University of Warsaw
Faculty of Mathematics, Informatics and Mechanics

Samvel Abrahamyan
Student no. 391211

Michał Chojnowski
Student no. 394134

Adam Czajkowski
Student no. 394148

Jacek Karwowski
Student no. 359713

Apache Parquet support for ScyllaDB

Bachelor's thesis
in COMPUTER SCIENCE

Supervisor:
dr Robert Dąbrowski
Institute of Informatics

Warsaw, May 2020

Abstract

ScyllaDB is a high performance NoSQL database for Big Data. It achieves its primary design goal, low latency, thanks to its purpose-built concurrency framework for C++ – Seastar. However, for correct operation, all I/O and concurrency management in Seastar-based projects has to be performed exclusively through the framework. This excludes many existing libraries from use in Scylla. In particular, this affects libraries for Apache Parquet, a popular columnar data storage format for Big Data, which has some potential uses with Scylla. For example, direct import/export tools would be desirable for users of both products. In this thesis, we present a new Parquet library for Seastar, parquet4seastar, and we compare its performance against the closest competitor – Apache Arrow. We also present a CLI tool, parquet2cql, for converting Parquet files to statements of CQL (Scylla's query language), show a proof-of-concept Scylla module for storing Scylla's backend data in Parquet, and compare Parquet's disk usage against Scylla's native format, SSTables.

Keywords

Big Data, Databases, Data formats, Parquet, ScyllaDB, C++

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

Information storage systems – Record storage systems – Relational storage – Column based storage
Information systems – Data management systems – Database design and models – Physical data models
Software and its engineering – Software organization and properties – Extra-functional properties – Interoperability

Tytuł pracy w języku polskim

Wsparcie dla Apache Parquet w ScyllaDB

Contents

Chapter 1

Introduction

The goal of our Team Programming Project was, in broad terms, to bring support for Apache Parquet – a popular columnar storage format for big data analytics – to Scylla, a high performance NoSQL database.

Scylla's developers took interest in Parquet for two reasons. First, some Scylla users import big volumes of data from Parquet files into the database, but the typical way to perform this task is unnecessarily complex and involves routing the data over the network. A direct import path is desirable for efficiency reasons. Second, since Parquet is expected to fit certain workloads better than Scylla's native format (SSTables), it is a candidate for internal use in Scylla as a secondary storage format.

However, existing implementations of Parquet could not be integrated with Scylla due to their fundamental incompatibility with Scylla's specialized concurrency framework. We were tasked with providing an adequate solution.

Ultimately, our project resulted in three products: a library for reading and writing `.parquet` files, a tool for converting `.parquet` to CQL (Scylla's `CREATE` and `INSERT` statements), and a Scylla module for outputting `.parquet` files as a secondary internal format.

Chapter 2 describes the technologies in question. First, it explains the motivation behind columnar data storage formats and the way they can be used for efficient storage of complex, non-tabular records. Then, some details about the structure and features of Parquet format follow.
The second part of the chapter focuses on ScyllaDB. It also introduces Seastar – Scylla's underlying asynchronous concurrency framework, whose existence prompted our project.

Chapter 3 presents the results of our work. A reader familiar with both Parquet and Scylla might want to skip to this part directly. The chapter begins with a summary of our solution and explores the details of its three constituent parts (the library, the converter to CQL, the database module) as it progresses.

Chapter 4 shows the path we took to get to the final result. Over the course of the project, we have tried two completely different approaches – first trying to reuse existing implementations, and then writing one of our own – and worked out the design in iterations.

Chapter 5 lists the development tools we have used in our project and talks about the difficulties we have encountered while working with them.

Chapter 6 discusses the effectiveness of our solution, includes the description of tests and metrics we used and talks about potential optimizations.

Chapter 7 fulfils the formal requirement of describing the division of responsibilities in our team.

In the end notes, we summarize the work done and point some further directions the project might go in the future.

Appendix A describes the contents of CD attached to the thesis.

Appendix B contains the listing of our SSTables-to-Parquet schema translation.

Acknowledgement: We would like to thank our supervisors – dr Robert Dąbrowski at the University of Warsaw, and Piotr Sarna at ScyllaDB – for their support during the development of this project.

Chapter 2

Technologies

2.1. Parquet

Parquet is a columnar data storage format for big data analytics focused on fast sequential reads and efficient use of disk space. It is free and open (under Apache Licence v2.0), developed actively under the auspices of Apache Software Foundation, and used by multiple popular data-processing frameworks, including Spark, Impala and Pandas. It provides multiple encoding and compression schemes and efficiently supports complex records (think: JSON-like data, with a known schema).

2.1.1. Columnar data formats

Parquet's defining feature is its columnar data layout.
We usually think about a dataset as a two-dimensional table (Figure 6.2), but that is an abstraction: memory is one-dimensional, so the concrete representation of the table on disk has to order the fields of the table in some way. There are two natural ways to do this: by column (Figure 2.2b), or by row (Figure 2.2a).

