University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Adam Boguszewski
Student no. 417730

Maciej Herdon
Student no. 418267

Marcin Mazurek
Student no. 359831

Gor Stepanyan
Student no. 404865

Library for Scylla Change Data Capture in Rust

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer–Daszkiewicz
Institute of Informatics

Warsaw, June 2022

Abstract

Change Data Capture (CDC) is a feature of the Scylla database that tracks all changes performed in a table. The logs of those changes are stored in a separate table and can be queried just like normal data.

The purpose of this thesis is to describe the process of creation of an open–source library for managing Scylla CDC feature in a relatively new programming language – Rust. Our main task was to implement a standalone library for interacting with CDC and an application for data replication that uses this library, but during the process we also had to modify the Scylla Rust Driver, which was developed in 2020 by students from the University of Warsaw.

Keywords

Scylla, database, Rust, asynchronous programming, Change Data Capture

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

Biblioteka dla Scylla Change Data Capture w języku Rust

Contents

Chapter 1

Introduction

Scylla is a distributed, real–time big data database that is API–compatible with Apache Cassandra and Amazon DynamoDB. Scylla embraces a shared–nothing approach that increases throughput and storage capacity to realize order–of–magnitude performance improvements and reduce hardware costs [9].

Starting from version 4.3, Scylla introduces the Change Data Capture [3] feature that allows to query the history of all changes made to a table in a database. The history is stored in a separate table and can be read like normal data. Some example use cases mentioned by the authors are:

- Heterogenous database replication,
- Implementing a notification system,
- In–flight analytics.

Although the CDC logs stored in a database table can be read with a Scylla driver or even a command–line application like *cqlsh*, the log format is too complicated to be used comfortably in this way. A hypothetical user would have to understand well the design of Scylla CDC. The main purpose of a library for CDC is to allow users to concentrate on business logic in their projects and not low–level intricacies of Scylla. Such libraries already exist for Go and Java, but thanks to the recently developed Scylla Rust Driver it is possible to create one for Rust, which will be a great addition for applications written purely in this language.

Our main task was to implement a convenient interface for interacting with CDC as a standalone library. The project uses asynchronous programming, with Rust–native async/await construction and Tokio runtime [16]. Because the Scylla Rust Driver is a new product, during the process we also had to tweak its codebase to suit our needs, for example to fetch metadata from the cluster or to add support for CDC partitioner, a hash function that computes which data is stored on which node in the cluster. The project also includes some example uses for this feature, the most important being the data replicator that copies data from one cluster to another. Naturally, the tests, the documentation and the benchmarks are also part of this project.

The thesis consists of the following chapters:

- **Chapter 2** contains a general overview of the problem, mentioning difficulties in regard to both Scylla and Rust.
- **Chapter 3** presents all elements of the solution in detail.

- **Chapter 4** describes the tests, the benchmarks and their results.
- **Chapter 5** is a summary of the whole thesis.

The project belongs to the free and open–source software in its entirety, so every stage of the project is documented in detail and this thesis is just a summary of it.

Chapter 2

Problem overview

2.1. Motivation

2.1.1. Environment matters

To build a successful product that will appeal to the hearts (and wallets) of software engineers all around the world it is not enough to strive for excellence in the product itself (although very much required) but one has to build and entire environment of tools around the product that will allow developers to comfortably interact with the product in a programmatic way. Well written tools encourage engineers to make use of the product in their projects while poorly thought out ones and even more so lack of them clearly discourage. This is very well understood at ScyllaDB where efforts are being made to provide libraries for interactions with Scylla in most popular programming languages. There has been some progress in achieving this in languages such as Go [6, 12], Java [7, 11], Python [8] but lately (since 2020) ScyllaDB's attention has shifted to Rust.

2.1.2. Rust's growth to power

Rust as of 15th March 2022 is still considered a young language but its popularity is massively growing. According to the StackOverflow's yearly Developer Survey, Rust is number one among the most loved programming languages since 2016 [15]. In 2020 it has entered into the top 20 of the most popular languages according to the same survey [14]. Such an accomplishment could not go unnoticed and a proper tooling for Rust has had to be provided. In 2020 ScyllaDB started to work on a client–side driver library for Scylla written in Rust [10]. This work, although far from finished, has laid a solid enough foundation for development of other required tooling. Among such tools is a library for interaction with Scylla Change Data Capture (from this point on referred to as Scylla CDC or simply CDC).

2.1.3. CDC Overview

Scylla CDC is an optional feature that can be enabled for each Scylla table individually. CDC creates a separate table, called Log Table, which stores additional information about every operation made on the original table, called the Base Table. It contains information such as what kind of operation was performed, whether a value was deleted as a result of the operation, timestamp of the operation and much more. As a result, it enables the users to track every single change made to a database which, for example, can be crucial in data analysis.

