University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Stanisław Czech
Student no. 448209

Piotr Maksymiuk
Student no. 449319

Zuzanna Ossowska
Student no. 448548

Stanisław Polit
Student no. 449061

New ScyllaDB JavaScript over Rust driver

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer-Daszkiewicz
Associate Professor
Institute of Informatics

Warsaw, June 2025

Abstract

The purpose of this thesis is to create an open-source ScyllaDB database driver written for Node.js and TypeScript as a compatibility layer over ScyllaDB Rust driver. Interface of the developed driver is based on the DataStax Node.js driver, one of the most common drivers used when connecting to ScyllaDB Cloud (at the time of writing this thesis). By taking advantage of Rust native performance we managed to increase the throughput of the driver over its predecessor in some of the tested conditions. Our approach leverages ScyllaDB-specific features, which were lacking in the DataStax driver, as it was designed to support only Cassandra and DataStax Enterprise databases. We put heavy focus on making the driver easy to maintain and develop further by future developers.

Keywords

ScyllaDB, Cassandra, JavaScript, Rust, database, driver, NAPI-RS, DataStax, NAPI

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Informatics, Computer Science

Subject classification

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

Tytuł pracy w języku polskim

Nowy sterownik do bazy ScyllaDB dla języka JavaScript jako warstwa kompatybilności nad sterownikiem w Rust

Contents

Chapter 1

Introduction

In the modern world, data is one of the most valuable resources. Most modern applications require efficient and reliable ways to manage, process and store data. As the amount of data grows, so does the need for scalable and performant databases. Ensuring that developers have efficient tools to interact with databases is crucial for the success of modern applications.

One of such databases is delivered by ScyllaDB [27]. ScyllaDB is a NoSQL distributed database offering high performance and availability. It offers low and consistent latency, with throughput reported to be up to 10 times higher than one of the most popular alternatives on the market — Cassandra [2]. ScyllaDB is available in two main options: Enterprise and Cloud. The Enterprise version is self-hosted and comes with commercial support, while the Cloud version is a fully managed service provided by the ScyllaDB company.

Cassandra is a database initially developed by Facebook, currently managed by the Apache Software Foundation [4]. It's written in Java, compared to ScyllaDB which is written in C++. Similarly to ScyllaDB, it can scale horizontally to allow higher throughput and increased availability compared to a single machine.

To enable applications to interact with databases, a driver is essential. A driver is a software component that enables communication between an application and a database. However, each programming language requires its specific driver. To address this need, companies often develop drivers tailored to leverage the full capabilities of their databases.

CQL protocol [8] is a protocol designed for the communication between drivers and the NoSQL databases. Although similar in syntax to SQL language, it lacks some of its features [6]. It was designed primarily for Cassandra databases, but it's currently used by several databases including already mentioned ScyllaDB and other databases like DataStax Enterprise [12].

DataStax currently provides a driver for JavaScript [9]. The driver was developed with a focus on compatibility with Cassandra and databases provided by DataStax. The driver is an open-source project, but it is not actively developed.

ScyllaDB currently offers drivers for its database for Rust, C/C++, Java, Python, and Go. By using CQL protocol ScyllaDB is compatible with the Cassandra database. Because of that, it is possible to use ScyllaDB in JavaScript with a driver developed by DataStax, which at the time of writing this thesis is one of the most commonly used drivers when connecting to ScyllaDB Cloud. Unfortunately, this driver does not support all of ScyllaDB functionalities and is not being developed in that direction.

JavaScript [15] is one of the most popular [33] programming languages in the world, widely used for web apps development. It is a cross-platform, dynamically-typed programming language running on one thread with a focus on being event-driven using its just-in-time compiler. JavaScript's versatility makes it a good choice for developers building modern, scalable applications. In the Cassandra and ScyllaDB ecosystems, the JavaScript driver developed by DataStax is currently the most popular option used for connecting the applications with the database.

TypeScript [34] is a language based on JavaScript, adding static typing. Code written in TypeScript can be compiled to JavaScript. It allows to use code written in one language in the other. A library written in JavaScript needs only a type annotation file to be used in TypeScript.

Rust [25] is the most admired (according to StackOverflow survey [32]), cross-platform, compiled, statically typed programming language. Unlike JavaScript, which supports mainly asynchronous execution, Rust also supports concurrency. One of Rust's key strengths is its emphasis on memory safety, ensured by the borrow checker, an integral part of its compiler. In the case of ScyllaDB, the Rust driver is a dynamically developed and well-maintained solution.

Rust driver [31, 24] was originally created as a student project for the Team Programming Project course at the University of Warsaw. It was developed as a dedicated driver for the ScyllaDB database, compatible with Cassandra and other databases using CQL protocol.

Our task, as part of the thesis, is to create a dedicated driver for the ScyllaDB database for JavaScript. We do not need to rewrite the entire database handling from scratch — the goal is to write an intermediate layer between JavaScript and the existing Rust driver. By implementing the majority of the driver logic in Rust, we plan to leverage the performance benefits of a compiled language, optimized for speed, compared to interpreted languages such as JavaScript. The driver should also be compatible with TypeScript, as was the DataStax driver.

The primary requirements of the project are to maintain the current interface of the DataStax driver [11], with an emphasis on the efficiency of the solution. We are also supposed to test our code with already existing tests from the DataStax driver to prevent any regression in functionality. Like the other discussed drivers (especially the one for Rust, which our solution is based on), our driver should be compatible with the Cassandra database and other NoSQL databases using the CQL protocol. In addition to implementation, we are going to benchmark the performance of the driver.

Chapter 2

Project details

Databases have clearly defined communication protocols that are used by the drivers. However, at the driver level, the interface varies between different implementations. The goal of this project is to rewrite an existing driver to work over a different one, which presents multiple challenges when connecting their interfaces. The final data path between the user and the database is presented in Figure 2.1. As a result, there will be some API differences between the new driver and the DataStax driver. However, these changes should be minimal and they should not impact the user's experience in a significant way.

Despite these challenges, writing a driver over an existing one has several advantages, which we describe in Section!2.1. We then provide a high-level overview of how the database processes requests and compare the approaches of the DataStax driver and the Rust driver.

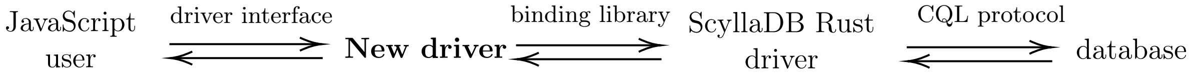


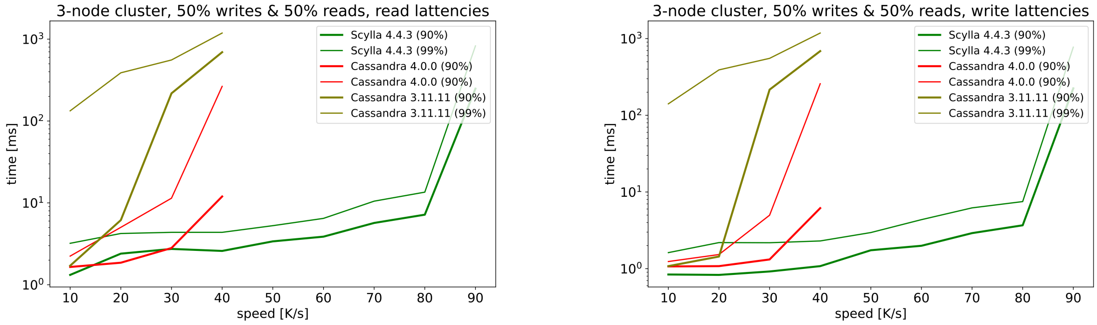
Figure 2.1: Operating diagram of the new driver

2.1. Motivations for the project

2.1.1. Performance

There are many reasons why clients are switching from Cassandra to ScyllaDB. One of them is the performance improvement that ScyllaDB offers. A speed comparison between ScyllaDB and Cassandra database is presented in Figure 2.2.

These improvements do not come just from the database itself. The driver that is used to connect to the database is also important. It should be efficient so there is no time overhead when connecting to the database.



(a) The 90- and 99-percentile latencies of SELECT statements.

(b) The 90- and 99-percentile latencies of UPDATE statements.

Figure 2.2: Speed comparison of ScyllaDB and Cassandra databases (logarithmic scale). Source: [3], raw data provided by ScyllaDB employees.

2.1.2. Maintenance

Apart from the performance, the maintenance costs are also important. The driver should be easy to maintain and develop further. The proposed solution is a driver that is just an overlay over the Rust driver. This concept was already proven in creation of the ScyllaDB C/C++ driver which was also written as an overlay over the Rust driver. This approach makes it easy to introduce new database features across all drivers, as the complex database handling logic only needs to be implemented once. Adding the feature to the other drivers is then a matter of creating a new function in the overlay driver, which is much easier and less error-prone.

2.1.3. Features

There are a lot of features that make ScyllaDB so performant. Some of them are inside the database and the driver does not need to be aware of their existence in order to take advantage of them. However, some features can improve the performance of the requests only if the driver is aware of them. This is why a creation of a database-specific driver is so important. Features that are introduced in the new driver are shard awareness [28] and tablets [29, 30].

Shards

ScyllaDB is a distributed database, meaning its data is stored across multiple machines. The data is divided into nodes, with each node running on a different machine. Since each machine can have multiple CPUs, ScyllaDB takes advantage of that by creating data shards, one for each core. These shards operate independently and can be accessed separately. Each shard is managed by a dedicated thread, which can communicate with other threads when cross-shard data access is needed. When a database driver is shard-aware, it can send requests directly to the appropriate shard, minimizing inter-shard communication. This reduces latency and improves performance.

Tablets

One of the recently added features to ScyllaDB is tablets. They are used to distribute data between the nodes as evenly as possible. When changes to the infrastructure — like adding or deleting a node or a whole datacenter — are made, tablets are moved to ensure that the data is still evenly distributed. When a database driver is tablet-aware, it can take advantage of them in a similar way as shards, by sending statements directly to the appropriate tablet to minimize inter-node communication.

2.2. Types of requests

There are several ways to categorize requests to the database. Each request can belong to multiple categories simultaneously. In this section, we describe categories relevant to this thesis.

2.2.1. Prepared and unprepared statements

Each request is represented as a string that must be parsed before execution. When users execute multiple requests with the same statement but different parameters using string-based requests, the database must parse each statement separately, upon each execution. This is known as an unprepared statement. Cassandra and ScyllaDB offer an alternative called a prepared statement.

Prepared statements work as follows:

- The driver sends the statement string for the request.
- The database prepares the statement internally and returns the ID associated with the prepared statement.
- The driver stores the ID as a reference to the prepared statement.
- When executing a prepared statement, the driver sends the ID along with the parameters instead of the statement string.
- Upon receiving the request, the database uses the preprocessed information from the preparation phase, eliminating the need to parse the statement string again.

The time cost of preparing a statement and executing it once is higher than that of executing a single unprepared statement. However, when the same statement is executed multiple times, using a prepared statement can offer a speed-up of even 2 times compared to the unprepared version.

2.2.2. Regular vs batch statements

A batch statement allows to execute a set of modifications in a single request. They only support inserting, updating and deleting data. Batch operations ensure atomicity, meaning that either all of the statements in the batch are executed or none of them. They also ensure isolation of the statements, meaning that a partial update of the data cannot be accessed by other requests until the batch is completed. Batch statements can improve performance when only one partition of data is accessed, but in case of large amounts of data updates, they can result in an additional time overhead of coordinating the inserts between partitions.

2.2.3. Paged and unpaged queries

Some database queries may produce responses too large for the driver to process all at once. For this reason, the driver developers recommend using query paging [23]. Paging allows to process a response in chunks (pages), each containing a limited number of rows. Once a page is processed, the next page of the query result can be retrieved. Since the database uses the CQL protocol, which supports paging, the driver can leverage this functionality.

2.3. DataStax driver

At the time of writing this thesis, the DataStax driver is in version 4. It had one minor release from version 4.7 to 4.8 in that period. This release contained minor bug fixes and an updated version of used dependencies. Its interface is described in the documentation [11].