The row-based layout is the more common one, and has a number of advantages:

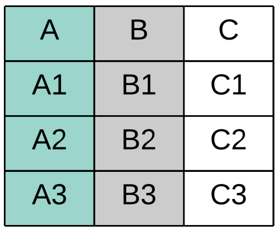
- adding new records is easy and does not require buffering
- access to a particular record is straightforward – the only needed information is the record's offset
- getting all data from a particular record is fast – it should require only one disk operation, since all needed data is in one place

However, in this approach:

- multiple different kinds and types of data are interwoven throughout the file, which decreases the effectiveness of compression algorithms
- some data encoding schemes (e.g. bit-packing of small integers) are impossible or impractical
- when multiple records are read sequentially, the reader always pays the cost of reading all record fields – including those he is not interested in. The performance penalty has two forms – first, disk throughput wasted on unwanted data, and second, reduced cache efficiency

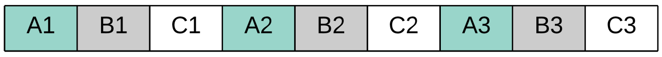
- the use of vectorized hardware instructions for processing such data is impossible

Then, what if we were to transpose this situation into a one where the data is stored column-by-column?

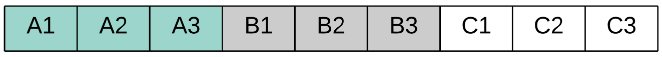


A	B	C
A1	B1	C1
A2	B2	C2
A3	B3	C3

Figure 2.1: A table that is to be represented in linear memory layout



(a) Placed in a row-by-row fashion



(b) Placed in a column-by-column fashion

Figure 2.2

Now records can't be simply appended – the columns either need to be buffered in memory or written to separate files. And more importantly: random access to rows on disk becomes prohibitively expensive, because each field requires a separate disk operation.

In exchange, however, we get to solve all the aforementioned problems with row-oriented storage.

If a workflow for a given table does not involve seeking for particular rows, but rather the sequential processing of all rows, read performance and disk space usage improve by using a columnar format, without any drawbacks. If one is also interested in reading only a subset of columns at a time, a columnar layout clearly becomes the superior choice. This often happens to be the case in data analytics.

As a side benefit, deleting and adding columns to the dataset becomes trivial and does not require modifying the data – just the metadata.

2.1.2. Complex records

It is easy to represent tables of simple atomic data in a columnar form. However, Parquet was built with the assumption that a modern data storage format, in order to be useful, needs to natively handle complex records, made of nested objects, lists, and optional values (think: JSON, but with a schema). Designing an efficient columnar representation of such records, which preserves all the necessary information when only a subset of columns is read, is a challenging problem. Parquet's authors decided to base their solution on the ideas introduced

```
message BookCatalogue {
  required string Title;
  repeated string Author(s);
  repeated group Translations {
    required string Country;
    optional string Publisher;
  }
}
```

Listing 2.1: Example schema of an book catalogue

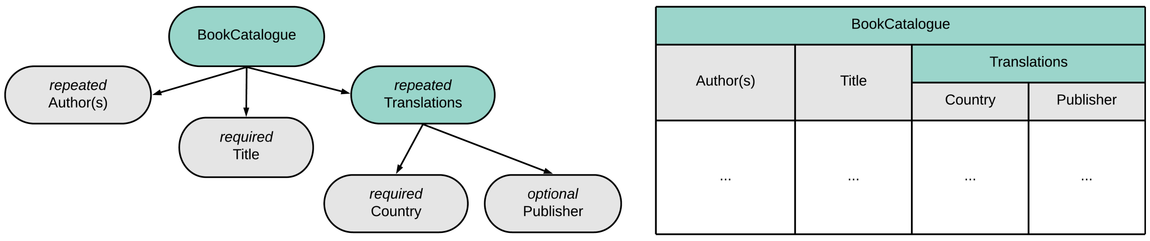


Figure 2.3: Tree-table schema correspondence

in Google Dremel [1].

The schemata we are working with are trees of strongly typed, named, nested fields. Each of them has an additional label describing multiplicity:

- **required** – there is exactly one value associated with this field
- **optional** – there is zero or one values associated with this field
- **repeated** – there are arbitrary many values (a list) associated with this field

A type of a field can either be:

- atomic (**int**, **boolean**, **double**, constant-length **string**, variable-length **string**, etc.)
- compound (**group**)

The top-level record type is denoted by a **message** keyword and does not have a multiplicity label (it's obviously **repeated**). An example schema is presented in Listing 2.1. If we view such a schema as a tree, each column of data corresponds to a leaf (Figure 2.3).

Definition levels

Consider a schema shown in Listing 2.2.

```
message Levels {
  optional group L1 {
    required group L2 {
      optional string L3;
    }
  }
}
```

Listing 2.2: Example schema with nested optional fields

As the corresponding tree has one leaf, the table will have one column: namely, `L1.L2.L3`. When we read a row with a `NULL` value, there is no way to tell which of these two structures it came from:

- `{L1: {L2: {L3: NULL } } }`
- `{L1: NULL }`

To capture this information, for every leaf that is optional or has an optional ancestor, we introduce a definition level (DL) – an additional column, describing where the field started to be a `NULL` – Table 6.3 shows what that would look like.