2.1.4. Task description

Our task was to provide an open–source library for high level interaction with Scylla CDC in Rust. Such a library has had its counterparts in Go [6] and Java [11]. This required to write on top of existing Scylla Rust Driver but also to expand the driver as it lacked CDC partitioner implementation (explained in subsection 2.2.3) and it did not provide sufficient metadata making database's schema introspection impossible. Moreover we were supposed to create two applications using the aforementioned library — one of them being printer app that writes to the standard output all the rows from CDC Log and the other one being replicator app to replicate changes from one Scylla table to another one using CDC Log.

2.2. Technical challenges

Without doubt, one of the most characteristic features of the Scylla database is the scalability, which implies some non–trivial solutions in the architecture. Due to that, the structure of the CDC log is also affected.

2.2.1. Scylla architecture

A cluster is a collection of nodes (Scylla instances). This structure is visualized as a ring. A collection of tables, analogous to databases in SQL, is called a keyspace. Data in Scylla can be stored on many nodes and the number of them is called the Replication Factor. This value is set by the user for every keyspace during its creation.

In order to enable the process of data distribution while keeping the possibility to read the data quickly, the primary keys in Scylla consist of two other keys: the partition key and the clustering key. The former is responsible for distributing the data between the nodes by dividing it into partitions (a subset of data with the same partition key). The latter is optional, and its role is to indicate how to sort the data within given partition.

As mentioned, partition key is used to identify the node in the cluster the partition belongs to. This is done using the partitioner. It calculates a token for each partition key. The set of all possible tokens is divided in continuous ranges. Such a range is called a virtual node or simply a VNode. Each physical node has some virtual nodes assigned to it (a single VNode may be assigned to multiple physical nodes to achieve desired data replication) and that assignment may change in reaction to cluster topology changes such as node addition or removal.

2.2.2. Implications for the log table

Because the log table is a table like any other, it also must follow these rules. The clustering key for this table comprises the time column and the column with a number of operations in a batch. The partition key is a binary large object (blob) value called a stream. Streams are the main reason, why reading the log data is not trivial. They are closely related to the data distribution between the nodes and change when a new node joins the cluster. A set of stream ids that are valid in the same time is called a stream generation. Because the structure of the log table does not order *every* operation by time, it is essential to pay attention to the moment when the generations change if we want to process the log in chronological order.

2.2.3. CDC partitioner

The ring nodes corresponding to the given partition key are calculated using a hashing function called partitioner — by default Murmur3 [1], however in case of log tables the partitioner is the primary key itself. Every clever driver should be aware of this in order to send its queries directly to the proper nodes saving time and network exhaustion. However Scylla Rust Driver had only the default partitioner implementation and treated log tables as every other table, so CDC partitioner had to be additionally implemented.

2.2.4. Asynchronuous programming

Aforementioned structure of the log table makes it convienient to read it concurrently. Because this requires many in–out operations with the database, asynchronuous programming is suited better to this task than the traditional, synchronuous approach. Rust supports this through built–in async/await construction. In this project we are using Tokio — asynchronous runtime built on top of Rust language.

Tokio by default is using work–stealing based thread pool along with a cooperative task yielding to reduce latencies tied to the I/O bound nature of our library. The runtime provides a high level of abstraction and common tools like tasks or channels, so it has low entry threshold.

2.3. Summary

This chapter introduced basic concepts of Scylla Change Data Capture and reasons for developing a library in Rust that simplifies reading the CDC log — the simplification of the process and potential demand for such a library in a flourishing programming language.

The next chapter will describe in detail the solution delivered by our team.

Chapter 3

Solution

The result of our work is an open source library available at `https://github.com/piodul/scylla-cdc-rust`, facilitating development of applications that read and process data from a CDC log of the Scylla database and a pair of applications (scylla-cdc-printer and scylla-cdc-replicator) demonstrating it, available in the same repository.

3.1. Library for CDC

3.1.1. Architecture overview

Our work consists of a couple of loosely connected modules. Those are:

- *cdc_types.rs*,
- *consumer.rs*,
- *stream_reader.rs*,
- *stream_generations.rs*,
- *checkpoints.rs*,
- *log_reader.rs.*

First one introduces utility types such as `GenerationTimestamp` and `StreamID`. The other ones are more complex so they require our special attention. In addition to this, our efforts on improving Scylla Rust Driver will also be discussed.

3.1.2. Module *consumer.rs*

From the user's perspective, probably the most important aspect of a library is its interface. In our case, this especially means the way the user can react to upcoming data from the database.

Out of all options for using user's code that Rust offers, the most convienient one are the traits. Traits can be considered as something similar to interfaces in object-oriented languages, but they also share some common concepts with type classes from functional languages like Haskell. To be concise, to implement a trait for a struct in Rust, one has to implement all functions related to that trait, which is almost identical to implementing an interface in an object-oriented programming language.

