University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Anita Śledź
Student no. 406384

Kajetan Husiatyński
Student no. 406160

Jan Ciołek
Student no. 406120

Michał Sala
Student no. 406329

Scylla Rust Driver

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
Jacek Sroka PhD

Warsaw, June 2021

Abstract

The purpose of this thesis is to describe the creation of an open-source Scylla database driver written in Rust. The driver was developed as a part of this thesis.

Scylla is a NoSQL database offering high performance and availability. It provides low and consistent latency and throughput 10 times higher than the most popular alternative on the market — Cassandra.

Scylla creators expressed the need for a functional driver implemented in and for Rust language. Rust is a modern programming language often called the successor of C++. It offers solutions to many issues regarding memory management and safety without impacting performance.

The driver is officially supported by ScyllaDB and is the first such complete tool available for developers that provides fully asynchronous API using Rust's build-in async/await syntax. It is fully compatible with Cassandra and supports Scylla-specific features.

Keywords

Rust, Scylla, driver, asynchronous, CQL, ScyllaDB, NoSQL

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics

Subject classification

Software
Software and its engineering
Software creation and management
Designing software
Software design engineering

Tytuł pracy w języku polskim

Sterownik do bazy danych Scylla w języku Rust

Contents

Chapter 1

Introduction

1.1. Motivation

The purpose of this thesis is to describe the creation of an open-source Scylla database driver written in Rust. The driver was developed as part of this thesis during the Team programming project course at the University of Warsaw as a Bachelor Thesis in Computer Science.

The team that worked on the project consisted of four people: Jan Ciołek, Kajetan Husiatyński, Michał Sala and Anita Śledź, under the supervision of Jacek Sroka PhD.

The project was commissioned by the ScyllaDB company. The project was looked after by two ScyllaDB employees — Piotr Sarna and Piotr Dulikowski, who were a part of a team responsible for creating the initial version of the driver. The driver started its life during a hackathon at ScyllaDB, but it was lacking most needed functionality. Our team was tasked with adding all of the expected features.

The development of the project took place during the academic year of 2020/2021, and has been divided into three main phases:

1. requirements specification,
2. development and implementation of a project and its documentation,
3. publishing and testing.

The main goal of the project was to create a fully functional Rust driver. The driver that was developed is optimized for Scylla database, using Scylla's specific shard-per-core architecture.

The first release of the driver took place on the 7th of April 2021.

1.2. Scylla

Scylla Database is one of many products created and offered by the authority contractor ScyllaDB company.

Scylla is a NoSQL database, which was released in September 2015. NoSQL databases are characterized by different data model than relational databases [13]. The modeling differs between NoSQL databases, for example, they can use a wide column model, key-value model or document model. The differences in modeling structures are a result of optimizing the response time and availability of such database. The data model of Scylla is a wide column model. It uses structures from relational database, but the data stored and their names

can be different from row to row. It is often interpreted as two-dimensional key-value store. Scylla is a distributed database, which means that the data stored in it can be stored across different locations, called data centers with multiple nodes. Singular information is being stored in many nodes, to prevent data loss. All of this offers high performance, scalability and functionality.

It operates using CQL protocol, which makes it compatible with the most popular alternative on the market — Cassandra [3]. There are some things that distinguish Scylla from Cassandra.

Scylla's language of choice is C++, unlike Cassandra, which is implemented in Java. As a result of the C++ specific implementation decisions, Scylla offers up to 10 times higher throughput.

Scylla is also characterized by its sharding feature. It divides the data between CPU cores, which allows for minimized synchronization and improves performance. The driver makes full use of this feature. Requests are sent directly to specific CPU cores to achieve the best speed possible.

More information about the Scylla Database can be found in here: [16, 11, 13, 5, 7, 9, 8, 12].

More about the Cassandra can be found in here: [3].

More about comparison of those two databases can be found in here: [4, 10].

1.3. Rust

Rust is a programming language of increasing popularity. For 5 years it has been gaining recognition among programmers thanks to its innovative approach to memory control. It ensures full security of memory management and thus the inability to trigger "undefined behavior". As a result, it is often referred to as the C++ successor. It has very similar capabilities and provides solutions to corresponding problems with comparable execution times and no issues with memory.

