding logging functionality and database storage to disk

The next stage of development was to add logging functionality to the database. This essentially would allow tracking of changes, crash recovery and in disk storage, making the database durable (D in the ACID acronym, see Related Work Part II). The latter is done by reading the log (which is a text file) when the database boots and restoring the database to the state it was at the end of the log (if the log exists, otherwise a blank file is created.) Crash recovery is implemented as writing to the log is done before modifying the database. It is significantly quicker to write to a file which makes it unlikely in the event of a crash that a command was not logged, even though the database may not have been able to execute it (later on I would test this, see the Testing section).

A simple version of this was easy to implement as every time the user typed in a command, if it was a valid command, the string that was typed would get written to the log before the appropriate

database command function got called.

Then, when the database booted up, it would replay the log file. Effectively simulating a user typing in the commands all over again until the database was restored to the state it was at before it shut down. This has obvious drawbacks, however.

The first is that one is writing everything to the log file even though the only commands that affect the state of the database are the remove and set-value commands, this issue I quickly fix later on.

The second is that the database executes commands that don't affect the end state of the database after booting e.g. in the log, a user added a key-value pair which is removed later on. It would be ideal to implement an optimisation/compression algorithm that removes "useless" operations when restoring the database from the log as they are wasted time and resources with the previously mentioned implementation.

2.6. Extracting the database functionality into its own module

At this point, I had all my code in one file much like it is common to find when following procedural programming practices. I decided it was time to clean it up and use Object-Oriented principles by employing the "struct" and "impl" functionality of Rust. Essentially, the aim would be to make a database structure (i.e. object) and then use the "impl" keyword which in Rust allows one to *implement* functions that access this structure and can only be called from external code. These functions can only be accessed through an object instance and only when the functions are explicitly public (all struct data and functions are private by default unless explicitly stated by the "pub" keyword.)

The database object holds a reference to the database lock and to the log file lock. The latter is needed so that the log is not opened for reading every time a user types in a command. It would be opened once when the database boots and stay open while the database is operating. Log writing should be handled by the database's own mutator functions i.e. the remove and set function only, and not by some external entity (which previously was the string parser).

```rust
pub struct Database {
    database_arc: Arc<RwLock<DBSignature>>,
    log_file_arc: Arc<RwLock<File>>,
}

impl Database {

    pub fn new() -> Database{
        let mut db = Database{
            database_arc: Arc::new(RwLock::new(DBSignature::new())),
            log_file_arc: Arc::new(RwLock::new(OpenOptions::new()
                .read(true)
                .append(true)
                .create(true)
                .open(LOG_FILE)
                .unwrap())),
        };
        db.restore_from_log();
        db
    }
```

Snippet 2.2: Part of the database.rs file where the struct and the constructor are showed

In Snippet 2.2 above, one can see how the constructor for a database object works and how the database calls its restore function to restore itself from the log file when initialised. When I got to the point that I was fairly satisfied with the extraction of the database into its own file ("database.rs" which is then imported by the server file) I decided that it was time to implement proper, automated testing.

Chapter 3: Testing & Benchmarking

So far, I have been testing my database server using the windows Telnet client, manually. This is not good practice. In general, one wants testing to be as automated as possible, ideally removing all variability from the testing conditions (which is not possible in practice, but it is the pursuit of it that matters.) The first step to automate testing was to write a client that would connect to the database, send it commands and then would check that the server's responses are what is expected. For this, I had to research more about the Tokio library functionality [15], specifically the TcpStream object which is used to connect to network sockets. During this stage, noticed that it would benefit me to use Tokio 0.2 instead of the version I was using (0.1), this is because the documentation was clearer and code examples all used 0.2. I had used 0.1 in the first place because the first tutorial I followed to create a Tokio "echo" server used version 0.1 and didn't find out until later on that there was a newer version. Because of this change I had to re-write a lot of how the server handled connections, so I used this opportunity to also lay the groundwork for parsing transactions. The major changes were that I was now manually reading the user's commands into a buffer (8 byte buffer) and then interpreting this buffer according to certain rules: if a character that was received is a semicolon ";", that would mean the end of a command or *chain* of commands, just like in SQL. Then, I would separate the input (and any previous input that was read that did not contain the semicolon character) by newline characters to signify different commands. Lastly, I would parse these commands into database command objects as I have been doing from the start. In contrast to my previous version, I stopped using the library that allowed me to split the socket input automatically by newlines as this was not useful to me anymore. It is also deprecated in Tokio 0.2.

After re-writing my server, I implemented my client using the new 0.2 Tokio nomenclature using "async" and "await" keywords for asynchronous programming which were introduced into Rust in 2018 but only became stable towards the start of 2019. I also implemented assertions into my code for testing, similarly to how I was taught JUnit for Java.

```
#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    // Allow passing an address to listen on as the first argument of this
    // program, but otherwise we'll just set up our TCP listener on
    // 127.0.0.1:8080 for connections.
    let addr = env::args()
        .nth(1)
        .unwrap_or_else(|| "127.0.0.1:6142".to_string());

    // Next up we create a TCP listener which will listen for incoming
    // connections. This TCP listener is bound to the address we determined
    // above and must be associated with an event loop.
    let mut socket = TcpStream::connect(&addr).await?;

    socket.write_all("set Name Sephiroth;\n".as_bytes()).await.expect("failed to write data to socket");

    let response: String = read_from_socket(&mut socket).await;
    assert_eq!(true, response.contains("Ok"));

    socket.write_all("get Name;\n".as_bytes()).await.expect("failed to read data from socket");
    let response: String = read_from_socket(&mut socket).await;
    assert_eq!(true, response.contains("Sephiroth"));

    socket.write_all("remove Name;\n".as_bytes()).await.expect("failed to write data to socket");
    let response: String = read_from_socket(&mut socket).await;
    assert_eq!(true, response.contains("Sephiroth"));

    socket.write_all("get Name;\n".as_bytes()).await.expect("failed to read data from socket");
    let response: String = read_from_socket(&mut socket).await;
    assert_eq!(true, (response.contains("not found") && response.contains("Name")));


    Ok(())
}
```

Snippet 2.3: First version of the database testing client

3.1. Automating Testing

After I coded the testing client, I met with my supervisor. He suggested I should automate my testing even further before moving on with anything else. Therefore, what I decided to work on next is to have a Rust file that, in a script-like fashion, starts a database server then starts a client server and has them communicate, all by running a single command. In the future I could have this script spawn multiple threads of clients to test concurrency. I could also test reliability by having the script "kill" one of the servers as a client has sent a command. This would save me time as before this, I manually started a server and client separately, using the command line.