2.3.1. Prepared statements

When a statement is prepared in the DataStax driver, it's stored in the internal cache. When the following requests with the same statement string are executed, the internal cache is used to avoid preparing previously prepared statements.

When a statement is prepared, the driver has information about the expected parameter types. This information is then used to correctly serialize the provided values. When the type is known, the driver allows the user to supply a value of that type in various forms. For instance, a CQL `date` parameter can be passed as a string, a JavaScript `Date` object, or a custom class representing a CQL date.

Regardless of the form used, all parameters are then encoded according to the CQL protocol specification [8].

2.3.2. Unprepared statements

When a user wants to execute an unprepared statement, the driver can learn about the types associated with the statement in the following ways:

1. Type hints: the user explicitly states the type as an optional argument to the execute function. These hints can be provided either for all parameters or only some of them. When hints are provided, the driver treats this information similarly to the type information, received from the prepared statement.

2. Type guessing: when the user does not provide type hints for some of the arguments, the driver attempts to guess the type of these parameters. This option is more limiting to the user, as some pairs of CQL types like the CQL `list` and the CQL `set` use the same representation. As a result, the driver cannot guess the CQL `set` type.

2.4. Rust driver

The Rust driver is actively developed by the ScyllaDB company. At the beginning of writing this thesis, the latest version of the driver was 0.14 [20]. During the writing process, version 1.0 [21] was released, introducing a stable interface as described in the official documentation [22]. As a result of different languages and design choices the approach of the Rust driver varies in many places from the DataStax driver behavior.

2.4.1. Prepared statements

When a statement is prepared, the Rust driver returns an object representing the prepared statement. This object can then be used to execute requests based on that statement. Unlike the DataStax driver, where the driver itself avoids re-preparing the same statement, the Rust driver places this responsibility on the user.

2.4.2. Unprepared statements

In the Rust driver, there are unprepared statements, but they do not have parameters. When executing a statement with parameters, the driver still calls the database to prepare it. However, this preparation occurs on a single connection to a node, compared to connections to all nodes required during regular preparation. As a result, this process is slightly faster than regular statement preparation.

This approach allows the driver to determine the types associated with a statement, whether it is prepared or unprepared. While being written in a statically typed language imposes certain limitations, the driver's design still provides a high degree of flexibility.

When providing a value for a parameter, any type can be used as long as it can be serialized into the raw format required by the CQL protocol (this can be achieved by implementing the `SerializeValue` trait). This approach allows users to define custom types representing CQL values, while still enabling the use of types that are already provided by the driver.

When the number and types of parameters are known at compile time, it is recommended to pass the parameters as a tuple. However, when this information is not available at compile time, as will be the case for our driver, one of the available options is to use a vector of `CQLValue` objects — each can store a value of any CQL type. The overhead of this approach is significant, but it is necessary to support JavaScript's dynamic typing.

Chapter 3

Development overview

3.1. Language bindings

Language bindings act as a bridge between two different programming languages. They allow to use code written in one language in another language. This enables seamless integration of libraries written in different languages. In our case, language bindings are used to connect the JavaScript part of our driver with already existing Rust driver for ScyllaDB. Two main libraries that can be used to create JavaScript language bindings in Rust are NAPI-RS and Neon.

3.1.1. NAPI-RS

NAPI-RS [17] is a library that enables the creation of JavaScript bindings in Rust. It is built on the Node-API (N-API), an API provided by Node.js for creating native addons. While it is a newer library compared to Neon, it is actively developed and has a growing community.

Key features claimed by the developers of NAPI-RS include:

- Performance — designed to be fast and efficient, bringing native performance for Node.js.
- Safety — provides a memory-safe API, guaranteed by the Rust compiler, that prevents common mistakes.
- Ease parallelism — provides a high-level API that allows for easy implementation of parallelism.
- Zero copy — allows for efficient data transfer between JavaScript and Rust via `Buffer` and `TypedArray`.

NAPI-RS provides a high-level API that allows for easy and clean implementation of JavaScript bindings in Rust. In Listing 3.1 we can see an example of how to create a simple JavaScript function in Rust using NAPI-RS.

Listing 3.1: Example of NAPI-RS usage.

```rust
use napi_derive::napi;

#[napi]
fn sum(a: u32, b: u32) -> u32 {
  a + b
}
```

3.1.2. Neon

Neon [18] is another framework for creating native Node.js modules using Rust. While it is an older library, it has a community of similar size to that of NAPI-RS.

Key features claimed by the developers of Neon include:

- Simplicity — adds a level of abstraction over the native Node.js addon API.
- Safe parallelism — provides a safe way to run Rust code in parallel.
- Direct V8 Support — allows working directly with the V8 JavaScript engine, offering more control over performance.

Neon provides a lower-level API than NAPI-RS, which allows for more control over the bindings, but also requires more work. In Listing 3.2 we can see an example of how to create a simple JavaScript function in Rust using Neon. As we can see, the code is more verbose than the equivalent code in NAPI-RS.

Listing 3.2: Example of Neon usage.

```
use neon::prelude::*;

fn sum(mut cx: FunctionContext) -> JsResult<JsNumber> {
  let a = cx.argument::<JsNumber>(0)?.value(&mut cx);
  let b = cx.argument::<JsNumber>(1)?.value(&mut cx);
  Ok(cx.number(a + b))
}

register_module!(mut cx, {
  cx.export_function("sum", sum)
});
```

3.1.3. Tests results

We conducted a series of tests before selecting a library that would be suitable for our project. These tests were divided into 3 categories of function calls: simple, receive and send. Tests were executed with different number of calls. For each value, the time measurement was performed 5 times and then the result was averaged. The standard deviation was also calculated and plotted on the graphs. We analyzed the NAPI-RS library, the Neon library and pure Node.js for comparison.

- Simple function call

  The test involved calling a simple function in a loop and measuring the execution time of all calls. Listing 3.3 presents the function called in the test.

  Listing 3.3: Simple function adding two numbers. Pure NodeJS implementation

  ```
  function aPlusB(a, b) {
      return a+b
  }
  ```

  Figure 3.1a presents the results of the tests. The NAPI-RS library has a smaller time overhead.

- Receive function call

  The test involved generating data of a specified size using a library function, which was called multiple times and the resulting data was passed to JavaScript.

  Listing 3.4: Data generating function. Pure NodeJS implementation

  ```
  let _tmp = null
  function generateLotOfData(size){
      if(_tmp == null)
          _tmp = Array(size).fill(null).map((_, i) => i);
      return _tmp
  }
  ```

  The `generateLotOfData(size)` function from Listing 3.4 generates arrays of the specified size, filled with consecutive integers. Upon passing from the library to JavaScript, the correctness of data transfer was checked.

  Figure 3.1b presents the results of the tests. The NAPI-RS library has a smaller time overhead.

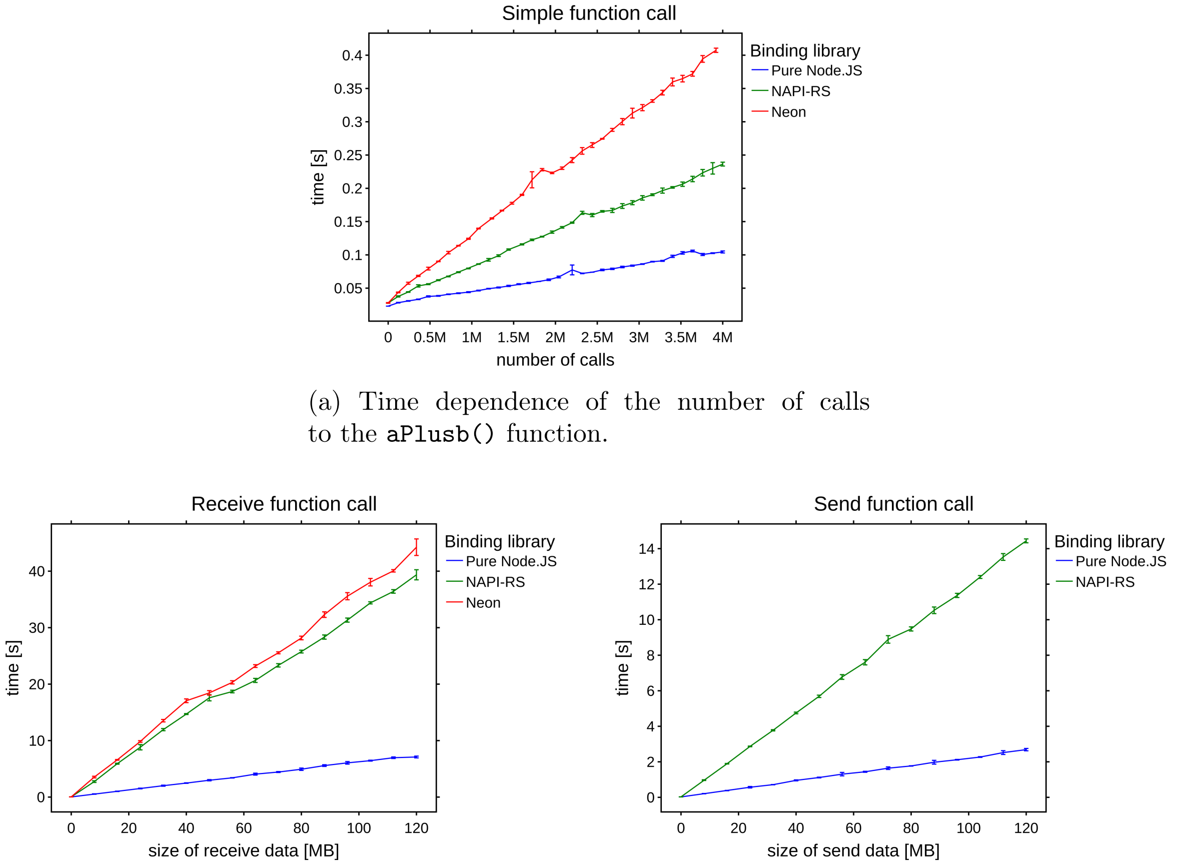
- Send function call

  The test involved sending data to the library by repeatedly calling a function to retrieve the right chunk from JavaScript. When trying to implement this using Neon, we ran into problems — despite its advertised simplicity, the documentation and code examples were lacking. Because of that, we were unable to create the required code in a reasonable amount of time.

  Listing 3.5: Data receiving function. Pure NodeJS implementation

  ```
  function getLotOfData(data) {
      const sum = data.reduce((partialSum, a) => partialSum + a, 0);
      return sum % 0xFE000000
  }
  ```

  Figure 3.1c presents the results of the tests.



(a) Time dependence of the number of calls to the `aPlusb()` function.

(b) Time dependence of the size of data received.

(c) Time dependence of the size of the sent data.

Figure 3.1: Results of the library comparison.

3.1.4. Conclusion

We decided to use NAPI-RS over Neon for the following reasons:

- Library approach — In NAPI-RS, the library handles the serialization of data into the expected Rust types. This lets us take full advantage of Rust's static typing and any related optimizations. With Neon, on the other hand, we have to manually parse values into the correct types.
- Simplicity of use — As a result of the serialization model, NAPI-RS leads to cleaner and shorter code. Moreover, as described in Section 3.1.3 , we were unable to implement code in Neon for a simple example. Based on that experience, we assumed similar issues would likely occur in the future.
- Performance — Our tests showed that NAPI-RS had the lowest time overhead. Since driver efficiency is a critical requirement, this was an important factor in our decision.
- Documentation — Although the documentation for both tools is far from perfect, NAPI-RS's documentation is slightly more complete and easier to navigate.

Since our choice of NAPI-RS over Neon was based on more than just performance, we decided not to spend an extended amount of time implementing the `Send` function call in Neon, and instead relied on partial performance data.

3.2. Documentation

To ensure the maintainability of the driver, we put a lot of effort into extensive documentation of the code. We decided to use JSDoc — a tool that was used by DataStax in their driver.