1.4. Project's main objectives

Driver is a full-feature library for the Rust language which allows communication with CQL-based NoSQL databases.

For asynchronous operations the driver used the Tokio framework. It is a popular and async library in the Rust ecosystem. Use of Tokio library for asynchronous actions is offering more than any other alternative library that exists in Rust — it provides non-blocking components, such as tool to work with asynchronous tasks, timeouts, sleeps, channels, and many many more. More information about it can be found in chapter 3.

In addition to being compatible with the Scylla database, the driver is compatible with all NoSQL databases using the CQL protocol. However, it is optimized for the specifics of Scylla, i.e., the driver can work in shard aware mode. Shard-per-core architecture is a feature that distinguishes Scylla among other databases. Also, each node in Scylla cluster operates peer-to-peer, without primary/replica model.

Apart from the features described in the Requirements chapter, there are several benchmarks created. The benchmarks compare the driver's speed with other Rust drivers, and also Scylla drivers in other programming languages.

The first release of the driver took place on the 7th of April 2021. The publication can be found on crates.io, Rust community's package registry. It is now officially available for Rust

projects.

1.5. Contents

The rest of the thesis is organized as follows:

- Chapter 2: describes the specifications of the project agreed upon with the client.
- Chapter 3: contains a description of Rust as a language, including Tokio, the asynchronous Rust platform. It also describes the Scylla database, and the role of containerization in the testing process.
- Chapter 4: describes the technical specification of the driver.
- Chapter 5: contains the documentation and description of the benchmarking process and its results.
- Chapter 6: summarizes the work on the driver, describes the publication process and potential future development directions.

There are also 3 appendixes in the thesis:

- Appendix A describes the team work organization and division of tasks.
- Appendix B describes the task division among the teammates.
- Appendix C contains description of files attached to the thesis.

Chapter 2

Requirements

At the beginning of the project, the team had a series of meetings with Scylla representatives.

An agreement was reached with the client regarding the features that needed to be implemented in order to provide a functional driver. The list is presented below:

1. Driver-side metrics — implementation of multiple metrics, for example number of queries performed by the driver, number of errors and latency,
2. Handling topology changes inside the database and presenting them to the user,
3. Custom error types,
4. Performance benchmarks,
5. Configurable load balancing algorithms — making driver more efficient by changing the request distribution algorithms,
6. Configurable retry policies of queries,
7. Support of Transport Layer Security Protocol (TLS),
8. Query builders — support for each of the four types of Scylla queries,
9. Support CQL batch statements,
10. Support CQL Authentication,
11. CQL Tracing — query trace recording,
12. Handling events pushed by the server,
13. Speculative execution,
14. Correctness and integration tests,
15. Documentation of benchmark,
16. Publication on crates.io.

At the end of the project, all of these features have been implemented and the driver has been published to crates.io, which is the official Rust package repository.

Implementation details on each feature can be found in the chapter 4.

Chapter 3

Development platform

3.1. Rust as a language

Rust is a programming language that is quickly gathering interest among programmers.

The first stable release was published in 2015, and since then the popularity has skyrocketed. In the years 2016-2020, the Stack Overflow community declared Rust the "Most Loved" programming language [15].

What distinguishes Rust, is that it does not permit any null pointers, dangling pointers or data races.

Other prominent features of Rust include: memory safety, no garbage collector, speed of execution.

3.1.1. Tokio and asynchronous operation in Rust

Rust has async/await keywords built in, but the standard library does not support asynchronous input-output operations. The solution to this problem was using one of the community crates. The library, which adds the missing features is Tokio[17]. It provides asynchronous sockets, files, and channels along with a runtime. This library was used in the project to support any asynchronous operations.

3.2. Scylla Database

Scylla was created as an attempt to create a wide-column modeled NoSQL database, like Cassandra. To achieve full compatibility with other databases, it uses the popular CQL binary protocol to communicate with clients [1, 14]. CQL binary protocol is a protocol firstly introduced in Cassandra database. This made it possible to create a single driver which can now be used for both Cassandra and Scylla.