Therefore, in order to *consume* data using our library, the user has to create a struct implementing the `Consumer` trait defined by us:

```
#[async_trait]
pub trait Consumer {
    async fn consume_cdc(&mut self, data: CDCRow<'_>) -> anyhow::Result<()>;
}
```

This trait requires only one function to be implemented, which should be responsible for processing the data received from the database.

The data passed to this function is represented by a `CDCRow` struct. It represents a single row in the observed table in the database. The user can get all the column values from the log table just using a name of a corresponding column in the base table. An important fact to notice is that due to `CDCRow`'s internal structure and how the memory management works in Rust, the `CDCRow` instances cannot be used after the function has finished, and if the user wants to save the data for later, they should map it to some other data structure.

Since the library works concurrently, many instances of `Consumer` need to exist in order to consume data simultaneously. Because of that, the user is also required to create a struct implementing another trait — a `ConsumerFactory`.

```
#[async_trait]
pub trait ConsumerFactory {
    async fn new_consumer(&self) -> Box<dyn Consumer>;
}
```

This allows our library to create new `Consumer` instances when necessary.

3.1.3. Module *stream_reader.rs*

This module provides `StreamReader` component. It is responsible for querying the CDC logs and performing custom user defined operations on results. Based on user defined configuration it periodically fetches data from CDC log. The configuration includes:

- set of stream ids to track changes from,
- start timestamp from which the reader component should read the streams,
- time interval parameters:
  - single query time window size,
  - safety interval, so that the reader will not try to read data with later timestamp than the current time,
  - sleep interval, to sleep after processing results returned from all streams.

The public interface of this structure is presented below:

```
pub fn new(
    session: &Arc<Session>,
    stream_ids: Vec<StreamID>,
    start_timestamp: chrono::Duration,
    window_size: chrono::Duration,
    safety_interval: chrono::Duration,
    sleep_interval: time::Duration,
)

pub async fn fetch_cdc(
    &self,
    keyspace: String,
    table_name: String,
    mut consumer: Box<dyn Consumer>,
)

pub async fn set_upper_timestamp(&self, new_upper_timestamp: chrono::Duration)
```

Calling `fetch_cdc` starts querying CDC logs of a table given by `keyspace` and `table_name` parameters. The parameter `consumer` is a `Consumer` instance created with a `ConsumerFactory` provided by the user. The fetching will continue indefinitely unless some upper timestamp will be provided. It can be done using `set_upper_timestamp` method. To achieve this it is important to make these methods `async` and to properly manage shared `upper_timestamp`. This value is internally guarded by `tokio::sync::Mutex` and these methods have to interact with it properly. The `StreamReader` process follows the hereby algorithm:

- Keeps track of start timestamp `T`, which is either the user provided `start_timestamp` or the end timestamp of the last queried window.
- Calculates the end timestamp of the next query window with the following algorithm:

$$\texttt{U} = \max(\texttt{T}, \min(\texttt{T} + \texttt{window\_size}, \text{now} - \texttt{safety\_interval}))$$

- Queries CDC log rows from the time interval $[\texttt{T}, \texttt{U})$ from all streams at once in a single query.
- Processes fetched rows by passing them to the `Consumer`.
- Ends the process if upper timestamp is set and the end timestamp of the query window `U` exceeds the upper timestamp.
- Sleeps for the configured amount of time `sleep_interval`.
- Repeats from start.

3.1.4. Module *stream_ generations.rs*

One of the most important reasons for creating a library for Scylla CDC is simplification of the process of reading data in the correct order, which requires understanding how stream generations work. As mentioned before, a stream generation is set of stream ids valid in the same time. Scylla instances provide special tables that contain timestamps indicating their start time and also all the stream ids.

In order to meet that objective, we created a component providing a convenient API with standard operations for CDC generations:

- Get all generations.

  ```
  pub async fn fetch_all_generations(&self) ->
      anyhow::Result<Vec<GenerationTimestamp>> {
      (...)
  }
  ```

- Get a generation by a timestamp.

```
pub async fn fetch_generation_by_timestamp(
    &self,
    time: &chrono::Duration,
) -> anyhow::Result<Option<GenerationTimestamp>> {
    (...)
}
```

- Given a generation, get the next one.

```
pub async fn fetch_next_generation(
    &self,
    generation: &GenerationTimestamp,
) -> anyhow::Result<Option<GenerationTimestamp>> {
    (...)
}
```

- Given a generation, get all stream ids that belong to it. The returned stream ids are grouped together by VNodes.

```
pub async fn fetch_stream_ids(
    &self,
    generation: &GenerationTimestamp,
) -> anyhow::Result<Vec<Vec<StreamID>>> {
    (...)
}
```

Through this API, other components of our library can easily fetch valid stream ids without worrying about how and when do the generations change.