The first idea I had for implementing the script came from my experience with Java. I would create a main Rust file that would call the server and client start functions. To do this, I first renamed the server and client "main" functions to "start_server" and "start_client" respectively. I then renamed both Rust files which were "main.rs" (as I was using them as independent applications) to "server.rs" and "client.rs". What this achieved was that I could now put both the server and client files into the

same directory as the main Rust file I was going to use to start the server and client. Now, I could import the server and client files into my main file using the "mod" keyword (Snippet 2.4). This allowed me to call the server and client start functions from my main file.

```rust
use tokio::runtime::*;

mod server;
mod client;

fn main() {
    // build runtime
    // let mut threaded_rt = runtime::Builder::new()
    // .threaded_scheduler()
    // .thread_name("my-thread-pool")
    // .build()
    // .unwrap();

    let server_rt = Runtime::new().unwrap();
    let client_rt = Runtime::new().unwrap();

    // use runtime ...
    // threaded_rt.spawn(server::start_server());
    server::start_server(server_rt);
    // threaded_rt.spawn(client::start_client());
    client::start_client(client_rt);

    return
}
```

"mod" keyword

Using the "Builder" (commented out)

Separate Tokio Runtimes

Snippet 2.4: Creating separate runtimes and using the "mod" keyword

Initially, I wanted to have the server and client start on different threads. After doing some research I thought that using the Tokio "Builder" [16] would work. The Builder creates a Tokio runtime (which is the main Tokio thread where Tokio functions must run on for the asynchronous operations to work properly.) I ran into a problem though, when calling an async function with Tokio, one has to use the await keyword to get the "Future" for that function which is an object representing the result a function will yield when it completes in the future. There are a couple of these functions in the server (specifically, ones that deal with client connection over the network) and they return a "Result" which is either an "Ok(result)" or an "Err(error)". If the function returns an error, the server stops because there was a connection problem. If this error is returned to the calling function, it will create a compiler error because the Error type that Tokio uses cannot be sent "safely" between threads (Snippet 2.5). I found out about this when reading a Rust language blog post about the subject. [17] To overcome this issue, I had to catch the error and re-wrap it in an Error type that is standard to the Rust library. (Snippet 2.6)

Snippet 2.5: Error cannot be sent because it does not implement "Send"

```rust
// pub async fn start_server() -> String {
pub async fn start_server(mut tokio_runtime: Runtime) -> std::result::Result<(), std::boxed::Box<std::io::Error>> {

    // let addr = env::args()
    //     .nth(1)
    //     .unwrap_or_else(|| "127.0.0.1:6142".to_string());

    tokio_runtime.block_on(async {

        let addr = "127.0.0.1:6142".to_string();

        // let mut listener = TcpListener::bind(&addr).await?;
        let mut listener;
        match TcpListener::bind(&addr).await{
            Ok(tcp_listener)=>{
                listener = tcp_listener;
            }
            Err(error)=>{
                return Err(Box::new(error));
                // return box.into_raw();
                // return "Error for server binding to address".to_string();
            }
        }
    }
}
```

Using "block_on"

Re-wrapping the error

Snippet 2.6: Runtime Block and Error re-wrapping

Unfortunately, when I implemented this fix, I ran into another obstacle: I could not run the server and client functions in the way I envisioned because they have to run on the main Tokio "Runtime" thread, which is the one created in the main function. (Snippet 2.7)

Snippet 2.7: Error for calling server function outside of the main thread

To fix this, I tried creating two separate Tokio runtimes (Snippet 2.4) and passing them separately to the server and client functions and creating the Network connections (TcpListener for the server, TcpStream for the client) on those specific threads using the "block_on" function (Snippet 2.6). It compiled and ran, but it seemed like nothing happened i.e. the process exited, and the command prompt was blank. Therefore, the print statements were not working. As it seemed I was getting nowhere with these methods, I started to think about alternatives.

```
#[tokio::main]
async fn main() {
    let server_handle = tokio::spawn(server::main());
    let client_handle = tokio::spawn(client::main());

    // let server_result = server_handle.await.expect("Server handle panicked.");
    let client_result = client_handle.await.expect("Client handle panicked.");
    match client_result{
        Ok(()) => {
            println!("All tests completed successfully.");
        },
        _ => {
            println!("Some error has occurred!");
        }
    }

}
```

Snippet 2.8: Using async and tokio::spawn

After taking a day off, the morning after I had an idea that proved to be successful. When looking at the Tokio documentation, I stumbled upon the documentation of the spawn function [18]. It allows to create a Tokio specific task that runs asynchronously. I realised I never tried using it for the purposes of running the server and client separately. In retrospective, it was the simplest and most "obvious" of solutions and while it took me a few days to find it, it worked! (Snippet 2.8 and 2.9)

```
Listening on: 127.0.0.1:6142
[115, 101, 116, 32, 78, 97, 109, 101, 32, 82, 117, 115, 116, 97, 99, 101, 97, 110, 59, 10]
Before filtering buffer: set Name Rustacean;

Parsing command: "set Name Rustacean"
[103, 101, 116, 32, 78, 97, 109, 101, 59, 10]
Before filtering buffer: get Name;

Parsing command: "get Name"
[114, 101, 109, 111, 118, 101, 32, 78, 97, 109, 101, 59, 10]
Before filtering buffer: remove Name;

Parsing command: "remove Name"
[103, 101, 116, 32, 78, 97, 109, 101, 59, 10]
Before filtering buffer: get Name;

Parsing command: "get Name"
All tests completed successfully.
```

Snippet 2.9: Successful testing of server through client with one function

Note: the number lists are the unsigned 8-byte representations of the input the server is receiving. This is for de-bugging

purposes.