Scylla has a few features not present in Cassandra, that required explicit support from the driver. The most important one being "shard-per-core" implementation. This implementation means, that on each node data is divided between CPU cores. Each core controls some part of the data so that no data is shared between threads. This allows for reduced synchronization thus improves performance. When sending a query, a Scylla-optimized driver has to ensure, that it is sending the data via connection to the proper CPU core.

3.2.1. Testing and Containerization

To ensure the best quality various testing methods were employed. The most important one was the Continuous Integration. Before each change could be accepted, a suite of automated tests was performed.

The tests consist of:

- Unit tests — Each important module has a test suite, that could be ran using standard rust tooling.
- Integration tests — There are multiple tests that connected to the database and performed various queries. This provides end to end testing of all functionalities and ensures compatibility with Scylla and Cassandra.
- Format check — All code is checked for compliance with formatting style guidelines using rustfmt tool.
- Automated code suggestions — cargo clippy is used, which is a tool providing suggestions ranging from common mistakes to code improvements.

If a proposed change failed any of tests above it would not be merged.

Some additional manual tests were created which could not be included in the CI. They covered various scenarios in which nodes were added, removed or even crashed. To make the setup of various scenarios as easy as possible each manual test is containerized using Docker containers.

Scylla and Cassandra provide docker images which allows to easily set up a local database instance.

3.2.2. Other compatible databases

The driver is fully compatible with Cassandra and Scylla

3.3. Driver

The driver is a standard Rust library, alternatively called: crate, that can be easily added to new Rust projects.

For asynchronous operations the driver used the Tokio framework, which is one of the most popular async libraries in the Rust ecosystem.

The driver is a library written fully in Rust. This means that there cannot be any invalid memory accesses or data races between threads. However other errors such as deadlocks are still possible, the compiler is not able to detect them. Alas this also means that in some situations performance had to be sacrificed to achieve safety, but benchmarks results are still great.

The only part of the driver that is not Rust dependent is the C++ openssl dependency used for TLS connections. The user does not have to depend on it as long as TLS connection is not necessary. Pure Rust implementation of TLS is on the way, but it is not ready to be used in the driver yet.

The driver was published on the official Rust package repository — `crates.io`.

Chapter 4

Architecture

4.1. Introduction

In the driver there are two key sections. The public interface available to the user and the internal architecture.

4.1.1. Public interface components

- **Session** — coordination of all requests and connections,
- **SessionBuilder** — creation of new **Session** instances,
- **Query** — creation and configuration of simple queries,
- **PreparedStatement** — representation of prepared query, which has far better performance than **Query**,
- **BatchStatement** — a batch of queries that can be executed at once,
- **QueryBuilder** — utility allowing simple creation of **Query**,
- **Metrics** — representation of metrics collected by **Session**,
- **RetryPolicy** — customizable strategy used to retry failed queries,
- **LoadBalancingPolicy** — configuration of distributing load equally between database nodes and sending queries to the nodes containing queried data,
- **SpeculativeExecutionPolicy** — customizable strategy used to decide whether the driver should try to do a new query that could finish faster than the one sent previously,
- **RowIterator** — iterator which allows for seamless access to rows acquired from multiple paged queries.

Query, **PreparedStatement** and **BatchStatement** can be passed to **Session** object to trigger execution. **RetryPolicy**, **LoadBalancingPolicy** and **SpeculativeExecutionPolicy** are passed to **SessionBuilder** during session creation process.

4.1.2. Internal components

- **Cluster** — database nodes and topology changes connection management,
- **Connection** — handling of request serialization and deserialization,
- **ConnectionKeeper** — detection of connection failure and reconnection,
- **TopologyReader** — fetches latest information about active nodes in the cluster.

All of these components work on different asynchronous fibers and communicate using message channels.

4.2. Prepared statement

A statement can be prepared to improve query performance. During preparation the database parses the query and prepares it for quick execution. Then when the driver decides to execute the query the database is ready and the query can be performed with a much better performance.

Preparing also has influence on load balancing. Prepared statements contain additional information, which allows to route queries so that the query is always sent to a node containing desired data.

4.3. Batch statements

A batch statement allows to execute multiple statements as an atomic operation. A single batch statement consists of multiple **Query** or **PreparedStatement**. All statements included in a batch must either an INSERT, UPDATE or DELETE. Statements that return rows are not allowed.