Every function in the public API was tested using unit tests. For testing purposes we have hardcoded test tables that have the same schema as the tables containing information about CDC generations and streams. There was a small problem during testing since tables containing information about generations and streams use replication factor equal to 3, which means the data is replicated to 3 different nodes. Querying such table by default needs to achieve quorum to perform, in our case it means that at least two nodes are needed for the query to work, so it didn't work for databases with only a single node. We solved this problem by creating `new_distributed_system_query()` function that given a query checks number of nodes and modifies given query to make it work.

3.1.5. Module *checkpoints.rs*

In this module, we have created a component responsible for periodically saving progress during processing CDC rows by our library.

Firstly, we introduce a new type — `Checkpoint` that consists of the last processed timestamp, an ID of the stream that made this checkpoint and a generation of this stream. Whenever we use the term 'checkpoint', we mean an object of this type.

Secondly, we introduce a new trait `CDCCheckpointSaver` that will be a contract for an object to implements these methods:

```
async fn save_checkpoint(&self, checkpoint: &Checkpoint)
    -> anyhow::Result<()>
async fn save_new_generation(&self, generation: &GenerationTimestamp)
    -> anyhow::Result<()>
async fn load_last_generation(&self)
    -> anyhow::Result<Option<GenerationTimestamp>>
async fn load_last_checkpoint(&self, stream_id: &StreamID)
    -> anyhow::Result<Option<chrono::Duration>>
```

To save progress or load previously saved one, the user has to pass an instance of the object implementing `CDCCheckpointSaver` to the `CDCLogReaderBuilder` and set the correct options in the builder, i.e., whether the readers should save or load progress. `CDCLogReaderBuilder` will be briefly described in the next subsection.

We also provided an implementation of the `CDCCheckpointSaver` that saves checkpoints in a Scylla's table. This default implementation stores the latest checkpoint for each stream. It also stores the information about the latest generation in a special row with stream ID equal to 0. The user can set TTL for the table with checkpoints. If he doesn't, the default value of 7 days will be used, i.e., after 7 days Scylla will delete a record.

3.1.6. Module log_reader.rs

The modules described above are used to do various, separate tasks. The module that combines them together and provides an interface for the user to actually use the library is the `log_reader` module.

The `CDCLogReader` component requires many arguments from the user, so we decided to use the Builder pattern as a way to create a new `CDCLogReader`. As for the necessary steps, the user must provide:

- a `ConsumerFactory`,
- a `Session` (a component from the Scylla Rust Driver) connected with the database,
- keyspace and table name.

Additionaly, they can also configure some other parameters:

- start and end timestamps,
- time interval parameters,
- configuration options regarding saving and loading progress.

After passing all important arguments to the builder, the user should use the `build()` method. It will start processing the CDC log.

The result of `build()` method of `CDCLogReaderBuilder` is a tuple with two elements — the `CDCLogReader` component and a `RemoteHandle`. The `CDCLogReader` allows user to stop reading the CDC log at any moment. The handle is used to make sure that the reader has stopped working — one can `await` the handle to wait until the reader finishes.

The algorithm used in `CDCLogReader` coordinates usage of all the other modules. It works in an event loop, waiting for one of the following events:

- change of the stream generation,
- change of the end timestamp,
- end of work of every `StreamReader`.

In the first iteration, the reader fetches stream ids with the `fetch_stream_ids` method and creates a `StreamReader` for every VNode. The `StreamReaders` work in separate Tokio tasks. The reader also configures the checkpoint saver, if checkpoint saving was enabled.

During the next iterations, the reader waits for one of the aforementioned events. If there is a new generation, the `StreamReaders` are told to read data only until the new generations starts. If the user changes the end timestamps, all `StreamReaders` are notified about this fact. Finally, if every `StreamReader` finished its job, the reader checks if the reading should be finished. If yes, it ends its execution. If no, the same steps as during the first iteration are repeated.

3.1.7. Scylla Rust Driver improvements

Schema metadata

The structure `SessionConfig` has gained a new field — `fetch_schema_metadata` and `SessionBuilder` corresponding method. This parameter (`false` by default for the sake of efficiency) determines if schema metadata fetch will be performed periodically. If so the driver will query for all keyspaces, tables, columns, user defined types and partitioners and save the data. It can be accessed by `get_cluster_data` method in `Session`. This provides the possibility of schema introspection by Scylla Rust Driver clients.

Result metadata

Similarly metadata for query results had to be provided. To this end structures `RowIterator` and `TypedRowIterator` have gained `pub fn get_column_specs(&self) -> &[ColumnSpec]` methods.

CDC Partitioner

Support for CDC-specific partitioning scheme in Scylla Rust Driver was provided by refactoring token calculation logic, making it partitioner aware and finally implementing the CDC partitioner.

3.2. Scylla-cdc-printer

This is a very basic toy program made to demonstrate the ability of the library to interact with CDC log. It simply tracks the given table and prints the CDC log to the standard output.