While looking into my next step of testing which is benchmarking, I came across the Builder documentation again. I decided to attempt to make it work again as I was frustrated that I didn't understand how it worked. Finally, after tinkering and reading the documentation, I go it to work! I figured it out thanks to all the previous errors I had come across when designing this testing script. It works like the following: Tokio functions (specifically the ones that perform asynchronous tasks) need to run in what is called the "Runtime", otherwise they cannot use the proper functions for I/O (input and output) and non-blocking processing. Once I realised this, in the documentation I noticed how a "block_on" function was used with a non-Builder Runtime and, *inside* of the "block_on" function, the previously mentioned "spawn" function was used. Effectively calling the asynchronous tasks within the context of the Runtime. Once I understood this, I applied the same principle to a Builder runtime with a multi-threaded configuration and it worked! (Snippet 2.10)

```rust
use tokio::runtime::Builder;

mod server;
mod client;

fn main() -> std::result::Result<(), std::boxed::Box<std::io::Error>> {

    // build runtime
    let mut rt = Builder::new()
    .threaded_scheduler()
    .enable_all()
    .build()
    .unwrap();

    rt.block_on(async {
        let _server_handle = tokio::spawn(server::main());
        let client_handle = tokio::spawn(client::main());

        // let server_result = server_handle.await.expect("Server handle panicked.");
        let client_result = client_handle.await.expect("Client handle panicked.");
        match client_result{
            Ok(()) => {
                println!("All tests completed successfully.");
                Ok(())
            },
            Err(error) => {
                println!("Some error has occurred!");
                Err(error)
            }
        }
    })
}
```

Snippet 2.10: Working Builder-Runtime configuration

● ● ●

3.2. Benchmarking

After completing this concurrent testing function, I decided I should start working on benchmarking. This is because you cannot prove database efficiency and advantages without running benchmarks.

For benchmarking, I used the "time" Rust library (Snippet 2.11.) My first two tests were to run the clients with logging to file enabled on the server, then with logging to file turned off. I ran each test 10 times in a loop, taking the average time at the end, this is to eliminate a bit of the variability there can be between tests (because of CPU load on the computer due to different programs running in the background, memory access time differences etc.)
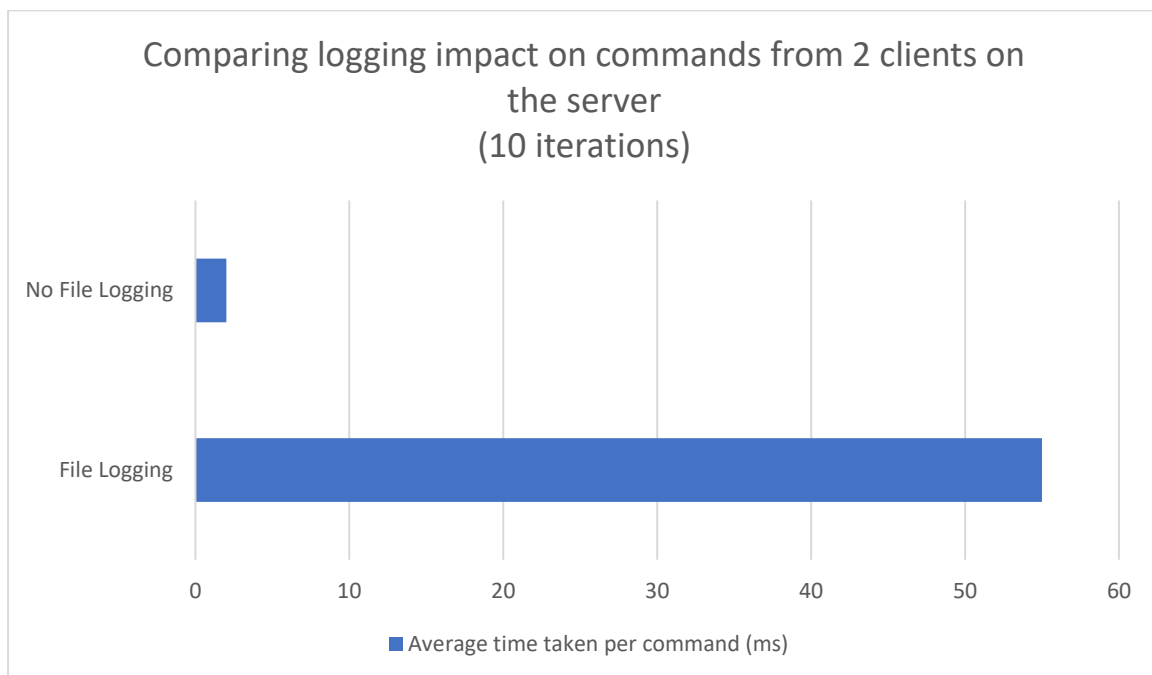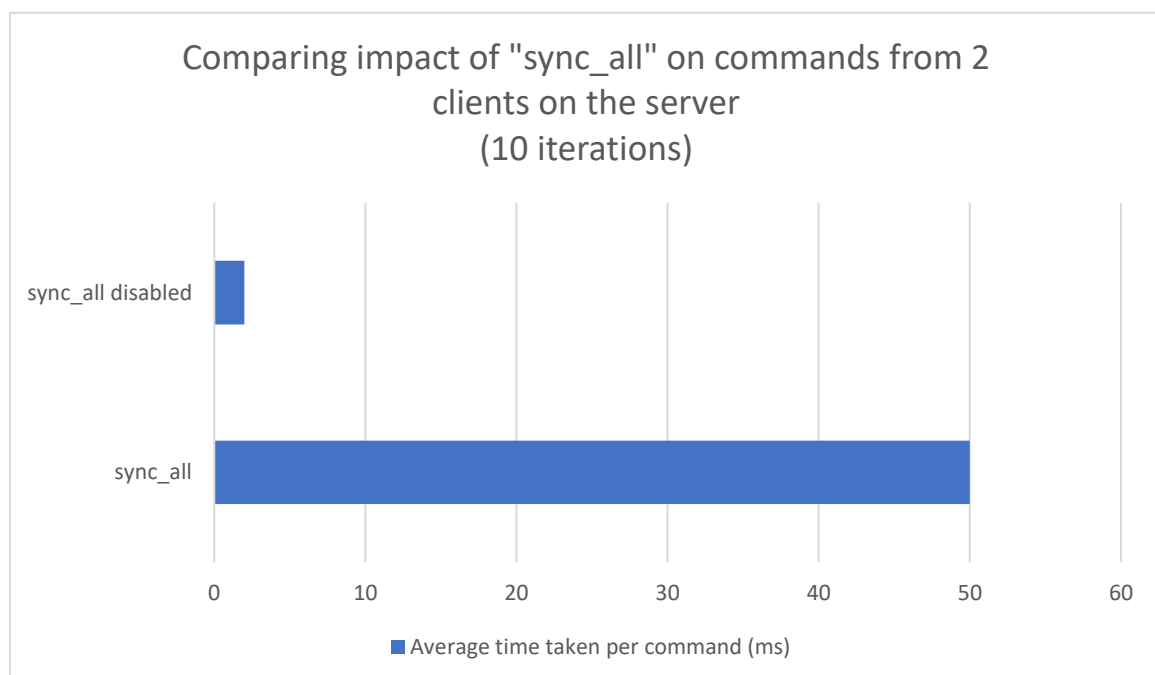