JSDoc [16] is a documentation-generating tool for JavaScript. It generates API documentation from comments in the code. This way, we can ensure that the documentation is always up to date with the code. It allows us to write comments that explain the code and its purpose, which is crucial for future developers who will work on the project, but also to generate something easy to read and understand for the users of the driver.

JSDoc scans source code looking for comments starting with specific formatting. Based on these and special attributes added to the relevant comments, it constructs HTML documentation. A programmer can specify multiple attributes for example defining the type and number of the parameters, return type and description of the function.

JSDoc was already used in the DataStax driver which makes our task a bit easier. We can reuse the existing comments, but early on we realized that there are some issues with the existing documentation. The comments are not always complete, correct or easy to understand. Along with the development we are improving them, for example adding missing types, fixing typos or adding more detailed and clear descriptions. DataStax also used HTML tags to style the documentation, which we decided to avoid. To improve in-code readability we switched to markdown formatting that is easier to read in its raw form.

Because of the already mentioned differences between JavaScript and Rust, it is helpful to document types of parameters and return values. This way, the user can easily understand what the function expects and what it returns, preventing type errors from happening.

Thanks to GitHub Pages [14], we can host the documentation online [13]. This way it is easily accessible and automatically updated with every driver release. In Listing 3.6 there are some examples of documentation in the source code.

Listing 3.6: Examples of documentation comments in the source code.

```
/**
 * Creates a new random (version 4) Uuid.
 * @param {function} [callback] Optional callback to be invoked with the error as
 * first parameter and the created Uuid as second parameter.
 * @returns {Uuid}
 */
static random(callback)

/**
 * Gets the hours, a number from 0 to 24.
 * @readonly
 * @type {Number}
 */
get hour()
```

3.3. Tests

The DataStax driver has a number of tests ensuring correct behavior. These tests are split into two categories — unit tests and integration tests.

3.3.1. Integration tests

These tests ensure the correct end-to-end behavior of the driver. They create a database, keyspaces and tables required for the test. Instead of running the database for the purpose of tests, they make use of a tool called CCM [1]. CCM is a Python script to create, launch and remove Cassandra clusters on a local machine. Two versions of CCM are important for this thesis. The original one, developed by DataStax [10], is in the process of being ported to the Apache incubator [7] at the time of writing this thesis. The second one, a fork of the original, is developed by ScyllaDB company [26].

Integration tests of the DataStax driver are set up to use the DataStax CCM. When we attempted to run these tests on the code of the DataStax driver, we encountered several issues. First, we were unable to launch the tests on our local machines. Second, when we ran the tests on GitHub Actions their behavior was inconsistent between launches. Each execution on the same code produced a different number of passing tests. Some tests consistently failed on every launch, while others failed only occasionally. With those issues in mind, based on the suggestion from our mentor from ScyllaDB, we switched to scylla-CCM. This fully fixed our first issue and reduced the amount of tests that were not working properly. We identified faulty tests, disabled them and passed to the company for further investigation.

Tests created for the old driver are important for us as, alongside the documentation, they describe the API of the driver. Compared to the documentation they go more in-depth into how the driver behaves. This includes things like error throwing, which is not described in the documentation but is defined in integration tests (and by extension is also the expected behavior of our driver). There are some tests that, instead of using the Cassandra database, are run with the DataStax Enterprise database. We removed these tests because implementing features specific to the DataStax database is not within our plans.

3.3.2. Unit tests

By definition, the goal of these tests is to ensure the correct behavior of single units of code. They may be a part of the exposed API, or they may test some code that is not directly exposed to the user. Some unit tests verify the correctness of some inner logic that is specific to the DataStax driver, but will be implemented differently in the new driver. It means that the new driver will still be compatible with the API, but may fail some unit tests.

During the development of the driver, we identify and enable unit tests that are responsible for testing the interface exposed to the user. This is done to ensure that the code we create behaves in the same way that is expected in the DataStax driver. Existing unit tests are updated to cover the newly developed code when possible. Otherwise, they are removed.

To comply with the best practices we also extend existing unit tests or create new ones for the new code. This is especially important for verifying the correctness of data passed through the NAPI layer, as we found several bugs related to having incorrect types of values in Rust and JS. We also discovered some bugs related to the inner workings of the NAPI-RS library (see Chapter 5).

3.3.3. Examples

In the DataStax driver, there are several examples showing the usage of the driver. While they do not test the code in the way that the integration tests are supposed to do, they ensure that some basic parts of the driver logic work correctly. Moreover, they are helpful during development as they can be run faster than integration tests, are easier to debug and can identify issues when they fail.

Chapter 4

Implementation

4.1. Client

Every driver needs a specified endpoint that allows to create a connection to the database. In the DataStax code, the `Client` class represents that connection and all requests are executed through it. Similarly, in the Rust driver, a `Session` represents a connection to the database. In our implementation, the JavaScript `Client` creates an instance of the Rust `Session` class and delegates all the communication with the database to it.

4.1.1. Client options

During the creation of the `Client` instance, the user can provide a number of options that define how the driver behaves. These include options like contact points, keyspaces or application name. Implementing them was straightforward as they are the same in both drivers. Some options required more effort like load balancing policies, encoding configuration or protocol options. Additionally, there are a few options that are present in the DataStax driver, but not supported by the Rust driver. These include some cloud or DataStax-specific options. We marked them as not supported.

4.1.2. Shutdown

The DataStax driver allows to shutdown its current connection, whereas the Rust driver does not support closing a `Session` once it is opened — it remains active until the program terminates or the variable representing the connection is dropped out of context.

We decided to deprecate the ability to explicitly close a connection to a database. To maintain compatibility with the DataStax driver API, calling `shutdown()` on a client is still supported. However, its functionality is now limited: it only prevents the execution of new requests using a given client instance. Compared to the DataStax driver it does not close the connection to the database, deallocate any structures related to that connection or stop any requests already in progress.

In the new driver, a connection is closed when the client variable associated with this connection is collected by the garbage collector.

4.2. Requests

The main interface for executing requests in both the DataStax and the new driver is the `Client.execute()` endpoint. This method executes a single request and returns a `ResultSet` object representing the request result.

When executing requests, developers of the Rust driver recommend explicitly specifying request result types and parameters before compilation. However, since we are developing a driver for a dynamically typed language, we cannot specify these types in advance. Fortunately, there are some ways to solve this problem.

The Rust driver exposes a `CqlValue` type — an enum that stores both the type of a value and the value itself. This allows storing any valid database value without needing to know its type at compile time. We used a Vector of `CqlValue` objects to represent request parameters or a single row in a result.

4.2.1. Prepared statements

When executing a prepared statement, we first call the database to prepare it. The returned object contains information necessary to execute the statement and the types of values expected for that statement. We use this information to parse the provided parameters into values that match the expected CQL types.

Because of the differences described in Chapter 2 we needed to implement some form of caching of prepared statements.

Caching session

Rust driver implements a mechanism that perfectly suits our needs — the caching session. `CachingSession` is a wrapper around the `Session` class provided by the Rust driver. If a statement has been prepared previously, `CachingSession` returns the cached object instead of preparing it again. This improves performance when executing the same statement multiple times and ensures proper use of prepared statements in the new driver. To avoid excessive memory usage, `CachingSession` limits its cache size to a predefined value. This limit can be set by the user using the `maxPrepared` client option that also exists in the DataStax driver.

4.2.2. Unprepared statement

While the Rust driver performs some preparation even for unprepared statements (for type validation purposes), the process is invisible to the Rust driver users. We cannot get any information about the types expected for a given statement. To address this we can use type guessing or type hints. Similarly to the DataStax driver, the user can use both methods in the same request.

Type guessing

Our goal was to create an interface for guessing the types in unprepared statements, similar to the one in the DataStax driver. In the DataStax driver, the encoding allows to determine the appropriate CQL type that should be inserted into the database, based on the type of the provided object. We implemented the same mapping as in the old driver, though with some limitations.

The approach used in the DataStax driver allows to use different CQL types interchangeably if they have the same encoding in the CQL protocol. An example pair of such types is `Uuid` and `TimeUuid`. When using the default `CqlValue` type in the Rust driver, we must precisely determine the type of each value provided. The current version of type guessing in our driver suffers from this limitation.

We added thorough documentation for type guessing, which was almost non-existent in the DataStax driver. We are aware of the limitations of the current approach, but the use case for the unsupported type guessing is limited. Improvements to this module are marked as future plans.

Type hints

The DataStax driver allows hints to be provided as either a string or a hint object. A hint object (including string representation) can be full, or partial. A hint is considered full when it specifies either a simple type (like int or date) or a complex type (like a list or a map) with all of its subtypes specified. A hint is considered partial if it defines a complex type without specifying hints for its subtypes. Providing hints for only some subtypes — such as specifying the type of map keys but not the values — is not considered a valid hint.

When a partial hint is provided, the driver attempts to guess the remaining part of the type of that value. Although this behavior is implemented in the DataStax driver, it is not documented anywhere. The description of type hints is based on an analysis of the source code.

In our implementation, we stayed close to the original approach. We reused the logic for converting a string hint to a hint object and implemented a conversion from a hint object into `ComplexType` — an internal type used for representing string-based `CqlValue` types.

We used NAPI-RS `Object` to pass hint values from JavaScript to Rust. This minimized the need for converting hint objects before passing them to Rust. There was one exception: in JavaScript, a value can be either a single value or an array of values, requiring us to standardize this representation before passing it to Rust.

4.2.3. Paging queries

In DataStax driver all queries are paged. The driver exposes several interfaces for iterating over result pages:

- `pageState` attribute in `ResultSet`. This attribute can be used to create a new query that retrieves the next page of results.
- Asynchronous iterator over `ResultSet`. This iterator allows to iterate over all response rows. The driver is responsible for executing the queries needed to fetch subsequent pages.
- `Client.eachRow()` endpoint. This endpoint allows specifying callbacks which will be called on each row and, optionally, on each page. Depending on the `autoPage` query option, it may be either the user's or the driver's responsibility to issue queries for subsequent pages.
- `Client.stream()` endpoint. This endpoint creates a stream and pushes each row into it as soon as it arrives. Internally, it uses `Client.eachRow()` to accomplish this.

In the new driver, we implemented support for all of these endpoints. When executing paged queries, we use the `(query/execute)_single_page` endpoint from the Rust driver to fetch a single page of results. We then attach the page state information to the result, enabling the execution of follow-up queries to retrieve subsequent pages. Additionally, we extended the driver interface to support unpaged queries through the `Client.execute()` endpoint, by setting the `unpaged` query option.

4.2.4. Concurrent execution

The DataStax driver exposes an endpoint that allows for concurrent execution of statements. There are two executors for this endpoint: `ArrayBasedExecutor` and `StreamBasedExecutor`. The first executor takes an array of statements to be executed, while the second one uses a stream, allowing driver users to input statements dynamically.

While in the DataStax driver those executors worked by calling `Client.execute()` endpoint, in our implementation we introduced several optimizations, related to reducing the usage of the NAPI-RS layer.

4.2.5. Statement options

Similarly to Client options, the user can provide a number of options that define how the statement execution behaves. These include options such as the page size, hints or statement timestamp. Again, implementing some of them was straightforward (such as page size), as they are the same in both drivers, while others (such as hints) were used in the new part of the code and were not passed to the Rust driver. Several options were impossible to implement, as the Rust driver does not allow control over certain parts of statement execution. We marked them as not supported. This includes the host option, which was available in the DataStax driver but its use was strongly discouraged. During the creation of the `Client` instance, the user can provide default statement options. This means that if certain option fields are not specified for a given statement, the default values will be used.

4.3. Types

4.3.1. Types decoding

Type decoding — also known as deserialization — is performed as part of parsing responses from the database. In our driver, we can distinguish two stages of type decoding: from raw bytes to Rust objects, from Rust objects to JavaScript objects.

Deserialization from raw bytes is done by the Rust driver. Our role is limited to specifying the expected data format and passing the values between the Rust driver and the user of the new driver.