3.3. Scylla-cdc-replicator

This one is yet another program built on top of our library but this time more complex. Scylla-cdc-replicator is a program that subscribes to the given table and replicates changes made in it in another given table using CDC log. Support was provided for all CDC operations, such as:

- row insert,
- row update,
- row delete,
- partition delete,
- range delete.

It works with all types supported by Scylla including User Defined Types.

3.4. Summary

This chapter presented concrete elements of the created library, such as the user interface and the components responsible for fetching the data from the database. It has also mentioned necessary changes in the Scylla Rust Driver and example applications that use the library.

The next chapter will focus on the methods used to verify the correctness and usefulness of the library.

Chapter 4

Validation

In order to verify that the library works correctly, we have run benchmarks and various types of tests.

4.1. Tests

The library was tested with three different types of tests. All of them are acceptance tests and have passed.

4.1.1. Unit tests

First of all, there are standard unit tests for every module, which check the basic functionality of the library. These tests were written natively in Rust and can be run easily by using the Rust-native `cargo test` tool. Most of them connect to a Scylla cluster to verify that the functionalities work with real data.

A special case of the unit tests are the tests created for the replicator. These tests check if every type of operation with every type of data is replicated correctly, without using the library. To achieve that, we created a small framework that facilitates adding the data to the database, replicating and comparing the results — it is located in the file `scylla-cdc-replicator/src/replication_tests.rs` in the repository.

4.1.2. End-to-end tests

The purpose of these tests was to check that one of the most important assets of the library works — namely, that changes referring to the same partition key are read in a chronological order and no row is missed during the process. To verify that, the tests perform some operations on a table and remember their order for each partition key. After that, the library is used to track all the changes and also remember their order for each partition key. Finally, both sets of operations are being compared to check if the chronological order was preserved by the library. These tests also have been written natively in Rust and are located in the file `scylla-cdc/src/e2e_tests.rs`.

4.1.3. Replicator integration tests

The last type of tests we used are tests of one of our example applications — the replicator. These tests verify that the replicator replicates the data correctly.

These tests were provided by ScyllaDB and were originally created for the Java version of the replicator. Due to that, we had to make some changes so that they work with our version of the replicator. The source code of the tests is located in a repository of a Scylla employee [4].

4.2. Benchmarks

Performance is one of the most commonly mentioned advantages of Rust. Because of that, there was a natural urge to test how fast the library works.

4.2.1. Method

At the beginning, we decided to compare the performance of Scylla-cdc-java [11] (on Java 8) with our library. At first, we did not compare with Go version of the library, because it contained some bugs that showed up when we tested it on such big data. The benchmark applications for both libraries work in the same way. They start reading from the first CDC generation in the cluster and then they count the number of read rows, and when that number reaches a certain value the application stops. To ensure that no row has been read twice, the applications calculate control sums — they sum all the values of clustering keys. In both cases, the benchmark applications were based on the printer.

The only varying parameter in these benchmarks was the length of the query window size — we tested values of 15, 30 and 60 seconds. The safety interval played no role here, because all the data was written before running the benchmark applications. The sleep interval was set to one millisecond in the Rust application in order to mimic Scylla-cdc-java's behavior, which does not sleep between processing old data.

The input data was written with Scylla-bench [5]. The table used had the default Scylla-bench schema — partition key and clustering key are of type *bigint*, while the only value column is of type *blob*. The table contained 40000 different partition keys with 2000 clustering keys for every partition key — 80000000 rows in total. All the corresponding CDC rows were of type *RowInsert*.

The results were timed with the GNU time command [2] with flag `-v`. The value "Percent of CPU got" refers to sum of values for all cores. For example, value of 200% means that the application has used amount of CPU time comparable to the amount of time it would use on 2 cores with 100% percent of CPU got for both of them.

The Scylla cluster for benchmark purposes had three nodes, each of them running 14 shards.

The machine on which we launched the benchmarks had 36 CPU cores and 68 gigabytes of memory.

4.2.2. The first try

The first try of the benchmarks resulted in a small failure. Scylla-cdc-rust did run faster as shown in Table 4.1, however the elapsed time was not even two times shorter than for Scylla-cdc-java.

Parameter	Rust			Java		
Window query size (s)	15	30	60	15	30	60
User time (s)	118.55	105.71	96.64	571.60	584.82	593.50
System time (s)	20.51	15.87	12.85	266.08	247.48	241.30
Percent of CPU got	50%	58%	60%	245%	245%	247%
Elapsed (wall clock) time (m:ss)	4:35.90	3:28.36	3:01.78	5:41.66	5:39.05	5:36.72

Table 4.1: Results of the first benchmark run

As the metrics have shown, all the requests from the Rust driver were received by the same single shard, which resulted in big latencies between the requests. When in a CQL query one or more partition keys did not have corresponding bind marker, e.g. `pk IN ?` or `pk = function(?)`, the driver used to get empty information to obtain a token to route the request to the appropriate shard. As the mentioned queries are valid, the driver will always calculate a dummy token and all the requests will be routed to the same node/shard. The problem was quickly discovered and fixed in a way that requests with empty information about partition keys are detected in the driver and sent to random shards.