We can see how writing to a log text file after each command affects runtime greatly, in this case it took 27.5 times more time to complete the tests with text file writing operations. This makes sense as it is very costly to perform writing which is then saved to disk. Especially, what really causes delay is when I am making sure that writing is always saved to the disk using the "sync_all" function, this is for database durability purposes -see ACID under the Related Work section-. This means that the database is telling the OS to immediately write information to the hard disk and not wait until the writing buffer is full (it is designed this way by default so the OS can write with more efficient resource usage.) On this note, my next benchmark would be to turn off this function to see how much forcing hard disk writes after each command affects performance.

```
let mut sum_of_times = 0;

for i in 0..10{

    let start_time = Instant::now();

    socket.write_all("set Name Rustacean;\n".as_bytes()).await.expect("failed tso

    let response: String = read_from_socket(&mut socket).await;
    assert_eq!(true, response.contains("Ok"));

    socket.write_all("get Name;\n".as_bytes()).await.expect("failed to read data
    let response: String = read_from_socket(&mut socket).await;
    !(true, response.contains("Rustacean"));

    ite_all("remove Name;\n".as_bytes()).await.expect("failed to write d
    let response: String = read_from_socket(&mut socket).await;
    assert_eq!(true, response.contains("Rustacean"));

    socket.write_all("get Name;\n".as_bytes()).await.expect("failed to read data
    let response: String = read_from_socket(&mut socket).await;
    assert_eq!(true, (response.contains("not found") && response.co

    let end_time = Instant::now();

    println!("Loop {} time {:?}", i, end_time.duration_since(start_time).as_milli

    sum_of_times += end_time.duration_since(start_time).as_millis();
}

println!("Client report: Time taken to complete tests is {}", (sum_of_times/10));
```

Saving tests start and end time

Getting the sum and then dividing it by number of tests done

Snippet 2.11: Client tests with completion time

Using the "sync_all" all function to force disk writing and then turning it off I got the following results:

Comparing impact of "sync_all" on commands from 2 clients on the server
(10 iterations)

sync_all disabled	▮
sync_all	▬▬▬▬▬▬▬▬▬▬▬

0   10   20   30   40   50   60

■ Average time taken per command (ms)

The data gathered allows us to draw the following conclusions: writing data to disk is costly, but at the moment it seems that my tests do not have enough commands sent to the server for the OS to write the commands to disk when it isn't forced to. This is because the time without forcing disk-writing is the same as not writing to the disk at all, because the time result is the same as when I turned off disk-logging (see the second paragraph of this section). This means I need more commands to fill the buffer so that the OS deems it fit to write what's in the buffer into the text file. On the latter, I will address this issue next by devising a test that sends a significant number of commands to the server.

Thanks to the Rand crate, which is a "A Rust library for random number generation." [19] I created a test that generates random alphanumeric Keys and Values and sends these commands to the server in a loop whose iteration number is passed into the testing function as an argument. This allows me to test any number of functions being sent back to back to the server, in this case I simply used "set" commands. (Snippet 2.12)

```rust
for _i in 0..number{

    let first_time = Instant::now();

    let range = Uniform::new(5, 100);

    let x = thread_rng().sample(range);
    let key: String = thread_rng().sample_iter(Alphanumeric).take(x).collect();
    let x = thread_rng().sample(range);
    let value: String = thread_rng().sample_iter(Alphanumeric).take(x).collect();

    let command = format!("set {} {};\n", key, value);


    socket.write_all(command.as_bytes()).await.expect("failed to write data to socket");
    let _response: String = read_from_socket(&mut socket).await;

    let second_time = Instant::now();
    let time_taken = second_time.duration_since(first_time).as_millis();
    // println!("Iteration {}, time taken {}ms", i, time_taken);
    sum_of_times += time_taken;
}
```

Snippet 2.12: Loop for generating random key-value "set" commands and sending them to the server

When I first ran this function with 100 iterations i.e. 100 set commands being sent to the server, I had the "sync_all" function on. I got the following result:

```
Client report: Time taken to complete the average test was 27ms. Total time taken 2s
All tests completed successfully.
```

*Snippet 2.13: 100 set commands to server test report*

When I turned off the "sync_all" function, I got 0ms taken for the average test and 0 seconds overall. Therefore, I thought I needed even more iterations to be able to measure how long *not* using "sync_all" was taking, I tried 100 thousand iterations.

```
Listening on: 127.0.0.1:6142
Client report: Time taken to complete the average test was 0ms. Total time taken 73s
All tests completed successfully.
```

Snippet 2.14: 100,000 set commands without "sync_all"

```
         Running   target\debug\database-server.exe
Listening on: 127.0.0.1:6142
Client report: Time taken to complete the average test was 25ms. Total time taken 2593s
All tests completed successfully.
```

Snippet 2.15: 100,000 iterations with "sync_all"

Using "sync_all", the average test (i.e. set command) took 25ms to complete and it took almost 40 minutes to complete all 100 thousand commands! In contrast, *not* using "sync_all" only took 73 seconds to perform all 100 thousand set commands. This is a significant cost in performance which questions if "sync_all" should really be used or if alternatives are available. This is why I decided to look for alternatives. In the Rust library, one can also find the "sync_data" function which in contrast to "sync_all", only synchronizes content to the disk and not the file metadata (timestamps, ownership data etc.)