Passing data from Rust to JavaScript required us to implement a solution based on the limitations of the NAPI-RS library. A core part of this process involves converting data from a `CqlValue`, provided by the Rust driver, into appropriate JavaScript objects as `CqlValue` may hold one of multiple types. When returning a `CqlValue` we convert it into a `value` or tuple of elements: `(cqlType, value)`

A plain `value` is returned when the type is known and there is no ambiguity in how it should be interpreted. A tuple is returned when there is uncertainty about how the value should be provided to the driver user.

For example if we return a `string`, no matter if it's `CqlAscii` or `CqlText`, we treat it the same and do not need to specify the `CqlType`. On the other hand, both `CqlBigInt` and `CqlTimestamp` are returned from the Rust layer as `BigInt`, but are provided to the driver user as `BigInt` and `Date` respectively.

4.3.2. Types encoding

The DataStax driver allows values to be provided in multiple formats:

- one of the predefined types,
- as a string representation of the type,
- as a pre-serialized byte array.

Since the DataStax code lacked any documentation on accepted formats of values, during our implementation we deduced this information based on the source code and the integration tests of the DataStax driver.

Encoding from string

Following types can be represented in a predefined text format:

- `Timestamp`,
- `Inet`,
- `UUID`,
- `TimeUUID`,
- `LocalDate`,
- `LocalTime`.

We decided to keep this form of serialization in the new driver. However, these values can only be correctly converted from a string to the corresponding object type if the target type is explicitly known. The type of these values cannot be guessed.

In our code, we convert strings representing the supported types into appropriate JavaScript objects before passing them to Rust. This ensures consistency with values that are already provided as objects of the corresponding class.

The DataStax driver also allows `Double` and `Float` values to be provided as strings, but this feature was kept there only for historical reasons. Based on that, we decided to drop support for `Double` and `Float` serialization from a string.

Regarding integer types, according to the documentation, the DataStax driver allows to provide values as strings, but there is no logic in the source code to handle this case. Based on this, we decided not to implement the conversion from strings into integers when encoding these values.

Value handling

The CQL protocol allows values to be null or unset. A CQL `null` is represented by the `null` value in the DataStax driver and as `None` in the Rust driver. Since NAPI-RS converts JavaScript `null` to Rust `None`, we used this conversion to avoid writing unnecessary code. When it comes to unset, in the DataStax driver it is represented by the `types.unset` value. In the Rust driver, all parameters are wrapped in the `MaybeUnset` type, which allows to pass parameters as `Set(CqlValue)` or `Unset`.

In the DataStax driver, an already serialized value can be provided as a statement parameter. As this is not a common use case, we did not implement this functionality.

While it is technically possible with the current state of the Rust driver — with the use of the `SerializeValue` trait — we left this for future consideration, whether this feature should be implemented.

Wrappers

The DataStax driver provides built-in support for decoding CQL types into custom JavaScript classes, enriching them with helper methods that simplify interaction with the database. Table 4.1 presents all CQL-supported types, along with information on whether a dedicated JavaScript class is available for a given type in the DataStax driver.

In the DataStax driver, some values could be modified even though they were marked as read-only in the API. The new driver enforces these read-only constraints and throws an error when an attempt is made to modify such a value. This adjustment was necessary because the actual data is now stored and managed on the Rust side.

We replaced the internal data of the original JavaScript classes with the appropriate wrappers. These classes are exposed by NAPI-RS and wrap the serialized CQL values. Inner logic of the classes was modified to operate on these wrappers. In this solution, serialization from CQL format to a usable format for the user occurs on the Rust driver side. This takes advantage of Rust's speed over JavaScript. and means that the data is fully managed on the Rust driver side.

Although this approach ensures faster value conversion to CQL protocol byte representation it has other performance drawbacks. Because the data is kept on the Rust side, every access to it from JavaScript results in additional overhead. To avoid it we could store the Rust objects on the JavaScript driver side. We implemented it only for some of the types while the rest is planned as a future optimization.

4.3.3. NAPI-RS object

When creating NAPI-RS objects we can either expose Rust structs to JavaScript code with `#[napi]` or with `#[napi(object)]` macros (an example is shown in Listing 4.1). There are some differences in how these exposed objects behave.

Structs exposed with `#[napi(object)]` tag can be created in either Node.js or Rust. When passing a value from Node.js to Rust, the object is cloned, meaning the original value stored in Node.js cannot be modified. Objects can only be passed to Rust by value. Their TypeScript type definition is an interface (an example is shown in Listing 4.2).

In contrast, structs exposed using `#[napi]` tag, can only be created in Rust. When the value is created and passed to Node.js, its ownership is permanently moved to Node.js. Such value can then be passed back to Rust only by reference. This allows modifications of the value, which updates the value stored in Node.js. Their TypeScript type definition is a class.

This is just a brief comparison of objects vs classes. More information can be found in different parts of NAPI-RS documentation. Due to performance reasons and other functionalities, we stuck to creating classes, unless object-specific features were required.

CQL type	Dedicated JavaScript class
Ascii	-
Boolean	-
Blob	-
Counter	-
Decimal	BigDecimal
Date	LocalDate
Double	-
Duration	Duration
Empty	-
Float	-
Int	-
BigInt	-
Text	-
Timestamp	-
Inet	InetAddress
List	-
Map	-
Set	-
UserDefinedType	-
SmallInt	-
TinyInt	-
Time	LocalTime
TimeUuid	TimeUuid
Tuple	Tuple
Uuid	Uuid
Varint	Integer[1]
Custom	-

Table 4.1: CQL types and their dedicated JavaScript class in the DataStax driver.

[1] `Integer` class will be deprecated in the future and replaced with the native `BigInt` type.

Listing 4.1: Example object and class declaration in Rust

```rust
#[napi(object)]
pub struct Ymd {
  pub year: i32,
  pub month: i8,
  pub day: i8,
}

#[napi]
pub struct TimeUuidWrapper {
  uuid: UuidWrapper,
}
#[napi]
impl TimeUuidWrapper {
  #[napi]
  pub fn get_buffer(&self) -> Buffer {
      ...
  }
}
```

Listing 4.2: TypeScript types of these objects

```typescript
export interface Ymd {
year: number
month: number
day: number
}
export declare class TimeUuidWrapper {
getBuffer(): Buffer
}
```

4.4. Errors

Error handling in Node.js supports many error classes — both built-in and custom, which the DataStax driver heavily utilizes. Both integration and unit tests verified if an error of a specific class was thrown. This posed no problem when parts of the logic remained in JavaScript — for example when verifying the type of a value.

However, when an error was thrown in Rust code, we needed to convert it into a JavaScript error of the correct class. As NAPI-RS does not support throwing custom errors, we addressed this by encoding error type to the error message on the Rust side and then converting those encoded errors into correct JavaScript errors on the JavaScript side of the code. It behaves similarly to a decorator: it takes a function as an argument, executes it, and catches any thrown errors. We could not use actual decorators, as they are still in the experimental stage in Node.js. If the driver is refactored to TypeScript in the future — where decorators are supported — this function can be easily replaced with a proper decorator.

4.5. Testing

4.5.1. Unit tests

We have adapted a number of unit tests from the DataStax driver, mostly the tests for the classes representing specific CQL types.

We have also created a number of new tests. These tests focus mostly on the NAPI-RS layer. This includes tests ensuring proper serialization and deserialization of the values and correct error conversion.

4.5.2. Integration tests

Not all integration tests are passing in the current state of the driver, as some features are yet to be implemented. However, all tests for the implemented features are turned on and passing.

While testing our driver on GitHub Actions, we encountered some problems not arising directly from bugs in the code. Sometimes the integration tests failed in an undeterministic manner.

Tests timeouts

Sometimes tests failed with a timeout. This always happened during the `before` step for batch tests — the first executed test. We investigated this issue but have not found the root cause yet. We assume that this is a problem with the ScyllaDB CCM that we are using to run the tests.

Memory allocation failures

Sometimes tests failed with a memory allocation failure. After some investigation, we found out that the issue is caused by a bug in the NAPI-RS library. (see Section 5.1.4) As the fix for this issue is not yet available, we worked around this problem, either by stopping concurrency or by rewriting code, so it does not use vectors of references as arguments. The first approach significantly slows down the driver so it was only used as a temporary solution while we implemented the second one.

4.5.3. TypeScript tests

In the DataStax code, there are some TypeScript tests to ensure it can be used in TypeScript projects. They verify if a correct API can be created from JavaScript code. They were modified to ignore features that are not supported by the new driver and switched on. They are passing in the current version of the driver.

4.6. Upgrading the Rust driver

As the Rust driver is an actively developed project, there were new versions released during the development of our driver. This meant we needed to update our code to be compatible with the new version.

4.6.1. Upgrade from version 0.14 to 0.15

With version 0.15 new deserialization API was introduced. This meant that we needed to update parts of our code that were responsible for handling rows and columns deserialization.

4.6.2. Upgrade from version 0.15 to 1.0

In March, after almost 4 years of development, version 1.0 of the Rust driver was released. This version was a major release, meaning that it introduced breaking changes. At the same time, it is a stable version, which means that the API is not expected to change before the next major release. Fortunately, the only change that affected our code was a reconstruction of Rust driver modules and some enums refactors. This meant that we needed to update most of the import directives in the code, as well as some minor changes to the handling of `ColumnType` enum in our code.

4.7. Future plans

During our project, we developed two versions of the new driver. The first iteration focused on implementing core functionalities to demonstrate the project's viability. The second iteration extended the driver with additional features and improved performance. In the thesis we describe the final result. There are still many improvements that can be made to the existing modules, and a lot of features remain unimplemented. We have attempted to categorize and document these areas for future developers. In this section, we describe the most important ones.

4.7.1. Client options

We implemented the most essential execution client options, but many others are still not supported by the new driver. Implementing some of these would require enhancements to the Rust driver itself.

Recconection policy

If the driver loses connection to a node in the cluster, it attempts to reconnect. The DataStax driver allows specifying a reconnection policy to define how the driver should behave in such cases. It provides two default policies — `ConstantReconnectionPolicy` and `ExponentialReconnectionPolicy` — and an option to define custom ones. In contrast, the Rust driver currently uses a hardcoded exponential reconnection policy. Adding support for additional and custom policies is an open issue (see: Rust driver, issue #184). Once this functionality is implemented, it could also be integrated into the new driver.

Load balancing policy

A load balancing policy defines how the driver distributes requests across nodes in the cluster. The DataStax driver provides several built-in policies and also allows users to define custom ones, though the process is not well documented. The Rust driver, on the other hand, offers a documented mechanism for implementing custom load balancing policies but includes only a default policy by default. Mapping custom load balancing policies from JavaScript — where they are implemented as extensions of a base class — to Rust — where they are implemented as traits — is currently marked as an open issue.

Auth providers

An authentication provider defines how the driver authenticates with the database. The DataStax driver includes several built-in authentication providers and supports the creation of custom ones. In contrast, the Rust driver currently includes only a default provider — plaintext authentication — but also allows for custom implementations. While we have implemented support for plaintext authentication, mapping custom authentication providers between JavaScript and Rust is marked as an open issue.

TLS

Both drivers support TLS/SSL connections to ensure secure communication with the database. The configuration is similar, but not identical in the two drivers. Integrating this functionality into the new driver is marked as an open issue.

Other options

There are several other options, mostly related to controlling low-level aspects of database connections. These provide a high degree of flexibility, primarily because users are expected to fine-tune the DataStax driver for optimal efficiency. In contrast, the Rust driver handles most of this configuration internally. As a result, these options require further evaluation and will likely not be implemented in the new driver.

4.7.2. Request options

The most commonly used request options have already been implemented, but some of them are still not supported.

Idempotence and retry policy

The DataStax driver allows specifying whether a request is idempotent, meaning it can be safely retried without side effects. This is closely tied to the retry policy, which defines how the driver should respond when a request fails or times out. The DataStax driver includes a specific retry policy that considers request idempotence, while all default policies in the Rust driver account for it by design. Both drivers support custom retry policies — mapping them is marked as an open issue.