Nevertheless, the benchmark still proved some points. First of all, both user and system time were significantly shorter in the case of Scylla-cdc-rust. Moreover, the percentage usage of a one core was roughly five times smaller — for Rust it varied between 50% and 60%, while for Java it varied between 245% and 247%. Both values are relatively low when compared to the total possible CPU usage, which is $36 \cdot 100\%$, however it is not a surprise, because both libraries have to communicate with the database and wait for the results. They also do not perform any operations that would require a higher CPU usage.

An interesting fact is that different query window sizes did affect the results of Scylla-cdc-rust positively, in Scylla-cdc-java the differences are barely noticeable. This was probably related to the bug — with a greater window size, the Rust application sent less requests to the database.

4.2.3. The second try

Thankfully the problem in the first try was quickly fixed and we were able to rerun the benchmark. We also managed to solve problems with Scylla-cdc-go — both in the library and in the driver. One of the bugs was the same bug that was present in the Rust driver on our first try. The results were better this time as shown in Table 4.2. Figure 4.2 focuses on elapsed time and shows clearly that our library outperforms competing ones.

Parameter	Rust			Java			Go		
Window query size (s)	15	30	60	15	30	60	15	30	60
User time (s)	91.96	94.06	90.95	618.27	594.69	570.11	272.72	236.36	217.81
System time (s)	15.46	13.02	13.88	304.02	273.86	254.97	20.45	16.66	15.90
Percent of CPU got	169%	147%	115%	261%	250%	243%	404%	310%	234%
Elapsed (wall clock) time (m:ss)	1:03.33	1:12.56	1:30.88	5:53.31	5:46.62	5:38.60	1:12.47	1:21.51	1:39.70

Table 4.2: Results of the second benchmark run

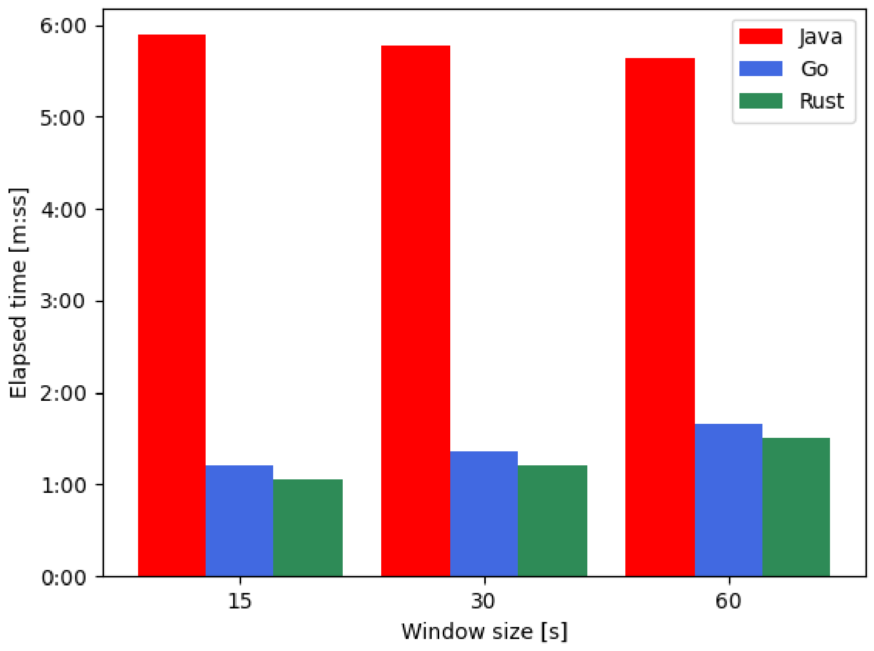

Figure 4.1: Comparison of CDC libraries performances with respect to different windows sizes

Besides a speed up of the Rust library, we can also see that it used a greater percent of a single CPU than last time. This is probably because now the latencies were much smaller and the benchmark application was not idle that often. Results of Scylla-cdc-java are roughly the same as before. Scylla-cdc-go did run as fast as Scylla-cdc-rust, but its user time was definitely larger, as well as percentage of used CPU — the benchmark machine had 36 cores, so it still was not a bottleneck, however the application probably would slow down on a machine with less cores.

What is interesting is that performance of Scylla-cdc-rust and Scylla-cdc-go have negative correlation with bigger window size whereas for Scylla-cdc-java it works in a positive way. This problem might be related to the database — it had to send more rows as a result of every request.