In the documentation it says, "The goal of this method is to reduce disk operations." [20] When I tested 1000 set commands, I got the following results:

Comparing "sync_all" and "sync_data" on commands from 2 clients on the server (1000 iterations)
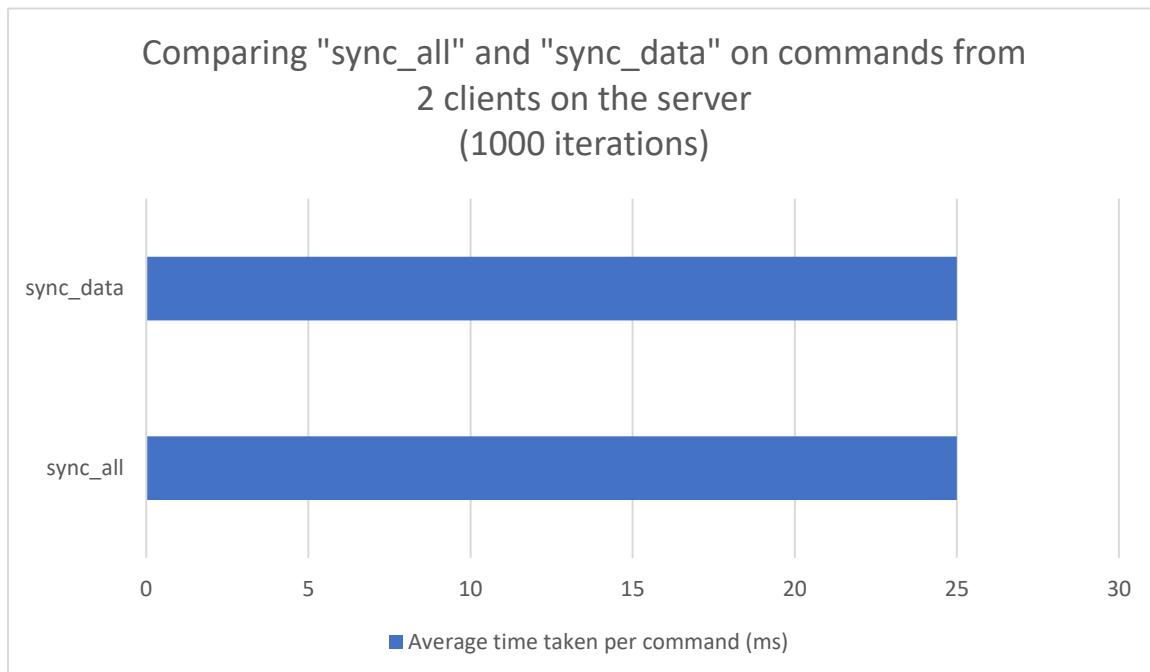
This shows us that unfortunately, there isn't a difference between using either command. I thought this was odd considering the purpose of having two different functions is to allow flexibility with regards to performance. Therefore, I had a look at the documentation again and found something I missed the first time I read it: it says that some platforms may implement "sync_data" in the same way as "sync_all" and it seems that Windows is one of those platforms, as I have done my testing on Windows thus far. This explains why there isn't a difference between test times.

I was frustrated that there wasn't a better way to guarantee that the log file was written to when logging commands. For that reason, I looked further into the documentation of the "File" and "io" libraries which are the ones used for file writing. After a bit of reading I found that there is another function that could fit my purposes: the "flush" function. In the documentation of "flush" it states: "Flush this output stream, ensuring that all intermediately buffered contents reach their destination." [21] When I replaced "sync_all" with "flush" on the previously tested 1000 iterations of set commands, I got a total time of 0 seconds! This is the same as not using "sync_all" at all. This result had me question whether the function was working properly, but upon editing my code to show me the result of calling "flush" it returned "Ok(())" every time, and since it is not returning any Errors I assume that the function is working as it expects to.

After discussing the testing I had done with my supervisor, we theorised that the difference between flush and sync_all is that flush sends the data to the OS buffer which in this case is Windows. It lets

Windows' C code handle when to write to the file, effectively making it so that the Rust program no longer has to worry about the data. In contrast, sync_all tell the OS code to make sure the data is written to disk and waits for a response. It will not let the program continue until it has received confirmation from the OS that the data has been written to disk.

In light of this line of thought, the database must use sync_all to make sure it doesn't lose any data. Clearly, it comes at an expense: making sure the data is written to disk costs a lot time. Therefore, there might be a balance that can be struck between how many operations are made and when the sync_all functions is called, at the cost of possibly losing data between sync_all calls. It is a classical trade-off of performance versus reliability.

3.2.1. Testing "sync_all" on an SSD

Another element that may be increasing the time it takes to write to disk is writing to a hard disk drive (HDD) instead of a solid-state drive (SSD). I decided it would be good to test the difference between HDD writing and SSD writing, therefore I ran the test that sends set commands to the server. I ran it for 1000 commands because, empirically, it seemed to not take too long but took long enough that there was a difference that could be seen in the data:
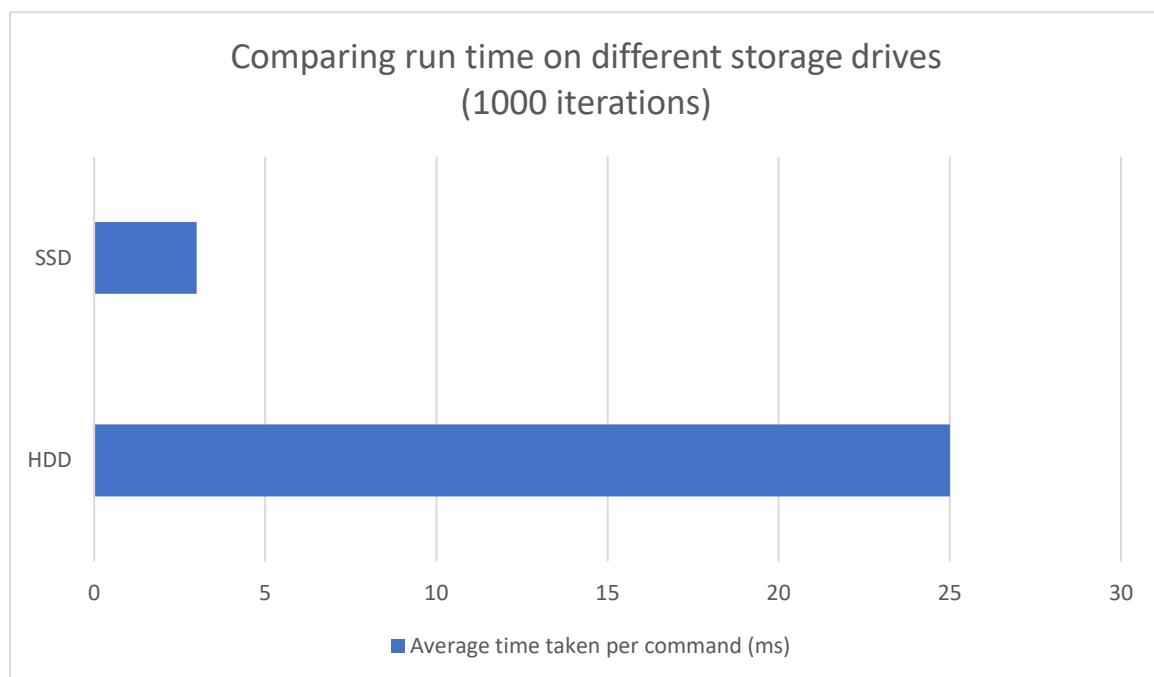
We can see how big of a difference writing to an HDD compared to an SSD makes: running the server on an SSD was 6.25 times faster than running it on an HDD! If we recall on one of the earlier tests, when 100 thousand iterations were run on the HDD with "sync_all" it took almost 40 minutes. Had I done that test on an SDD it would have taken approximately 6 and a half minutes!