4.4. Metrics

Every **Session** keeps track of the number of performed and failed queries and their latency, and presents them to the user with the use of a **Metrics** object.

Queries are split into two categories — those that return an iterator and those that do not. This division was motivated by the fact that performing a paged query spawns a task that can send more than one request to the database resulting in multiple latency entries and increasing of the performed queries counter.

Before executing each query the driver increments the appropriate counter and saves the timestamp at the begging of the request. After receiving the response driver logs the latency of a query by measuring the elapsed time and pushes it into the latency histogram. If the driver detects an error it also increases the error counter.

The user can access metrics with **Session**'s getter method get_metrics(). The metrics provide a getter for every counter and methods that return average latency or histogram's percentile.

A use case is presented in scylla-rust-driver/examples/basic.rs file.

4.5. Retry policy

When a query fails the driver has to decide whether to retry it. To make the decision it needs to know some information about the query and the reason for its failure.

The most important information about the query is whether it is idempotent. When a query is idempotent it can be executed any number of times and the result will be the same. This could be for example a select. On the other hand when a query is not idempotent executing it more than once would lead to an invalid state. For example adding one to an element in a table gives different results when done multiple times. If the query is not idempotent the driver will only retry it in a few cases — one of them is when the node is still starting and it is known for sure that the query was not executed and it can be retried on another node.

Once it is known that a query can be retried the driver has to look at the reason of the failure. If the queried node is simply overloaded it is probably a good idea to query another node. If the query failed because there was a syntax error the driver should just give up and report the error to the user.

Retry policies are fully configurable, if someone needs some custom behaviour they can implement their own.

4.6. Load balancing

A Scylla cluster usually consists of multiple nodes. Each node is responsible for storing different parts of data. If a driver wants to minimize the latency between sending request and receiving a response, it has to choose the best node to contact for a given query — the one that stores all the needed data.

The process of choosing which nodes to contact is called load balancing. Before sending each request, the driver creates a load balancing plan, which is a list of nodes to contact. The plan is created by a load balancing policy, a configurable component of **Session**.

When a query is executed, the connection to a node from a load balancing plan is picked and a request is send via it. In case of a request failure, such as a timeout, the next node from load balancing plan can be contacted, depending on the decision of the retry policy.

Load balancing policies supported by the driver:

- Round robin load balancing policy,
- Datacenter-aware round robin load balancing policy,
- Token-aware round robin load balancing policy,
- Token-aware datacenter-aware round robin load balancing policy.

4.6.1. Round robin load balancing policy

The simplest policy. It distributes requests equally between known cluster nodes in a round-robin fashion. Each plan produced by this policy is a list of all nodes cyclically shifted by some factor α. α is changed after each plan calculation.

This policy does not take into account the distribution of data on nodes. Using it as the main policy may negatively affect performance.

4.6.2. Datacenter-aware round robin load balancing policy

This policy is parameterized using a datacenter name. It allows the driver to prioritize nodes in the closest datacenter.

Each plan produced by this policy is a concatenation of two lists:

- The first list is a list of nodes from a chosen datacenter cyclically shifted as in the round robin policy.
- The second list is a list of nodes from all other datacenters cyclically shifted as in the round robin policy.

This policy does not take into account the distribution of data on nodes. Using it as the main policy may negatively affect performance.

4.6.3. Token-aware round robin load balancing policy

Each node in a Scylla/Cassandra cluster is storing data, that can be identified using a set of tokens. A token is a 64-bit signed integer. One of important driver features, is the ability to calculate a token for a given **PreparedStatement**. This allows for load balancing optimization by producing plans which contain only such nodes, that the data stored on them is associated with a **PreparedStatement**'s token.

This policy takes advantage of the optimization mentioned above. Plan construction works in two phases:

- In the first phase, the policy gathers nodes that store data associated with a load balanced query's token. If no token is provided, the policy falls back to a set of all nodes.
- In the second phase, the policy passes a list created in the first step to the round robin load balancing policy, and outputs the processed result.

This policy takes the distribution of data on nodes into account. Using it as the main policy affects performance positively.