ScyllaDB was very surprised with weak performance of the Java library, so we decided to run some more benchmarks — we tried different Java versions (11 and 17) and running the application on different number of cores to verify that the benchmark was fair. We did not find any bugs in the benchmark application. Below in Table 4.3 are the results of comparison of Rust and Java 11 on different number of cores.

Parameter	Rust				Java			
Number of cores	1	2	4	8	1	2	4	8
User time (s)	68.23	79.97	85.52	90.44	361.96	389.91	532.75	713.12
System time (s)	4.37	8.98	11.68	13.41	21.73	50.79	188.92	311.27
Percent of CPU	97%	140%	150%	159%	95%	137%	210%	255%
Elapsed time (m:ss)	1:14.75	1:03.32	1:04.51	1:05.14	6:42.09	5:19.82	5:43.56	6:40.92

Table 4.3: Results of a benchmark for Rust and Java 11 on different number of cores

Figure 4.2 contains the dependency of elapsed time on number of cores visualized.

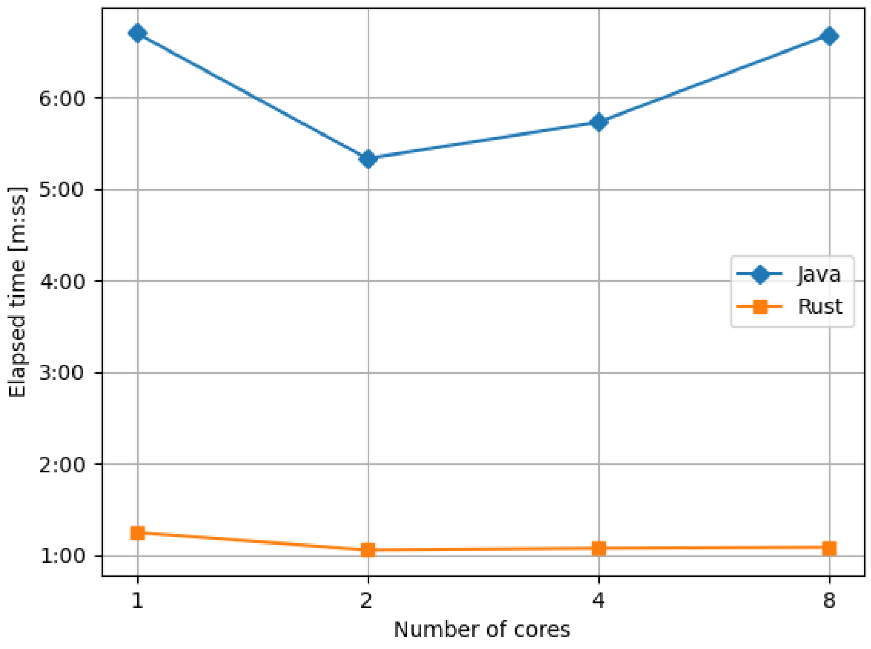

Figure 4.2: Comparison of CDC libraries performances with respect to the number of cores

Unfortunately, these results are not very useful to judge the performance of Scylla-cdc-rust. This is probably because of how it creates new `StreamReaders`. The cluster we used to run this benchmark on contained only 3 nodes with 42 shards in total. Due to that, the library's speed did not scale that well with the number of cores. To actually test that, we should have run the benchmark on a cluster with a lot of nodes, which was beyond our possibilities at the moment and would require a lot of computing power.

An interesting fact is that Scylla-cdc-java has run faster on two cores, only to slow down on more cores.

Because of the problems with Scylla-cdc-go, we did not have enough time to run that many benchmarks for it. Judging on its result in the basic benchmark, it probably would be slower on one and two cores, but would scale better.

4.2.4. Conclusions

In the benchmark Scylla-cdc-rust performed much better than Scylla-cdc-java and slightly better than Scylla-cdc-go. However, trying to overpass the results of Rust and Go might be difficult. The problem lies in the database — it simply cannot process requests faster. During the benchmark, we saw that the load reached high numbers (over 80%) and the latencies were a bit larger than usual.

4.3. Summary

This chapter concentrated on our attempts to verify the correctness and performance of the library. The correctness has been successfully confirmed by different types of tests. The performance was tested only partially — we did not have enough time to invent and run more benchmarks. However, the result of the benchmark seems satisfactory — it shows that Scylla-cdc-rust is comparable to Scylla-cdc-go and has a potential to be much faster than Scylla-cdc-java.

The next and final chapter will be a summary of the whole thesis.

Chapter 5

Summary

5.1. Conclusions

5.1.1. Project results

Our main goal was to create a library that facilitates reading the CDC log in Scylla database, using asynchronous programming in Rust. This task has been completed — we have managed to create an easy to use library that offers more than both, already existing, libraries in Go and Java. Our version of the library allows users to read data between any two timestamps, whereas Java and Go versions have some limits on time bounds. We have also implemented an option to save the progress while reading the data, which, up to now, was only a feature in the Go version of the library.