Thanks to this we can conclude that preferentially one wants to run the server on an SDD to be able

to mitigate the slowdowns that come from having to use "sync_all" for data durability and server reliability.

3.3. Testing on a real network

So far, I have been testing my server with my client on the same machine using the internal network of the machine, also called the "localhost". I did this by having my server run on the IP 127.0.0.1 which points to the local machine network. My client would connect to this same IP, effectively running both without the need for two different computers on a network. I decided it was time to test this on a real network using two machines. Some of the options for this would have been to setup my machine on a cloud server such as AWS, Google Cloud or Amazon Azure. After discussing about it with my supervisor he suggested the easiest way to test this would be to use the University's machines in the computer labs.

To do this, I remotely accessed (through SSH) two of the university computers, I ran the Unix command "ifconfig" to be able to know what the IP of my server computer is. Then I ran the server on that computer and on the other I told the client to connect to the IP of the server computer. (Snippets 2.16 & 2.17) I had the client configured to run a simple test suite where it tests the "set", "get" and "remove" commands. It worked, which means that there aren't any issues running on a "real" network.

Snippet 2.16: Running the server on a remote University computer

Snippet 2.17: Running the client on a remote University computer, having it connect to a server on another machine

3.3.1. Testing multiple clients at the same time

The next stage of testing was to test multiple clients connecting at the same time to see if the database would behave in a predicted manner. At the moment, the database is designed so that only one user can modify it i.e. setting and removing. However, multiple users can read i.e. getting values and keys from the database even when another might be writing to the database.

To design this test I thought of using Tokio's asynchronous capabilities to run multiple clients at

once, in the same fashion as how I previously ran the client and server using one file i.e. before I tested them over a network with separate computers.


```
jarbe@DESKTOP-RJJ29I8 MINGW64 /d/Uni Rust/Database - Client (master)
$ cargo run
   Compiling database-client v0.1.0 (D:\Uni Rust\Database - Client)
    Finished dev [unoptimized + debuginfo] target(s) in 1.22s
     Running `target\debug\database-client.exe`
Client report: Time taken to complete tests is 125
Client report: Time taken to complete the average test was 33ms. Total time taken 3s
All tests completed successfully.
All tests completed successfully.
```

Snippet 2.18: Testing two clients connecting and modifying the database


```
SET  Name Rustacean
SET  KDNmArmwOWoMiOpscvIyLi7FleT1QQmyURv9yJTTsKgqjLdeFPuhEZQUqpznU 52imMRHAGX7KsvmpsYQy
REM Name
SET  uZ2s6IDdht5iG12CVIBjVhC8KhqgXn5lhpc LeHoPXMIjjzVYdADyCTJ3E8wHDNHZXIs0YNXZWls5xSrmao2
SET  Name Rustacean
```

Snippet 2.19: Log from two simultaneous clients test

While running a test where two clients connect at the same time and one runs set and remove commands while the other just runs many random sets we can see from the output (Snippet 2.18), that the second client which ran the same test as Snippet 2.13 took a bit longer to complete this time around. We can safely conclude that this was due to the first client (which runs 4 commands) as in the log we can see that the commands from both clients are staggered (Snippet 2.19). Thankfully, this is the behaviour we expect and shows that the Database can handle multiple clients connecting and interacting with it at once.

The next test I designed was for two clients to change the same key and see what would happen:


```rust
async fn change_name(mut socket: &mut TcpStream, key: &str, value: &str){
    for _i in 0..10{
        let command = format!("set {} {};\n", key, value);
        socket.write_all(command.as_bytes()).await.expect("failed to write data to socket");
        let _response: String = read_from_socket(&mut socket).await;
        let command = "get Name;\n";
        socket.write_all(command.as_bytes()).await.expect("failed to write data to socket");
        let response: String = read_from_socket(&mut socket).await;
        assert_eq!(true, response.contains(value));
    }
}
```

Snippet 2.20: Change name test

In this test, the first client sets the "Name" key to Bob and the second to "Alice". I made sure for the client to get the value after modifying it and check if the value was still what it was set to by that client.

Thanks to the previous (Snippet 2.18) test I thought what might happen was that since the server was processing requests in a staggered way, it might receive both sets first (from the first and second client respectively) and then both gets i.e. "set Name Alice; set Name Bob; get Name; get Name" and therefore that the tests would not succeed as the assertion in the last line of the test would be false (Snippet 2.20). I was wrong though! The test was passing correctly (Snippet 2.21). So, to double check these results I had the server print the commands it was processing. It was indeed as I suspected: the asynchronous engine of the server is still staggering the commands, but it is receiving both the set and get from one client then it processes the other client's set and get (Snippet 2.22).

Snippet 2.21: Successful test of both clients modifying the same key

Snippet 2.22: Staggered set and get commands

I decided to modify the test scenario to simulate a network packet getting delayed between the setting and getting functions, effectively making it so that the get from each client would be received by the server slightly later and therefore, during that delay the other client's set would arrive first e.g. "set Name Alice; set Name Bob; get Name" where the last get expected Alice to be the name but the server tells it the name is Bob. I managed to do this by using the "thread" Rust library to put the thread to sleep before it sends the get request to the server. The expected behaviour occurred (Snippet 2.23).

```
Value: Bob, response: Alice


thread 'tokio-runtime-worker' panicked at 'assertion failed: `(left == right)`
  left: `true`,
 right: `false`', src\client.rs:143:9
```

Snippet 2.23: Artificially simulating network delay causes the client to get a value it didn't expect

The next thing I tried, was killing the server randomly while the client is writing to it. This would be to verify that the sync_all function is doing its job by making sure the log is written to, before the server crashes so that data loss is minimised. First, I did this manually by using one client connecting to the server and killing the server process while the client was running. I tried this multiple times and verified that the last command that the client managed to send was the last command that was written to the server log (Snippets 2.24 & 2.25).

```
"set GLrcNLt TwawIVz6;\n"
thread 'tokio-runtime-worker' panicked at 'failed to read data from socket:
```