Other options

There are several smaller options, some related to logging or metadata, that have not yet been implemented. Additionally, some options for custom request routing will likely not be implemented, as the Rust driver already includes a built-in mechanism for handling routing.

4.7.3. Metadata

The drivers include built-in mechanisms for retrieving and storing metadata. Some metadata, such as cluster, keyspaces, and table information, is shared by both drivers. Other metadata is specific to the database, like the token ring information in Cassandra or the shard information in ScyllaDB. The investigation into this metadata — determining what should be removed, retained, or added — is marked as an open issue.

4.7.4. Logging

An important feature of both drivers is the ability to log request execution. Currently, the systems are not integrated. In the future, the logging performed by the Rust driver should be made available to users of the new driver through appropriate endpoints. The extent of changes required to the original DataStax driver logging API has not yet been investigated. Adding support for this feature is marked as an open issue.

Chapter 5

Open-source contributions

During the development of the driver, we have made several contributions to other open-source projects. We have proposed bug fixes for some issues and opened bug tickets for other problems we encountered. Most of these contributions were related to the NAPI-RS project, which is a crucial part of our driver. We have also found and reported a bug in rust-analyzer — a tool used to provide IDE support for Rust.

5.1. NAPI-RS

5.1.1. BigInt

When running unit tests for the Duration class, we encountered test failures related to handling large numbers. Specifically, when passing negative numbers that are exactly 8 or 16 bytes long from Node.js to Rust — where they are converted to i64 and i128 respectively — we observed different values on the sending and receiving ends. Additionally, attempts to return `i64::MIN` or `i128::MIN` caused the Rust code to panic.

After investigation, we found out that the issue was arising from the NAPI-RS code, not our own. We fixed a number of different bugs that caused these issues and proposed these changes in a Pull Request to the library (see `https://github.com/napi-rs/napi-rs/pull/2356`). These changes were quickly merged and after some time they were also backported to version 2 of the library (see `https://github.com/napi-rs/napi-rs/pull/2524`).

Interestingly, the library already had unit tests that should have caught at least some of these issues, but for unknown reasons, the tests were expecting incorrect values. As part of our contribution, we corrected these tests and added new regression tests to ensure that the fixed issues would not reoccur in the future.

5.1.2. Tuple as a function argument

During development, we encountered inconsistency regarding passing a tuple between Node.js and Rust. Tuple implemented the `FromNapiValue` trait, which allows it to be passed to Rust code, but `ToNapiValue` was not implemented, meaning it was not possible to return it from Rust code back to Node.js.

After investigating why only one of these traits is implemented, we found out that tuple was used for passing arguments to callbacks — Node.js functions that can be called from Rust parts of the code.

After internal discussion, we decided to propose changes to the NAPI-RS interface. Our idea was to add a new custom type, used to provide arguments for callbacks and implement

ToNapiValue for tuples.

On top of these changes, we updated FromNapiValue from existing manual implementation for each acceptable length of tuple, into a macro that implements it automatically up to the specified length.

We have opened a Pull Request with these changes. They were accepted by the maintainers of the NAPI-RS repository and became part of the NAPI-RS in version 3.

We were unable to use this change in our driver, as version 3 of the library is still in development (the library maintainers plan the release for August 2025). However, these changes will be useful in a number of places in the future development of the driver.

5.1.3. Null values inconsistencies

During implementation, we encountered an undocumented behavior of the NAPI-RS library. The NAPI-RS Object allows fields to be empty. In JavaScript, a field can be set to null or not provided (represented as undefined). After some experimentation, we found out that fields which are not provided are interpreted as Rust None and fields set to null are read in Rust as Some(None) (an example is shown in Listing 5.1). This means that to handle all cases where a field is not being provided, we need to expect double nested Option type in Rust.

This is both undocumented and inconsistent with how Option works for function arguments. When calling a function, passing an argument as null or undefined results in a None value in Rust.

Listing 5.1: Inconsistent behavior example

```
// Rust part of the code:
#[napi]
pub fn fun1(arg: Option<i32>){}

#[napi(object)]
pub struct Example{
  pub v1: Option<i32>,
  pub v2: Option<Option<i32>>
}
#[napi]
pub fn fun2(arg: Example){}

// Node.js code - Object received in~Rust as comment:
fun1(); // None
fun1(null); // None
fun1(1); // Some(1)

fun2({}); // {v1: None, v2: None}
fun2({v1: 6, v2: 9}); // {v1: Some(6), v2: Some(Some(9))}
fun2({v2: null}); // {v1: None, v2: Some(None)}
fun2({v1: null}); // Invalid code
```

Unfortunately, due to a lack of time, we were unable to propose any changes that would ensure consistency of the behavior. However, we reported this issue to the NAPI-RS maintainers.

5.1.4. Use-after-free

As mentioned in Chapter 4, we encountered some problems with memory failures in integration tests. This section describes the process of identifying the cause of this issue.

After investigating the problem, we determined that the problem was arising from concurrent tests (tests in `concurrent/execute-concurrent-tests.js` file). There was no single test that was causing a failure. These problems were happening only when all of the concurrent tests were enabled and at least some of the tests from other files. Additionally, the test execution order affected whether the issue occurred — the problem only manifested when the concurrent tests were executed last. These failures were happening only on GitHub Actions — we were unable to recreate this problem on any of the local machines.

While we quickly determined that disabling the concurrency for `executeConcurrent` function solved the issue, we also wanted to find the root cause of the problem. As we were unable to recreate this bug on a local machine, we used breakpoint-action [5] to connect through SSH with a machine that was running tests as part of CI.

After launching the tests directly on these machines we were able to recreate the issue. We determined that the issue was caused by cloning a string with an invalid size — this appeared to be some random 64-bit value. For this reason, sometimes tests failed with the error message shown in Listing 5.2 (with a different value during each run). And other times they failed with the error message shown in Listing 5.3.

Listing 5.2: Error message 1

```
memory allocation of 3376683291075676088 bytes failed
Aborted (core dumped)
```

Listing 5.3: Error message 2

```
thread 'tokio-runtime-worker' panicked at library/alloc/src/slice.rs:161:25:
capacity overflow

thread 'tokio-runtime-worker' panicked at library/alloc/src/slice.rs:161:25:
capacity overflow
Segmentation fault (core dumped)
```

We attempted to launch the program under gdb to find more information about the issue. After some attempts, we were unable to start the integration tests under gdb, as loading JavaScript libraries posed some undefined issues. Instead of launching the integration tests with gdb, we were able to attach gdb to initialized integration tests, before they were run. However, the issue did not manifest in this setting.

With this in mind, we switched the approach and ran the program normally. When the program panicked we caught this error and then froze the program (with a very long sleep). We then attached gdb to the frozen program so that we could investigate the state of the program during invalid memory allocation.

Investigation of the stack trace, when the program was built in debug mode, revealed that although the issue was caused by an attempted allocation of memory for a cloned string, the type of value that was cloned at this point was supposed to be UUID. While this was still important information, investigation in the gdb did not reveal any information about the cause of this issue.

After further investigation, we determined that the sleep in the Rust part of the code, just before the problematic part, increased the likelihood of failure. This led us to suspect that the issue was related to the Node.js garbage collector. After testing this hypothesis, with a manual invoking of the garbage collector, we were able to confirm it.

Knowing that the problem arises from the behavior of the Node.js interpreter, we shifted our attention to the Node.js version, that was used locally and on GitHub Actions. When testing locally we were using either Node.js in version 22 or 24, while version 20 was used on CI machines. After manually downgrading the Node.js version on local machines we were able to recreate this issue locally.

Having determined the issue, we were able to create a minimal reproducer of this problem. This led us to the following conclusion regarding the root cause of the issue: when passing a value to Rust using NAPI-RS, Node.js does not count this as a reference to the object (see `https://napi.rs/docs/compat-mode/concepts/ref`). In a single-threaded program, this causes no issues, but when concurrency is used, this may lead to dropping of the value, while it's still used in Rust code. This can be avoided by manually declaring the variable as used. When executing Rust async functions (like in `https://napi.rs/docs/concepts/async-fn`), it's the responsibility of the NAPI-RS layer to ensure the value is not dropped. This works correctly when a reference to the value is provided, but it fails if the array of references to values is given.

After identifying the issue, we reported it to the NAPI-RS maintainers. Although it has since been fixed, the fix is not yet available in the latest stable version of the library at the time of writing this thesis.

5.2. rust-analyzer

During development we encountered some problems with rust-analyzer — a front end to the Rust compiler, that allows the code editor extension to analyze the edited code. We narrowed down the issue to the code snippet shown in Listing 5.4.

Listing 5.4: Self-referencing struct

```
#[napi]
pub struct SomeType{
  pub(crate) field: SomeType,
}
```

We have opened an issue at the Rust analyzer repository [19]. Due to its severity this issue was fixed, by the maintainers, the same day it was reported, compared to most of the issues which stay open for days. The issue turned out to be a stack overflow when computing the size of a struct that includes itself as the tail field.

Chapter 6

Benchmarks

6.1. Types of benchmarks

We implemented four simple benchmarks: two benchmarks for requests that insert data into the database and two benchmarks for requests that read data from the database. There are versions implemented in JavaScript and Rust. The JavaScript version tests the DataStax driver and the new driver, while the Rust version tests the Rust driver. Time provided on the charts is the time elapsed on each benchmark.

For all benchmarks, we used a replication factor of one, meaning that each piece of data is stored on a single database node. While this is uncommon in production setups, we chose this configuration to minimize internal database communication. This setup highlights the impact of efficient driver-to-database communication, which would be less noticeable with higher replication factors — especially when all nodes run on a single machine.

6.1.1. Simple benchmark

In both insert and select versions the benchmark works in a similar way. Firstly it makes sure the database is empty. Then it creates a table with `uuid` and `int` columns. The insert benchmark then inserts a specified number of rows into the table. The select benchmark firstly inserts a specified, small number of rows into the table and then selects all of them a specified number of times. In this case the benchmark executes one request at a time, waiting for the result of each request before executing the next one.

6.1.2. Concurrent benchmark

The concurrent benchmark works similarly to the simple benchmark but it executes multiple requests at the same time.

In the JavaScript version, this is achieved using the `executeConcurrent()` endpoint. Since there is no direct equivalent of this function in the Rust driver, the Rust version emulates its behavior. It uses the Tokio library to create multiple threads that execute the requests concurrently. This approach was based on an analysis of the DataStax driver's source code, as the internal workings of `executeConcurrent()` are not documented.

When comparing memory usage between the two versions, it is important to note that the JavaScript benchmark requires all statements to be pre-defined in an array before calling the endpoint, whereas the Rust version generates the statements on the fly.

6.2. Database backends

Driver behavior depends on the setup of the database. All of the database instances were created through Docker. In case of multi-node setup all nodes were created on the same machine the driver was launched on. In the benchmarks we used the following database setups for main benchmarks:

- `Three-node ScyllaDB with one shard per node. Tablets enabled`,
- `Three-node Cassandra`.

Additionally, benchmarks were run on the following database setups, to see the performance impact of shard and tablet awareness:

- `Three-node ScyllaDB with two shards per node. Tablets disabled`,
- `Three-node ScyllaDB with one shard per node. Tablets enabled`,

6.3. Running tests

As part of the development, we run the benchmarks for each of the introduced features, to see their impact on the performance. Those benchmarks are executed via GitHub Actions. A Python script runs the benchmarks on a dedicated server we host, ensuring consistent hardware for all tests. Three drivers are compared based on execution time and memory usage: the Rust driver, the DataStax driver and the new ScyllaDB Javascript Driver. The script benchmarks each driver using varying numbers of operations. For each configuration, the benchmark is executed five times. The results are then visualized using Matplotlib, and the generated plots are uploaded to a Discord server.

Benchmarks on GitHub Actions are run only on `Three node ScyllaDB with two shards per node. Tablets enabled.` version, the benchmarks presented here use all of the listed database setups.