The performance of the library is also satisfactory. A simple benchmark showed that it might be much faster than Java version of the library, even when tested with a bug that increased the latencies in the database requests drastically.

One of our side goals was to make modifications in the Scylla Rust Driver. This goal has also been achieved. We have added support for features regarding the metadata. The driver also can use the CDC Partitioner now.

Finally, we have managed to create two example applications, the Printer and the Replicator, that show how to use the library. Additionaly, the Replicator has been tested with tests provided by ScyllaDB, which is an additional confirmation of the correctness of the library.

5.1.2. Future of the library

The library is going to be published under the name *scylla-cdc* on `https://crates.io/`, which is a service that hosts Rust dependencies and allows programmers to convieniently download them with `cargo` tool.

One optional step that we did not manage to finish in time was to enable the library to consume data in a distributed way. However, the library was designed to support this feature and it can be added later.

5.2. Work division

In the project there were some tasks that have been divided into smaller subtasks and due to that, every member of our team took part in them — one such example is the Replicator. Among other tasks that could not have been divided, the work was divided as follows:

- Adam Boguszewski:
  - Module *consumer.rs*
  - Framework for unit tests for the replicator
  - End-to-end tests
- Maciej Herdon:
  - Module *stream_ generations.rs*
  - Module *checkpoints.rs*
- Marcin Mazurek:
  - Logic of the Log reader
  - Improvements in the Scylla Rust Driver
- Gor Stepanyan:
  - Module *stream_ reader.rs*
  - Log reader's builder interface

Bibliography

[1] Austin Appleby, *MurmuHash3*, 2016. Available at: `https://github.com/aappleby/smhasher/wiki/MurmurHash3` [Accessed 20 April 2022].

[2] Free Software Foundation, Inc., *GNU Time*, 2018. Available at: `https://www.gnu.org/software/time/` [Accessed 2 June 2022].

[3] ScyllaDB, *CDC Overview | Scylla Docs.*, 2022. Available at: `https://docs.scylladb.com/using-scylla/cdc/cdc-intro/` [Accessed 30 March 2022].

[4] ScyllaDB, *GitHub — kbr-/scylla-test*, 2020. GitHub. Available at: `https://github.com/kbr-/scylla-test` [Accessed 2 June 2022].

[5] ScyllaDB, *GitHub — scylladb/scylla-bench*, 2020. GitHub. Available at: `https://github.com/scylladb/scylla-bench` [Accessed 2 June 2022].

[6] ScyllaDB, *GitHub — scylladb/scylla-cdc-go*, 2021. Available at: `https://github.com/scylladb/scylla-cdc-go` [Accessed 16 March 2022].

[7] ScyllaDB, *GitHub — scylladb/java-driver: ScyllaDB Java Driver for ScyllaDB and Apache Cassandra, based on the DataStax Java Driver*, 2022. Available at: `https://github.com/scylladb/java-driver` [Accessed 16 March 2022].

[8] ScyllaDB, *GitHub — scylladb/python-driver: ScyllaDB Python Driver, originally DataStax Python Driver for Apache Cassandra*, 2022. Available at: `https://github.com/scylladb/python-driver` [Accessed 16 March 2022].

[9] ScyllaDB, *GitHub — scylladb/scylla: NoSQL data store using the seastar framework, compatible with Apache Cassandra*, 2022. Available at: `https://github.com/scylladb/scylla` [Accessed 16 March 2022].

[10] ScyllaDB, *GitHub — scylladb/scylla-rust-driver: Async CQL driver for Rust, optimized for Scylla!*, 2022. Available at: `https://github.com/scylladb/scylla-rust-driver` [Accessed 16 March 2022].

[11] ScyllaDB, *GitHub — scylladb/scylla-cdc-java*, 2021 GitHub. Available at: `https://github.com/scylladb/scylla-cdc-java` [Accessed 16 March 2022].

[12] ScyllaDB, *GitHub — scylladb/gocqlx: All–In–One: CQL query builder, ORM and migration tool*, 2022 GitHub. Available at: `https://github.com/scylladb/gocqlx` [Accessed 16 March 2022].

[13] ScyllaDB, *Welcome to ScyllaDB Documentation | Scylla Docs.*, 2022. Available at: `https://docs.scylladb.com` [Accessed 16 March 2022].

[14] Stack Overflow, *Stack Overflow Developer Survey 2020*, 2020. Available at: `https://insights.stackoverflow.com/survey/2020` [Accessed 16 March 2022].

[15] Stack Overflow, *Stack Overflow Developer Survey 2021*, 2021. Available at: `https://insights.stackoverflow.com/survey/2021` [Accessed 16 March 2022].

[16] Tokio, *Tokio — An asynchronous Rust runtime*, 2022 GitHub. Available at: `https://tokio.rs` [Accessed 30 March 2022].