4.6.4. Token-aware datacenter-aware round robin load balancing policy

This policy behaves the same as the previous one (Token-aware round robin), with the exception of using a datacenter-aware round robin load balancing policy in the second phase of building the load balancing plan.

4.6.5. Shard-aware load balancing

Scylla has one key optimization that differentiates it from Cassandra — sharding. A single database node is additionally divided into shards. Each shard keeps a separate part of the data. Usually there is one shard per CPU core. When making a request the driver can calculate which shard keeps the data that is needed and send the request exactly to this shard. This allows to minimize synchronization between CPU cores which improves performance.

Shard-aware load balancing is automatically enabled once the driver detects that a node is divided into shards.

4.7. Speculative execution

Speculative execution is an optimization technique which allows to reduce query latency in high load scenarios. If a query takes too long to finish, speculative execution policy can decide to start another query in parallel. There is a chance that this second query will finish before the first one.

4.7.1. Simple speculative execution policy

In this execution policy the driver waits for a set amount of time before performing the next request in parallel. If this new query also takes too long to finish the driver starts the next one. This repeats at most a specified number of times.

4.7.2. Percentile speculative execution policy

This execution policy is similar to the simple speculative execution policy, but the amount of time to wait before starting speculative queries is set to nth percentile of query latencies. The nth percentile of query latencies is taken from **Session**'s **Metrics**.

4.8. Topology

Driver maintains information about the topology of its cluster — which nodes are alive, which are down, what data centers are there, etc.

On startup the driver is given addresses of a few nodes from the cluster. It connects to them and downloads information about all other existing nodes. Even if one or two nodes are dead it connects to other alive ones and still works without any issues.

Topology information is fetched using a single control connection to one of the active nodes in the cluster.

4.8.1. Handling topology changes

The topology can change at any moment. New nodes can be added to the cluster, some nodes might be removed or just crash. The driver handles these changes gracefully. It queries the latest topology on regular intervals and listens for any topology change events.

4.8.2. Handling events pushed by server

Driver can subscribe to cluster events, in order to quickly react to certain situations. For example, when a node in Scylla/Cassandra cluster becomes unreachable, an event carrying information about it can be sent to a driver. The driver uses this information to mark nodes as down/up or to trigger topology refresh earlier than planned. As a result it avoids routing queries to removed nodes.

4.9. Value passing using rust type system

Passing values in queries is made in a clean and easy to use way. When executing a query all bound values are given as a tuple of Rust objects. This means that the user can use normal Rust types such as **i64** and pass them straight to a query without any difficulties.

4.10. CQL support

CQL is used both in Scylla and Cassandra. This language has specified binary protocol, called CQL Binary Protocol. Scylla and Cassandra both use CQL Binary Protocol v4. Documentation can be found here: `https://github.com/apache/cassandra/blob/trunk/doc/native_protocol_v4.spec`.

Model offered by Cassandra Query Language is similar to the one offered by SQL. The data is stored in tables with rows and columns. CQL offers many types, including custom types and collection types.

Putting focus on implementing all CQL Binary Protocol features allowed the users to access full functionality of the database via the driver.

CQL types Scylla and CQL support integer representation of different sizes. Different precisions have different names, for example a 64 bits integer is represented as "bigint" and 16 bits integer is "smallint". Implementation of parsing and representation of these integers as Rust data types was introduced as follows:

CQL datatype	Rust datatype
int	i32
bigint	i64
smallint	i16
tinyint	i8

CQL supports also representation of real numbers, represented by two different data types: float and double. The two types differ in precision and hence the size of the variable. Parsing and representing these two types in Rust has been implemented as follows:

CQL datatype	Rust datatype
float	f32
double	f64

Other types that were introduced were boolean and universally unique identifier (uuid), with 128 bit label:

CQL datatype	Rust datatype
uuid	Uuid
boolean	bool

Next types implemented were varint, counter and decimal. Varint and decimal represent numbers bigger than regular integers covered in regular integer types, so they are represented by matching Rust types:

CQL datatype	Rust datatype
counter	Counter
varint	BigInt
decimal	BigDecimal

Time related CQL data types were implemented using the chrono crate, to make the experience more user-friendly:

CQL datatype	Rust datatype
date	NaiveDate
duration	Duration

CQL Collections Support for the CQL collection types Map, Set, Tuple and List has also been implemented. Collections parsing and representation was implemented with Rust types with the driver's type CQLValue:

CQL datatype	Rust datatype
List	Vec<CQLValue>
Map	Vec<(CQLValue, CQLValue)>
Set	Vec<CQLValue>
Tuple	Vec<CQLValue>

CQL Authentication CQL Authentication mechanism was added, making it available for the user in **SessionBuilder**. User can define username and password that they want to create connection with.