6.4. Results

Across all benchmarks configurations, the new ScyllaDB Javascript Driver outperforms the DataStax driver for some of the tested values of n (see Figure 6.1). The specific values of n where this advantage appears vary between benchmarks, and the performance gap narrows as the number of executed requests increases. This suggests that the Rust-based driver establishes a connection to the database a few seconds faster than the DataStax driver.

As the number of requests increases, the Rust driver continues to outperform the DataStax driver. However, despite several optimizations (see Section 6.4.3), the new driver still consumes significant CPU time due to the overhead of passing values through NAPI and NAPI-RS thread synchronization. As a result, the DataStax driver eventually surpasses the new driver when executing a larger number of requests.

The new driver's CPU usage is approximately 50% higher than that of the DataStax driver, which means the point at which the new driver is overtaken may vary depending on the hardware. On faster machines, the threshold number of requests where both drivers exhibit similar performance will likely be higher. Despite the overall higher CPU usage, the time advantage observed with the new driver arises from efficient use of multithreading in Rust and NAPI-RS.

When tested on Cassandra (see Figure 6.2), the value of n for which the DataStax driver surpasses the new ScyllaDB Javascript Driver is smaller compared to the benchmarks run on ScyllaDB. This difference comes from lack of database features that tne new ScyllaDB Javascript Driver can use to stay ahead. Due to the different database capabilities to sustain concurrent writes, we were unable to finish some of the benchmarks with higher n values. As a result we present only first 3 steps, with the same starting values as benchmarks run on ScyllaDB.

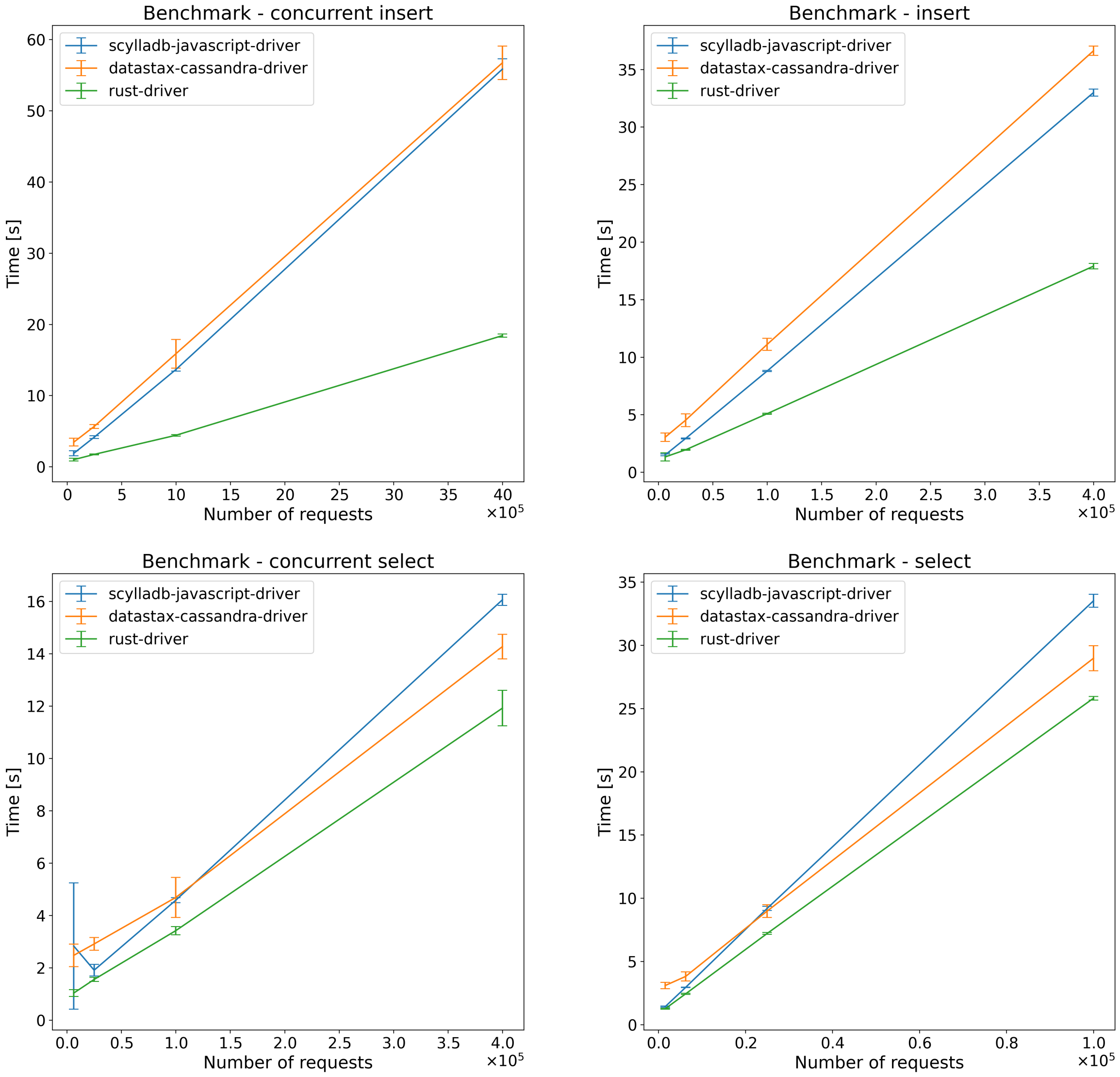


Figure 6.1: Results of benchmarks run on 3-node ScyllaDB. Comparison of our ScyllaDB Javascript Driver, the Datastax Cassandra Driver and the Rust Driver.

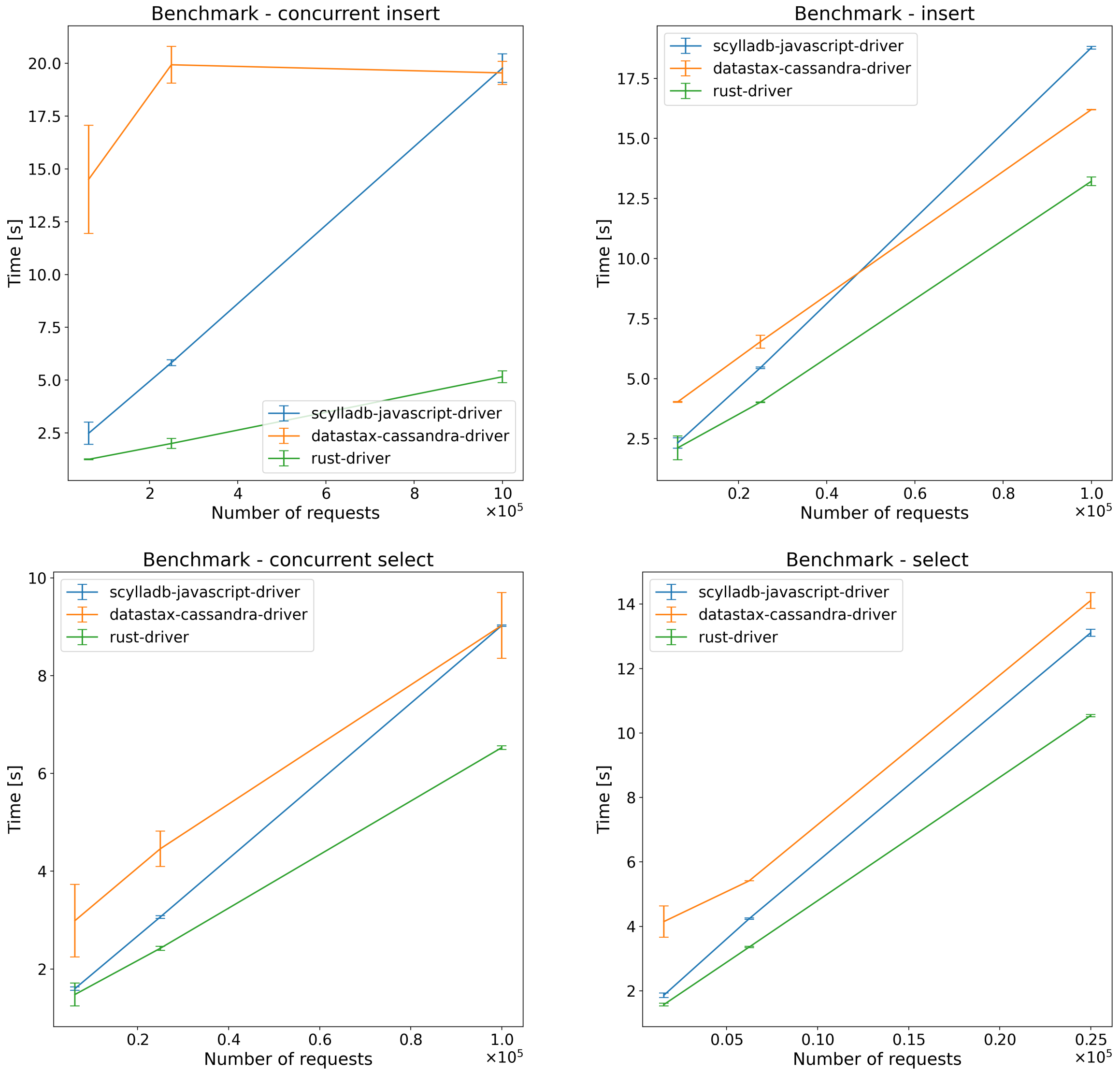


Figure 6.2: Results of benchmarks run on 3-node Cassandra. Comparison of our ScyllaDB Javascript Driver, the Datastax Cassandra Driver and the Rust Driver.

6.4.1. Tablets

For the insert benchmarks (Figure 6.3) we can see slight improvement of the new ScyllaDB Javascript Driver with the tablets enabled, the benchmark execution time for the largest number of requests increased by more than 44%, while the performance of the DataStax driver stays the same with or without tablets. This is expected, as the Rust driver and by extension the new ScyllaDB Javascript Driver is aware which node will save the given insert request. This way the database nodes do not need to send the requests between each other, as it's the case for the DataStax driver. This behavior can be confirmed when looking at the network communication — Section 6.4.2.

For the select benchmarks all drivers saw a very drastic performance drop. We asked ScyllaDB employees what is the reason for those results, but at the time of writing this thesis we did not receive any information about that.