Snippet 2.24: Last command from client on one of my manual tests

```
25    SET  GLrcNLt TwawIVz6
26
```

Snippet 2.25: Last command written to the server log

In the above test we can see that both, the client command and the last command written to the log match. I did this multiple times and both commands always matched. Therefore, I came up with a situation where it would be likely that the server would not manage to write the commands to the log before it crashed: by having multiple clients connect and send commands to the server at the same time, it is likely that the server might crash while one of the commands from one of the clients is queued for processing and therefore not written to the log. I put this to the test by having two clients send commands to the server simultaneously on two different threads. What I described is indeed what happened. (Snippets 2.26 and 2.27)

```
"set YSC6vJY 6Lw5Oj4Y5;\n"
"set XbsK3cDc FO3kNJZG;\n"
thread 'tokio-runtime-worker' panicked at 'failed to read data from socket:
```

Snippet 2.26: Output from two clients before server crash

Snippet 2.27: Server log file after crashing

As we can see above, before the server crashed, both clients sent two commands to the server but in the log file of the server, the last command written before the crash was the one from the first client: the second client's command never got written and therefore the information was lost.

After a discussion with my supervisor, he raised an interesting point: my server sends receipts as confirmation that a command was successfully processed e.g. for setting values the server replies with "Ok". Therefore, if the client was not receiving an "Ok" message back when their commands didn't get processed because the server crashed, then the client would know that they would have to send the command again. If, however, the client got an "Ok" back before the server had written their command to the log file, we would have a problem. To test that idea, I modified the previous test so that the client would print the server's response to the console, that way we could see what the last confirmation receipt received was on top of the last command that was sent. (Snippets 2.28 & 2.27).
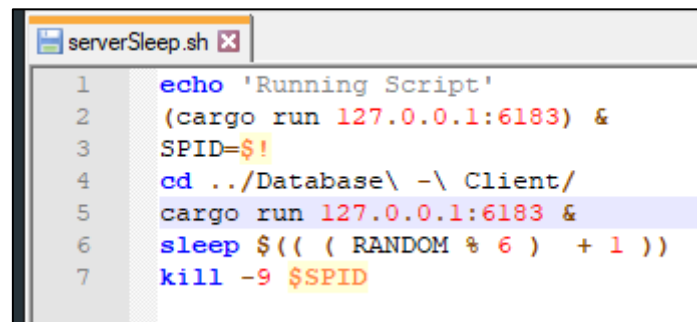

Snippet 2.28: Checking for Ok

Snippet 2.29: Last commands written to the server log (compare with Snippet 2.27 above)

In the above Snippets we can see that one of the clients did not receive an "Ok" back for the last command that it sent before the server crashed. This last command was indeed not written to the server log. Therefore, the server did not manage to process the command before it crashed, and the client is aware of this. This was a successful test as this is the behaviour we want.

My supervisor and I though a more automated way of testing these scenarios would be ideal. This is because I was manually killing the server process while the clients were sending commands to it.

This would have been very difficult to do in Rust as it would require learning OS level libraries to be able to run a server process (which would be a Rust process too) then having the OS kill this process after a certain amount of time. As my project deadline was near, I decided to use a bash script for this purpose (Snippet 2.30).

```
serverSleep.sh
1   echo 'Running Script'
2   (cargo run 127.0.0.1:6183) &
3   SPID=$!
4   cd ../Database\ -\ Client/
5   cargo run 127.0.0.1:6183 &
6   sleep $(( ( RANDOM % 6 )  + 1 ))
7   kill -9 $SPID
```

Snippet 2.30: Writing a bash script

The above Snippet shows a small bash script that is run in the folder where the Rust server is to be found. First, is prints a confirmation that the script has started running. Then, it runs the server using the Rust 'cargo' command (which compiles and runs the Rust project) and stores the ID of the process in a variable called 'SPID'. Then, it changes directory to the Rust Client folder and runs the same 'cargo' command, pointing the client to the address and port of the server. In the above example this is the local host server on port 6183. Then, it sleeps for a random amount of time and then kills the process using the process ID to effectively simulate the server crashing.

I suspected that after running this multiple times there might have been a test that "catches" the server crashing while still sending an "Ok" receipt to the client for a command that was not actually written to the log. In the many times I ran this script, it never happened. It means that, if there is such a bug, the probability of it occurring is likely to be very low. This is very good for proving the reliability of the database server.

Unfortunately, with the above script there was still a major issue: I still had to manually check that the last "Ok" received by the client corresponded with the last command written to the server's log file. Therefore, I had to automate my script even further. After researching about bash scripting I managed to create a script that takes the last line written to the server's log file and compares this to the last command that one of the clients sent to the database that received an "Ok" receipt from the server. If the commands matched, all is well and good. However, if the commands did not match, it would mean the server told the client it had processed a command when in reality it hadn't. I created this script in a separate file (Snippet 2.31) which is run at the end of the previous script (after the server is randomly killed). Lastly, I added a for loop so the test can be run back to back multiple times, and if at any point in time one of the tests fails, the whole loop immediately stops.

After checking that the failure check worked, I ran the test 10 times and it never failed (Snippet 2.32). From this we can assume that the database server is very robust when it comes to command handling and giving feedback to the client.


```
checkLines.sh
 1    LASTSERVER= tail -1 log.txt | cut -d' ' -f3
 2    cd ../Database\ -\ Client/
 3    PREVOK=0
 4    while IFS= read -r line
 5    do
 6        echo "line is: $line"
 7        if (($PREVOK == 1))
 8        then
 9          if [[ "$line" =~ .*"$LASTSERVER".* ]]
10          then
11            echo "Test Passed."
12            break
13          else
14            echo "Test Failed! Ok Received but wrong String!"
15            break
16          fi
17        else
18          if [[ "$line" =~ .*"Ok".* ]]
19          then
20            PREVOK=1
21          fi
22        fi
23    done < <(tac output.txt)
```

Snippet 2.31: Checking the last command written to log to the last command the server told the client it had processed

Snippet 2.32: Passing test for the script that checks for proper "Ok" command receipts

In the above Snippet, one can see a few lines before "Test Passed", the first is the key from the last command sent to the log, the fourth is the last command for which one of the clients received an "Ok" receipt for. These are the two outputs to compare and would be useful for debugging if at any point the test fails. The second and third are of no importance, they are just side-effects from how the script was written.