Authenticators are the database options by which user can authenticate their connection. The driver supports the following CQL Authenticators:

- AllowAllAuthenticator,
- CassandraAllowAllAuthenticator,
- CassandraPasswordAuthenticator,
- PasswordAuthenticator.

4.11. Error handling

Explicit error handling is an integral part of Rust. Each function that might fail returns a result that has to be explicitly checked for failure. In case of failure the user can access an enum type which covers all possible error causes.

4.12. Documentation

The driver is well documented including:

- Code comments,
- API documentation [2],
- A tutorial book [6].

Since the driver is an open source project, each line of code needed to be clean and understandable for the reader. Thanks to thorough review done by team members and ScyllaDB developers, any part that could be incomprehensible is commented.

API documentation is a part of project's documentation. It contains the description of driver's application programming interface.

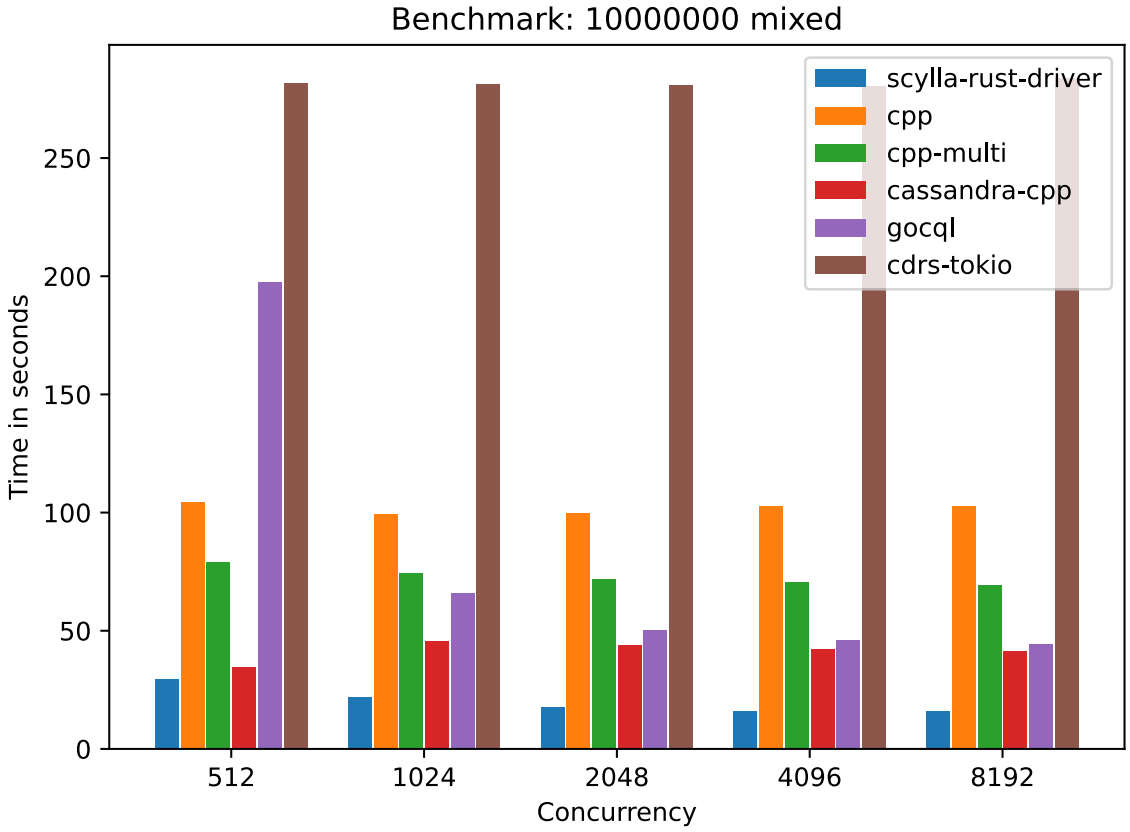
A tutorial book has been created. The style the book has been kept in similar to Rust tutorial book, to make it easier to start up with a clean project for new Rust users. It contains examples of real code and precise description of each feature.