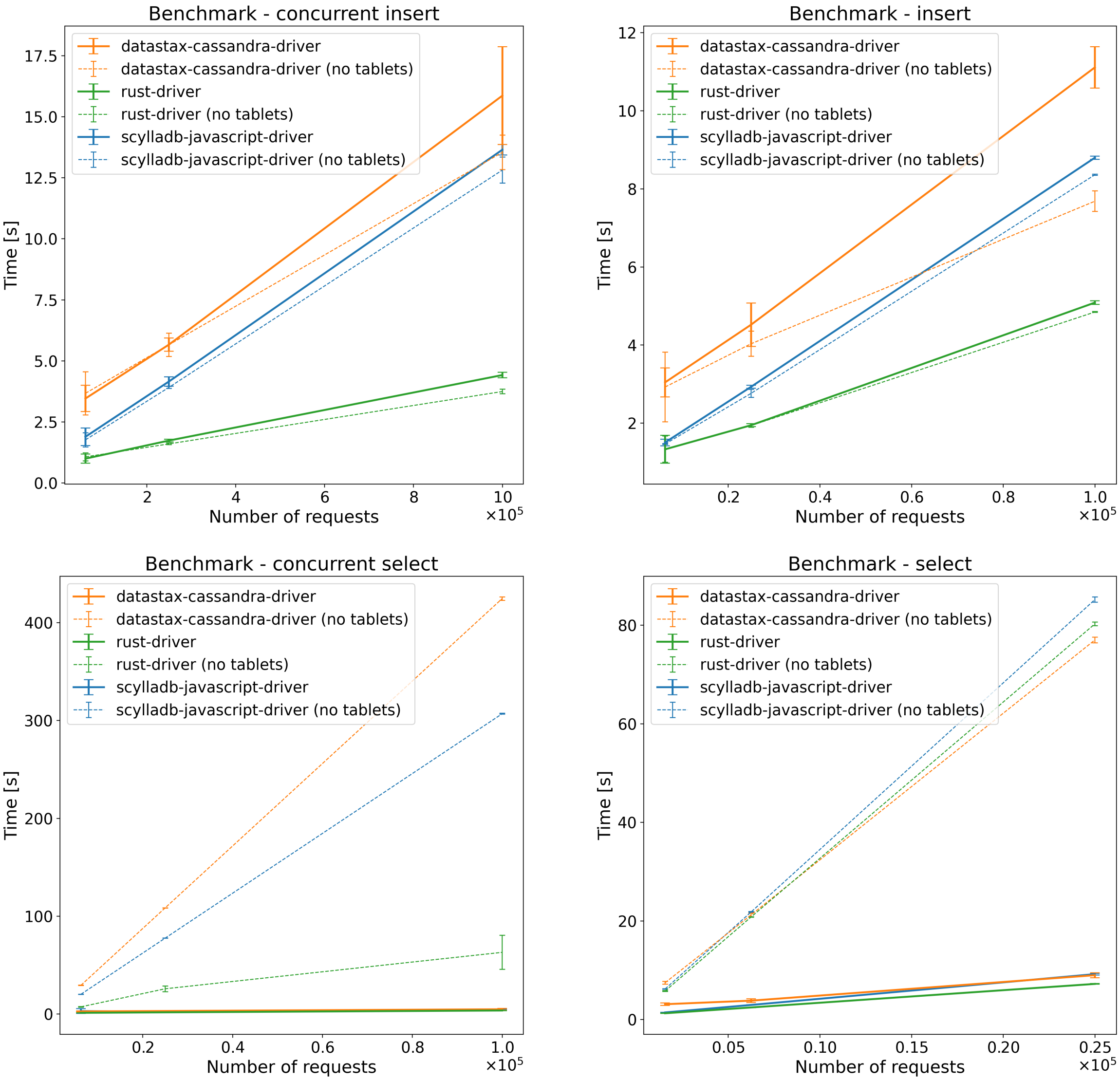
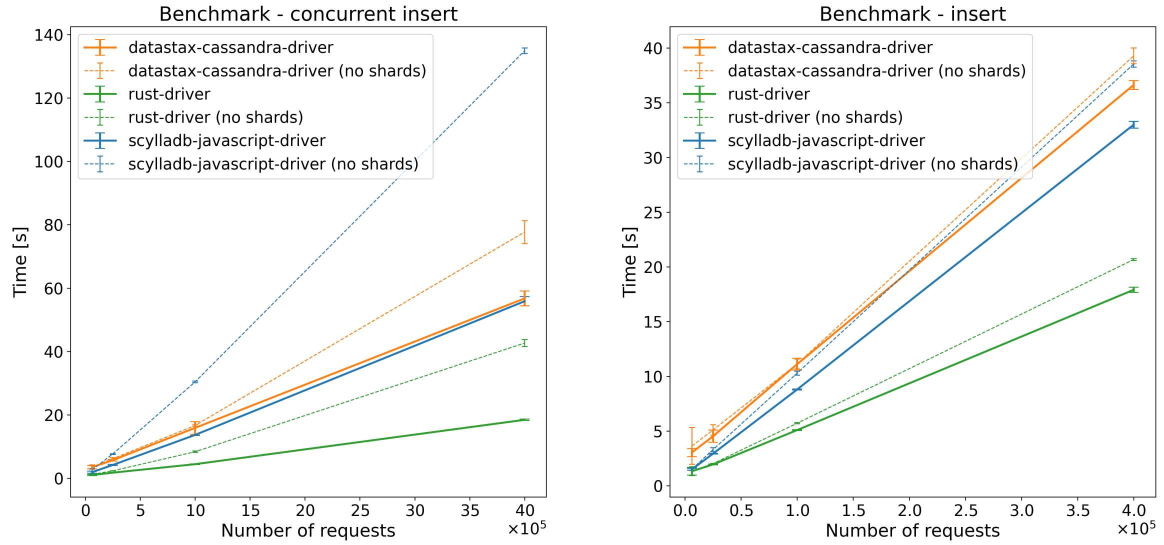


Figure 6.3: Comparison of the drivers with tablets vs without.

6.4.2. Shards

All drivers showed improved execution times in insert benchmarks when using shards. The Rust driver improved by around 36%, the new driver improved by around 46% and the DataStax driver improved by only around 10%. These results were observed on a concurrent insert benchmark using a three-node database with shards — see Figure 6.4a. Similar results were observed on other insert configurations, although with different percentage values — see Figure 6.4b.

These results show that while sharding provides some performance benefits for the DataStax driver, the new driver benefits significantly more — achieving performance improvements comparable to the Rust driver.



(a) Comparison of the drivers with shards vs without - concurrent insert benchmark.

(b) Comparison of the drivers with shards vs without - insert benchmark.

Figure 6.4: Comparison of the drivers with shards vs without.

6.4.3. Network usage

As part of our benchmarks, we measured network usage with Wireshark. All of network usage analysis was made with concurrent insert benchmark with a fixed number of requests ($n = 100000$). We measured both the communication between the driver and the database as well as the communication between database nodes. We tested network usage using two different database setups: single-node ScyllaDB and three-node ScyllaDB with 2 shards per node. The results of the comparison of the number of packets transmitted are shown in Table 6.1.

WHAT	**TOTAL**	**CQL**	**TCP**	**Total Size**
New driver 1 node	210,264	12,072	198,192	∼ 21.6 MB
DataStax driver 1 node	10,697	8,958	1739	∼ 9.2 MB
New driver 3 node all	412,764	112,318	300,446	∼ 43.7 MB
New driver 3 node | driver ↔ database	409,678	112,318	297,360	-
New driver 3 node | node ↔ node	3,086	0	3,086	-
DataStax driver 3 node all	268,037	45,052	222,985	∼ 81.2 MB
DataStax driver 3 node | driver ↔ database	90,978	45,052	45,926	-
DataStax driver 3 node | node ↔ node	177,059	0	177,059	-

Table 6.1: Summary of the number of packets and data transmitted over the CQL/TCP protocols for different node configurations.

Both the Rust driver that we use and the DataStax driver use coalescing – single CQL packet to send multiple requests. Due to different implementation of coalescing, drivers send different numbers of packets. Higher coalescing improves throughput, as less space is required for packet headers and at the same time increases latency, as it takes longer to form a single CQL packet to be sent.

In the single-node case, the DataStax driver had around 25% less CQL packets, around

95% less TCP packets and around 47.5% less bytes transferred. While we were unable to investigate specific reason for this disproportion, according to internal discussion with the Rust driver developers this disproportion in TCP packets is an expected difference in coalescing behavior.

In the three-node database case TCP packet count is still 35% lower for the DataStax driver, total bytes transferred is 85% higher than in the new driver and the number of packets sent between database nodes (internal packets) is 57 times higher. The number of internal packets observed for the new driver is similar to the number of packets sent between the database nodes when idle.

Based on this data, we can conclude, that the new driver, thanks to tablet awareness, is aware of the expected data layout in the database and, as a result, can reduce network communication necessary to execute requests.

We can also observe that the number of CQL requests sent by the DataStax driver is 60% lower compared to the new driver in the three-node case.

6.5. Driver optimization

Due to the lack of documentation on the efficient use of NAPI-RS, we had to experiment with various approaches to identify the fastest one. We used benchmarks as a tool to compare the performance impact of different changes. In Figure 6.5 we show the difference all of the implemented optimizations provide. We individually tested those optimizations to ensure that they speed up at least some of the benchmarks. All other benchmarks show the new ScyllaDB Javascript Driver already with optimizations.

6.5.1. FromNapiValue

When accepting values as arguments to Rust functions exposed in NAPI-RS, we can either accept values of the types that implement the `FromNapiValue` trait, or accept references to values of types that are exposed to NAPI — these implement the default `FromNapiReference` trait.

While the latter approach is easier to implement, it increases abstraction, resulting in significantly higher CPU usage by the NAPI-RS layer. We attempted to implement the faster of those approaches in places, which are used the most — during request execution.

ParameterWrapper

When retrieving a parameter on the Rust side, we must be aware of the `CqlType` of the value. Our initial approach exposed an endpoint for creating a value of a given type, which was then converted to a `CqlValue`, returned back to Node.js side as a wrapper. This wrapper was later passed as a parameter to a request.

In the updated approach, we pass parameters to requests directly, without wrapping or any other conversion steps. In order to correctly handle value types, each parameter is passed as a pair (type, value). Information about the type is then used on the Rust side with `FromNapiValue` to create correct `CqlValue` object.

ArgumentsWrapper

As described in Section 4.3.1, values for some of the types can be presented in multiple formats. Initially this conversion was done in the Node.js part of the code. In case of prepared statements, this required passing information about expected types from Rust to Node.js.

We created a draft version, in which we shift this responsibility to the Rust side of the code. While the draft version provides support for parsing only some of the types, it has promising results when it comes to the driver performance.

6.5.2. ToNapiValue

Similarly to accepting values as arguments, functions exposed from Rust via NAPI-RS can return values that implement the `ToNapiValue` trait. This trait can either have a default implementation for types that are already exposed to NAPI-RS, or it can be implemented manually for custom types.

While the default implementation works in most cases and correctly maps Rust values to their JavaScript equivalents, our testing revealed several situations where manually implementing the trait results in better performance. The most common case is when a type wraps another value and exposes an endpoint to access the underlying value after conversion.

RowWrapper

The `RowWrapper` type wraps a single row of a query result. It is created from the `Row` type — a struct exposed by the Rust driver. We tested two implementations of the `ToNapiValue` trait for this type.

In the first approach, we used the default `ToNapiValue` returning a `RowWrapper` object with a `get_columns` method. This method converted the row into an array of objects recognized by NAPI-RS.

In the second, faster approach, we implemented a custom `ToNapiValue` trait for the `RowWrapper` type. This implementation returned a JavaScript object with the same structure as the original row.

CqlValueWrapper

The `CqlValueWrapper` wraps a single `CqlValue`, storing both the type information and the value itself. We tested two implementations of the `ToNapiValue` trait for this type.

In the first approach, we used the default `ToNapiValue` trait returning a `CqlValueWrapper` object that exposed `get_type` and `get_...`(ascii, bigint, ...) methods for each of the possible types. When the value was retrieved on the JavaScript side, we first called `get_type` to get the type of the value and then we called the correct `get_...` based on the value type.

In the second approach, we implemented `ToNapiValue` trait for the `CqlValueWrapper` type. We could not return just the value itself, because some types had the same type on the Rust side, but they needed to be converted into different types on the NodeJS side. In our implementation, when there was no ambiguity, we returned just the value and when there was ambiguity about the types of the values, we returned a pair `(typ, value)`, as described in Types decoding Section 4.3.

After testing both implementations, we confirmed that the second implementation is faster — similarly to the situation with `RowWrapper`.

6.5.3. Other optimizations

Caching session

Implementation of caching session, as described in Chapter 4, significantly reduced network traffic in all benchmarks, (around 50% — it was not exactly 50% due to coalescing) when measured using Wireshark.

While the performance improvement in concurrent benchmarks was relatively minor — only a few percent — the difference in the plain benchmark was significant, with around a 50% improvement. This difference arises from how the benchmarks work: in the concurrent benchmark, the driver issues new requests while waiting for responses from previous ones, whereas in the plain benchmark, the driver remains idle while waiting for a response, making the overhead of statement preparation more noticeable.

Option reusing for concurrent requests

When executing requests using the `executeConcurrent` function, both the DataStax driver and the initial version of our driver created a new instance of `ExecutionOptions` for each request. Although the same options were used for all requests in a single call, a new instance was unnecessarily created each time.

After identifying this, we modified the code to reuse the same `ExecutionOptions` instance within a single call to `executeConcurrent`. Since creating this object involves a call through the NAPI-RS layer, reusing it significantly reduced overhead.