3.4. Testing & Benchmarking Conclusions

From the testing and benchmarking that was preformed, one can draw the following conclusions:

- The database's crash recovery mechanism seems to be working very well.
- The database properly handles multiple clients at the same time and does not relay false positives to the client when it crashes mid-command.
- For the database to be durable i.e. make sure data is not lost, there is a big performance cost because of the use of the "sync_all" function. This can be slightly mitigated by running the database on a Unix system to take advantage of the "sync_data" command. Lastly, running the database on an SSD or better, greatly benefits performance.

Summary & Conclusions

Summary

Over the course of this project, Rust skills were developed and used in a practical manner to code a key-value storage database that accepts basic commands. Testing was developed and benchmarking was done to compare different functionalities of the Rust library and different design decisions. Benchmarking on different hardware was also done, thanks to which insight was gained on what type of hardware would suit this database best.

Future Work

Many features can still be added to the database e.g. different protocols for accessing the server, like HTTP; more operations such as the way SQL handles counting and sorting when data is displayed.

However, there is one notable feature that I wished I was able to implement in time, which is storage of non-textual data. This works to the extent that the data passed in is does not contain a byte that can be interpreted as a semicolon in text. However, if for example an image file contains a byte that has the same value as a UTF-8 semicolon, the server won't process the entire data. A solution to this problem could be to change the protocol for communicating with the server so that the client sends the size of the data to be sent before it actually sends the data. With that information, the server could allocate a buffer of appropriate size and not interpret the data being sent as text.

Lastly, there are still a few bugs lurking in the code, such as operation transaction that are not processed properly. In the future these bugs could be fixed.

Contrasting Rust with other languages

As a coder, I've had experiences with other languages such as C++ and Java. I mention those languages specifically because they are some of the most frequently used for the type of work that Rust is targeted for: Networking & Concurrency. Rust has been built by trying to learn from the problems that have plagued these older programming languages, for example, a problem of C++ is dangling pointers, this is where a piece of memory stored in a variable points to a piece of memory that is no longer allocated to the C++ program. Not only can this cause unexpected issues during run time, it is also a security threat. In contrast, Rust does not allow this by making sure that all variables are deallocated once they are out of scope, this is determined by the blocks in program: everything between curly braces "{}" is considered a block. The Rust compiler will warn you if you are trying to use a variable that points to invalid memory and will not let you run the program. Another feature of Rust is that variables cannot be modified by default, you have to declare them as mutable. This

prevents accidental modification of data that can occur with languages like C++ and Java.

These, among other features, are an advantage: they allow the coder to focus on what matters for the program such as fixing bugs or implementing features instead of worrying about putting measures in place to prevent these issues intrinsic to the programming language.

In the past, I have created server and client software with Java, this included handling of Thread Pools, making sure functions are synchronized or variables are volatile and putting appropriate locks in place.

In contrast, my experience with Rust has been largely a breeze in this aspect. The Tokio library handles most of the details of concurrency, and the "async/awaits" keywords built into Rust make asynchronous programming much simpler. I simply had to worry about two things: which functions were running asynchronously and placing locks on variables. The latter one is much simpler compared to Java, the lock takes a couple of lines of code and I can then use it to read or write. Using those keywords, the Rust compiler can make sure that there aren't any data races as it keeps track of which variables are shared by which parts of the program at all times thanks to the Atomic Reference Counter (Arc), which I first mentioned on Chapter 1.2.

Overall, programming with Rust was an enjoyable experience.

Conclusion

There is still a lot of room to improve this database but through this experience I have learned that Rust can be significantly easier to code system programs in than other programming languages I've had experiences with. Overall, according to the original specification the development of this project proved to be quite successful.

References

1. Borthakur D. et al. *RocksDB Basics.* GitHub. 2019. Retrieved on the 13th of November 2019.
   https://github.com/facebook/rocksdb/wiki/RocksDB-Basics

2. Chapple M. *The ACID Database Model.* Lifewire. 2019. Retrieved on the 13th of November 2019.
   https://www.lifewire.com/the-acid-model-1019731

3. *ACID*. Wikipedia. 2019. Retrieved on the 13th of November 2019.
   https://en.wikipedia.org/wiki/ACID

4. 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

5. *PostgreSQL: Documentation: 7.1: Multi-Version Concurrency Control.* PostgreSQL. Retrieved on the 13th of November 2019.
   https://www.postgresql.org/docs/7.1/mvcc.html

6. *Distributed DBMS - Controlling Concurrency*. TutorialsPoint. Retrieved on the 13th of November 2019.
   https://www.tutorialspoint.com/distributed_dbms/distributed_dbms_controlling_concurrency.htm

7. *Introduction of B-Tree.* GeeksforGeeks. 2013. Retrieved on the 18th of November 2019.
   https://www.geeksforgeeks.org/introduction-of-b-tree-2/

8. Milanesi, C. *Beginning Rust: From Novice to Professional.* Apress. 2018.
   https://books.google.co.uk/books?id=LM5SDwAAQBAJ

9. BinaryAdventure. *Rust Tutorial - Lifetime Specifiers Explained*. YouTube. 2018. Retrieved on the 17th of October 2019.
   https://www.youtube.com/watch?v=QoEX-Vu-R6k

10. 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

11. 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

12. 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/

13. 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/

14. Klabnik S. Nichols C. The Rust Programming Language. Rust Lang Org. 2018. Retrieved on the 5th of February 2020.
   https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

15. Tokio Library Documentation. 2018. Retrieved on the 21st of February 2020.
   https://tokio.rs/docs/getting-started/echo/

16. "Builder" Tokio Library Documentation. 2020. Retrieved on the 28th of February 2020.
   https://docs.rs/tokio/0.2.13/tokio/runtime/struct.Builder.html

17. Inside Rust Blog. 2019. Retrieved on the 2nd of March 2020.
   https://blog.rust-lang.org/inside-rust/2019/10/11/AsyncAwait-Not-Send-Error-Improvements.html

18. "Spawn" Tokio Library Documentation. 2020. Retrieved on the 3rd of March 2020.
   https://docs.rs/tokio/0.2.13/tokio/fn.spawn.html

19. Rand Library Repository. 2020. Retrieved on the 7th of March 2020.
   https://crates.io/crates/rand

20. "sync_data" Rust "File" Library Documentation. 2020. Retrieved on the 8th of March 2020.
   https://doc.rust-lang.org/std/fs/struct.File.html#method.sync_data

21. "flush" Rust "io" Library Documentation. 2020. Retrieved on the 8th of March 2020.
   https://doc.rust-lang.org/std/io/trait.Write.html#tymethod.flush