Chapter 5

Benchmarking

Benchmarks were ran to find out how good the performance of various drivers is. The results were excellent. Rust driver is faster than any other driver available.

Here are the results of a benchmark, where the driver had to perform 10,000,000 inserts mixed with 10,000,000 selects, while doing at most the given number of concurrent requests at once:



- **scylla-rust-driver** — This driver.
- **cpp** — official Scylla C++ driver. Sadly at the moment multithreaded functionality is buggy, so it operates on a single thread.

- **cpp-multi** — DataStax C++ Cassandra driver. Optimized for Cassandra, but can also be used with Scylla. Uses all available CPUs.
- **cassandra-cpp** — Rust bindings to the DataStax C++ Cassandra driver mentioned above.
- **gocql** — Driver implemented in Go.
- **cdrs-tokio** — The only other asynchronous driver written in Rust. Currently it is much slower and lacks some features.

There are still some optimizations that could be done in this driver. With future development, better results can be achieved. All benchmarks are publicly available on a Github repository: `https://github.com/cvybhu/rust-driver-benchmarks`. Each benchmark is in a dedicated docker container, which allows to easily reproduce them.

Chapter 6

Summary

As part of this thesis, a driver for the Scylla database in the Rust language was created. The driver was fully implemented during the project and published as a public library, available to all developers of the Rust language. Each requirement from the requirements chapter has been implemented. The team was supported by ScyllaDB developers in order to come up with the adequate solution to any encountered issues. Finally, contracting authority was thrilled with the result.

The driver has been published on crates.io — the Rust package repository. Multiple users are already using this crate and have come to the team with questions and features requests.

Future plans for the driver include:

- Better support for lightweight transactions (LWT),
- Possibly replacing C++ driver with bindings to this driver,
- More integration tests to test for all possible scenarios.

Appendix A

Team work organization

During development Github was used along with its rich set of features. The code was hosted on a public git repository, which can be found here: https://github.com/scylladb/scylla-rust-driver.

A.1. Issues

The tasks were managed using Github issues. For each task there was an issue with a comprehensive description of the problem.

This made work organization efficient and effortless.

Additionally everyone can create an issue. This makes Github issues feature also serve as a bug tracker. There were multiple users who created new issues ranging from bug reports to feature requests.

A.2. Pull Requests

All proposed changes were created as Github pull requests by contributors. Every pull request contains description of the changes. Before accepting a pull request it had to be rigorously reviewed by Scylla developers.

A.3. Reviews

Github pull requests have great review capabilities. Everyone can create a review where they can leave comments about each line. This makes discussion about specific changes much more organized. Each comment starts its own thread where only this change is discussed.

A pull request can only be merged once all reviewers approve of the changes.

A.4. Open source

The driver is a public open-source repository, which means that anyone can create new issues, pull requests or review code. This allowed to build a community around the driver. During development the team was eager to help people from outside to implement small features that they were missing. Features described in this thesis were implemented only by the thesis' authors.

Appendix B

Division of tasks

Individual tasks have been done by separate team members or in smaller groups.

- Jan Ciołek:
  - Basic CQL types receiving and sending,
  - User defined types,
  - Reconnecting to the cluster,
  - USE KEYSPACE statement,
  - Retry policy,
  - Query tracing,
  - Documentation,
  - Benchmarks — Rust, C++, cassandra-cpp, cdrs-tokio.
- Kajetan Husiatyński:
  - Driver-side metrics,
  - CQL types support — collection types,
  - TLS support,
  - Schema agreement,
  - CI improvements,
  - Benchmarks — resolving datarace issues.
- Michał Sala:
  - Load balancing,
  - Receiving and reacting to events pushed by server,
  - Integration with logging library,
  - Replica sets calculation for token-aware load balancing,
  - Expose a type that represents unset value,
  - Batch statement support,
  - Speculative execution support,
  - Benchmarks — gocql.