Object wrappers vs raw data

When creating custom types, we can keep the underlying data either in the Rust or Node.js part of the code. When keeping it in Rust, we create a wrapper over this value, which is used in the Node.js part, to expose its functionality.

While this approach allows easier data representation, we discovered that with NAPI-RS overhead, this approach is very slow. The alternative we tried, was keeping the raw data in the Node.js part. This requires having unified raw representation in both Node.js and Rust parts of the driver.

This approach is currently implemented for `UUID` and `TimeUUID` types.

6.5.4. Reducing function calls

In the current implementation, `executeConcurrent()` is implemented on the JavaScript side. As a result of this implementation, each request requires a call to NAPI-RS. We partially implemented request execution on the Rust side. With limited time, we were unable to finish this optimization and it's not present in the final benchmarks, but some testing suggests promising performance impact for concurrent benchmarks. The improvement here comes from reduced need for thread synchronization.

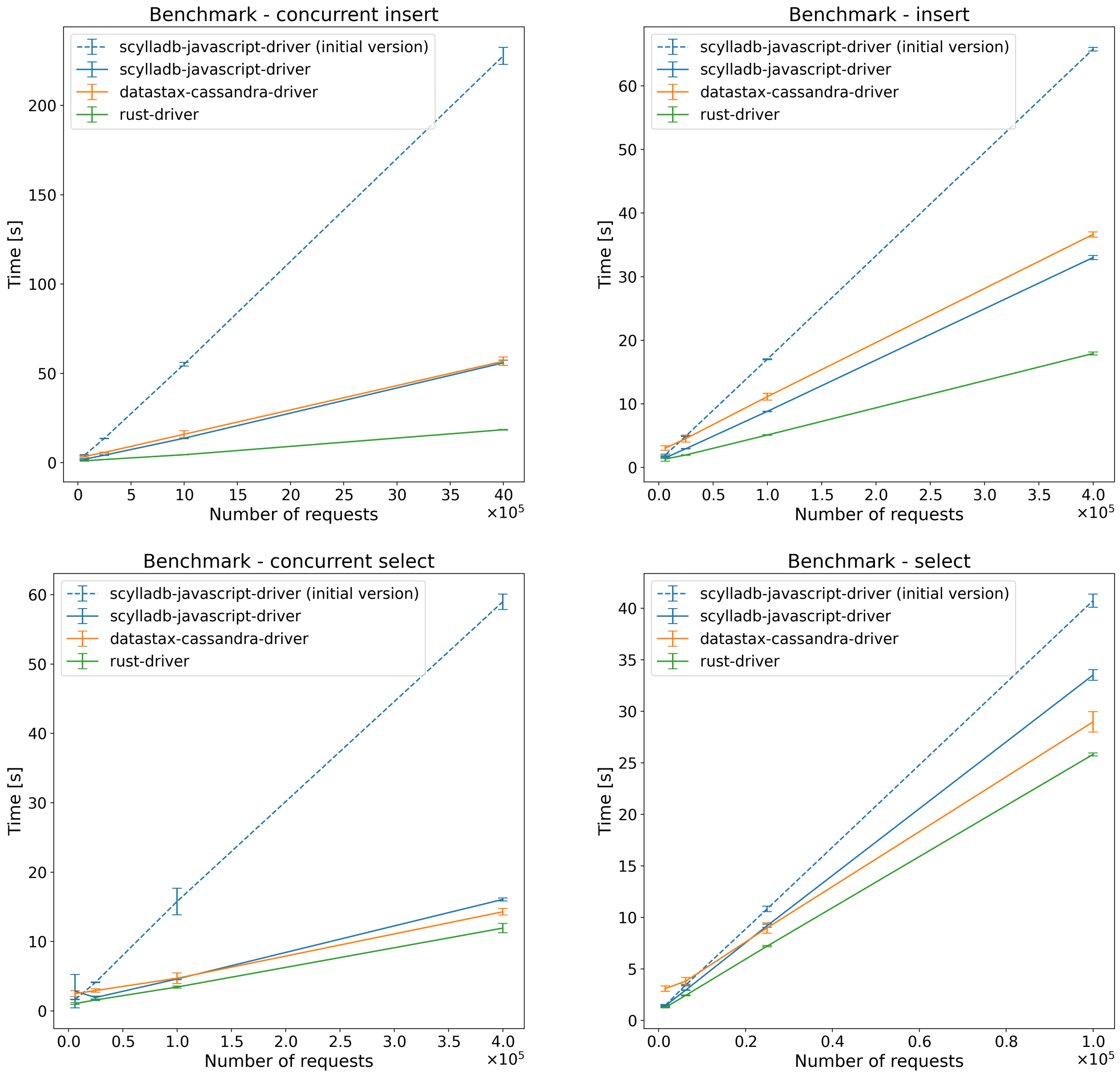


Figure 6.5: Results of running benchmarks on 3-node ScyllaDB. Comparison of our ScyllaDB Javascript Driver, the Datastax Cassandra Driver, the Rust Driver and the initial version of our ScyllaDB Javascript Driver.

Chapter 7

Summary

Our task was to create a new Javascript driver for ScyllaDB and Cassandra databases, based on the API of the existing DataStax driver. The goal was to use the existing Rust driver underneath.

7.1. Feature completeness

We implemented support for all endpoints responsible for executing requests that existed in the DataStax driver, including batch statements. We managed to implement all but one endpoint that allows retrieving the following result pages. Our driver supports all CQL types (as of version 4.0 of the CQL protocol) both as query parameters and results. The driver supports some of the query and execution options. The driver itself has no limitations on statement syntax, has support for tablets and shards. All of the implemented endpoints, with the few minor exceptions described on the GitHub page of the driver, have the same functionality as the DataStax driver.

In the current state, the new driver is a well refined proof of concept, showing the possibilities of creating a driver for the dynamically typed language as an overlay over the Rust driver. Due to the project size there is still number of features lacking for the driver to be considered production ready and a full replacement of the DataStax driver. All of the major missing features were described in Section 4.6.2.

7.2. Driver performance

We partially managed to achieve a goal of creating a faster driver. With the faster connection times, and the number of introduced optimizations, the new driver is faster in some of the created benchmarks compared to the DataStax driver. We consider such performance a partial success.

While we already implemented a number of driver optimizations, significantly reducing a percentage of time driver spends on parsing the data — from about 40% to only about 15% of CPU time — the driver still spends a lot of time in thread synchronization. Reducing time spent on synchronization is part of the future plans. This includes optimizations described in Section 6.5.3. We also came up with several ideas that could lead to further optimization. Those ideas include:

- shifting the thread synchronization from NAPI to the Rust part of the code,
- changing the default Tokio executor,

- experimentation with other Tokio library options.

When creating benchmarks we focused mostly on the main endpoints. As a result, other driver endpoints, mostly those related to paging, and CQL types with a dedicated JavaScript class, other than `UUID` and `TimeUUID`, lack the number of optimizations compared to the benchmarked parts of the driver. As part of future development, more focus should be placed on testing and optimizing those parts of the driver.

7.3. Division of work

Our effort was not divided into completely independent parts. Rather than that, we developed the project iteratively, splitting the work into small, manageable, separate tasks. Over the course of 8 months we created, revived, and merged 130 Pull Requests with over 320 individual commits.

We created over 2300 lines of Rust code, 7000 lines of JavaScript in the core part of the driver, 1000 lines for benchmarks, and 500 lines for examples. There are currently over 2000 lines of working unit tests, and almost 5000 of working integration tests. Almost all of the code has been worked on by multiple people — either through design, implementation, code review, bug fixing, or additional testing. As part of the ongoing tests, we run over 6000 individual workflows that took over 31000 minutes to complete. Integration tests had a 74% success rate, while unit tests had 92% success rate.

We believe that each author contributed a fair share to the combined effort. Below we list for each of the authors the parts of the project where they had a significant contribution:

Stanisław Czech

- implementation of all endpoints for query execution,
- adding support for paging,
- driver optimizations,
- adding support for CQL Duration type,
- benchmarks,
- driver documentation,
- unit and integration tests.

Piotr Maksymiuk

- adding support for CQL Inet, CQL Tuple, CQL Decimal, CQL Varint and UDT types,
- driver documentation,
- unit and integration tests.

Zuzanna Ossowska

- adding support for CQL UUID, CQL TimeUUID,
- error handling,
- analyzing future possibilities,

- driver documentation,
- unit, integration and TypeScript tests,
- adding support for client options.

Stanisław Polit

- adding support for CQL Time, CQL Date,
- benchmarks,
- driver documentation,
- unit and integration tests.

Bibliography

All links were visited on 12-06-2025.

[1] *About CCM.* https://www.datastax.com/blog/ccm-development-tool-creating-local-cassandra-clusters

[2] *Apache Cassandra.* https://cassandra.apache.org/

[3] *Apache Cassandra 4.0 Performance Benchmark Comparing Cassandra 4.0, Cassandra 3.11 and ScyllaDB Open Source 4.4.* https://lp.scylladb.com/cassandra-4.0-vs-scylla-benchmark-offer?siteplacement=scylladb-vs-cassandra

[4] *Apache Software Foundation.* https://www.apache.org/

[5] *Breakpoint action.* https://github.com/namespacelabs/breakpoint-action

[6] *Cassandra Query Language (CQL).* https://www.scylladb.com/glossary/cassandra-query-language-cql/

[7] *CCM transfer to Apache Incubator.* https://incubator.apache.org/ip-clearance/cassandra-ccm.html

[8] *CQL protocol specification.* https://github.com/apache/cassandra/blob/trunk/doc/native_protocol_v4.spec

[9] *DataStax.* https://github.com/datastax/nodejs-driver

[10] *DataStax CCM.* https://github.com/riptano/ccm

[11] *DataStax driver API documentation.* https://docs.datastax.com/en/developer/nodejs-driver/4.8/

[12] *DataStax support for CQL.* https://docs.datastax.com/en/dse/6.9/cql/cql-guide.html

[13] *Driver documentation.* https://scylladb-zpp-2024-javascript-driver.github.io/scylladb-javascript-driver/

[14] *GitHub Pages.* https://pages.github.com/

[15] *JavaScript language.* https://developer.mozilla.org/en-US/docs/Web/JavaScript

[16] *JSDoc documentation generator.* https://jsdoc.app/

[17] *NAPI-RS framework.* https://napi.rs

[18] *Neon library and toolchain.* https://neon-bindings.com/

[19] *Rust analyzer.* https://rust-analyzer.github.io/

[20] *Rust driver — release 0.14.* https://github.com/scylladb/scylla-rust-driver/releases/tag/v0.14.0

[21] *Rust driver — release 1.0.* https://github.com/scylladb/scylla-rust-driver/releases/tag/v1.0.0

[22] *Rust driver documentation.* https://rust-driver.docs.scylladb.com/stable

[23] *Rust driver documentation on paging.* https://rust-driver.docs.scylladb.com/stable/statements/paged.html

[24] *Rust driver for ScyllaDB.* https://github.com/scylladb/scylla-rust-driver

[25] *Rust language.* https://www.rust-lang.org/

[26] *Scylla CCM.* https://github.com/scylladb/scylla-ccm

[27] *ScyllaDB.* https://www.scylladb.com/

[28] *ScyllaDB shards.* https://www.scylladb.com/glossary/database-sharding/

[29] *ScyllaDB — Why tablets are used?.* https://www.scylladb.com/2024/06/13/why-tablets/

[30] *ScyllaDB — How tablets are implemented?.* https://www.scylladb.com/2024/06/17/how-tablets/

[31] Anita Śledź, Kajetan Husiatyński, Jan Ciołek and Michał Sala, *ScyllaDB Rust Driver*, Bachelor's thesis, Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, 2021. https://www.scylladb.com/wp-content/uploads/rust-driver-thesis.pdf

[32] *StackOverflow 2024 Developer survey — most admired.* https://survey.stackoverflow.co/2024/technology#admired-and-desired

[33] *StackOverflow 2024 Developer survey — most popular.* https://survey.stackoverflow.co/2024/technology#most-popular-technologies

[34] *TypeScript language.* https://www.typescriptlang.org/