- Anita Śledź:
  - Error reformatting,
  - CQL types support:
    * numeric types,
    * data and time types,
    * collection types,
    * other types,
  - CQL authentication support for:
    * Scylla,
    * Cassandra,
  - Schema change event handling.

Appendix C

Files attached to the thesis

Files attached to the thesis are clones of repositories the team worked on. Each repository contains README.md file with thorough instructions of how to build project or example.

- scylla-rust-driver directory — official repository clone from Scylla Github account for the driver, containing source code, examples, tests. The repository can be found online here.
- rust-driver-tests directory — repository clone with manual tests.
- rust-driver-benchmark directory — repository clone with benchmark source code,
- documentation directory — directory containing Documentation Book pdf named book.pdf,
- thesis directory — directory with source code for this thesis, written in Latex,
- Scylla_Rust_Driver_Thesis.pdf — thesis (this file).

C.1. Testing

The driver was thoroughly tested including unit tests and integration tests covering various scenarios.

Separately, the manual tests were developed and they covered scenarios:

- Various node events tests,
- Schema change test,
- Data Center Aware Network Test.

In order to test those scenarios, each test has a destined docker container. An instruction on how to run each test is attached.

Bibliography

[1] *Cassandra Query Language*. URL: https://cassandra.apache.org/doc/latest/cql/index.html.

[2] *Documentation*. URL: https://docs.rs/scylla/0.1.0/scylla/.

[3] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. URL: https://doi.org/10.1145/1773912.1773922.

[4] Ashraf Mahgoub et al. "Suitability of NoSQL systems — Cassandra and ScyllaDB — For IoT workloads". In: *2017 9th International Conference on Communication Systems and Networks (COMSNETS)*. 2017, pp. 476–479. DOI: 10.1109/COMSNETS.2017.7945437.

[5] *Scaling Up versus Scaling Out: Mythbusting Database Deployment Options for Big Data*. URL: https://go.scylladb.com/whitepaper-scaling-up-vs-scaling-out-offer.html.

[6] *Scylla Rust Driver Documentation Book*. URL: https://cvybhu.github.io/scyllabook/index.html.

[7] ScyllaDB. *6 Reasons to Switch from DataStax to Scylla*. URL: https://go.scylladb.com/switch-reasons-datastax-scylla-offer.html.

[8] ScyllaDB. *Building the Real-Time Big Data Database: Seven Design Principles behind Scylla*. URL: https://go.scylladb.com/real-time-big-data-database-principles-offer.html.

[9] ScyllaDB. *Maximizing Scylla Performance: A Guide to Getting the Most from Your Scylla Database*. URL: https://www.scylladb.com/wp-content/uploads/guide-maximizing-scylla-performance-1.pdf.

[10] ScyllaDB. *MIGRATION GUIDE:Apache Cassandra to Scylla*. URL: https://www.scylladb.com/wp-content/uploads/Cassandra-to-Scylla-Migration-Guide.pdf.

[11] ScyllaDB. *NoSQL and NewSQL: A Comparison of Distributed Database Systems*. URL: https://go.scylladb.com/nosql-to-newsql-distributed-database-wp-offer.html.

[12] ScyllaDB. *Reducing Complexity with a Self-Optimizing Database*. URL: https://go.scylladb.com/self-optimizing-database-offer.html.

[13] ScyllaDB. *SQL to NoSQL: Architecture Differences and Considerations for Migration*. URL: https://go.scylladb.com/sql-to-nosql-architecture-wp-offer.html.

[14] A. Singh. *Instant Cassandra Query Language*. **. Packt Publishing, 2013. ISBN: 9781783282722. URL: https://books.google.pl/books?id=%5C_eP7AAAAQBAJ.

[15] *Stack Overflow Survey*. URL: https://insights.stackoverflow.com/survey/2020#technology-programming-scripting-and-markup-languages-professional-developers.

[16] Nishant Suneja. “Scylladb optimizes database architecture to maximize hardware performance”. In: *IEEE Software* 36.4 (2019), pp. 96–100. DOI: 10.1109/MS.2019.2909854.

[17] *Tokio framework*. URL: https://tokio.rs.