L1.L2.L3	DL	Corresponding structure	Explanation
"value"	2	`{L1: {L2: {L3: "value" } } }`	A maximal possible DL, so the field is defined.
NULL	1	`{L1: {L2: {L3: NULL } } }`	A DL one less than maximal, so the lowermost level is undefined
NULL	0	`{L1: NULL}`	A DL two less than maximal, so the undefinedness starts two levels up.

Table 2.1: Examples of structures with varying DL

Note that DL is incremented only for fields that are optional. This means that a required field with all ancestors being required would have a constant DL of 0, which is why we don't need DL column for such fields. In particular, there is no DL overhead for always-required schemes – and otherwise, the overhead is small – DL values are very small, requiring only a few bits to represent, and also tend to be very repetitive, lending themselves to run-length encoding.
Note: for the purpose of discussing DL, **repeated** fields are optional too. Empty list is no different from an empty optional.

```
message Levels {
  repeated group L1 {
    repeated string L2;
  }
}
```

Listing 2.3: Example schema with nested repeated fields

Repetition levels

Consider the schema shown in Listing 2.3 representing a list of lists of strings.
The corresponding table has only one column `L1.L2`, and looks like that:

L1.L2
"a"
"b"
"c"
"d"
"e"

Table 2.2: Repeated data without repetition levels

Such values could have came from any of:

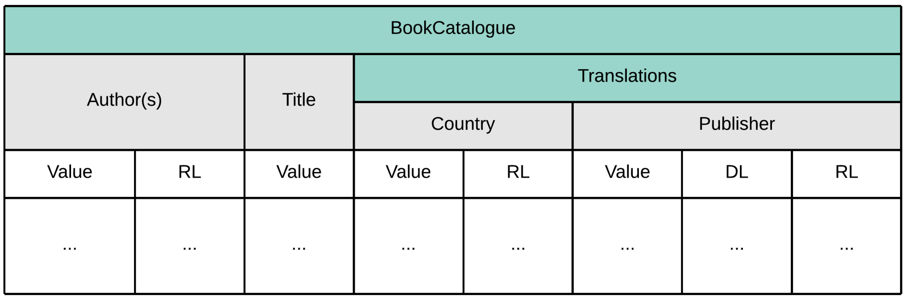
- `[[a], [b], [c], [d], [e]]`
- `[[a,b,c,d,e]]`
- `[[a,b], [c], [d,e]]`
- ...

In other words, we need more information to reconstruct the lists. We don't want to insert boundary markers into the data, though (this would ruin the uniformity and break coding schemes). Therefore, we introduce a new column again (for every field that is repeated or has a repeated ancestor), indicating how high the repetition 'starts over'. For example, the first case from the above corresponds to:

L1.L2	RL
"a"	0
"b"	1
"c"	0
"d"	0
"e"	1

Table 2.3: Repeated data with corresponding repetition levels

Together (as in Figure 2.4), the (value, definition level, repetition level) triplets fully capture the structure of complex records, and this information is preserved for every projection of the schema. Refer to the Dremel paper [1] for a detailed discussion of this encoding and the associated record shredding and assembly algorithms.



BookCatalogue							
Author(s)		Title	Translations				
			Country		Publisher		
Value	RL	Value	Value	RL	Value	DL	RL
...	...	...	...	...	...	...	...

Figure 2.4: The actual layout of the example schema, including additional definition level and repetition level columns.

2.1.3. Parquet file structure

A Parquet file has a hierarchical structure:

1. At the highest level, the data is divided into row groups. A row group is intended to be several hundred megabytes in size and be the unit of processing in a distributed computation framework. (Specifically, a row group is supposed to fill a single block in the distributed Apache Hadoop filesystem).

2. Each row group consists of a number of column chunks, each of them being continuous on disk and containing all data of a particular column in the row group. Column chunks are the units of sequential I/O. Their locations on disk are described in the file metadata, the reader is supposed to select the ones of interest and then read them sequentially.

3. Column chunks are in turn divided into pages, which are the units of CPU processing. Each page is a batch of data serialized, encoded and compressed as an atomic unit – to read a value inside a page, the reader has to decode the entire page, but doesn't have to process anything else. Pages should have a size of several kilobytes – small enough to be comfortably handled in memory after decompression, and big enough to have significant compression potential, and proportionally low space overhead of page metadata.

The file ends with a footer containing the file metadata (most importantly – the schema) and locations of individual row groups and column chunks. All metadata in Parquet is serialized to a binary form using the `TCompactProtocol` format borrowed from Apache Thrift.

A graphical representation of this structure is presented in Figure 2.5.

2.1.4. Encoding and compression

Parquet uses various encoding schemes, each providing an advantage under different conditions. An important thing to keep in mind is that the encoding is chosen on the level of page – this means that even in the case of globally non-uniform columns, on the local level there might be enough regularities to make these encodings work. Examples include [3]:

1. **Plain** – Trivial encoding. Data is simply concatenated.

2. **RLE (run-length encoding)** – If the same value is repeated many times in a row, all these repetitions are replaced by a a pair (value, count).

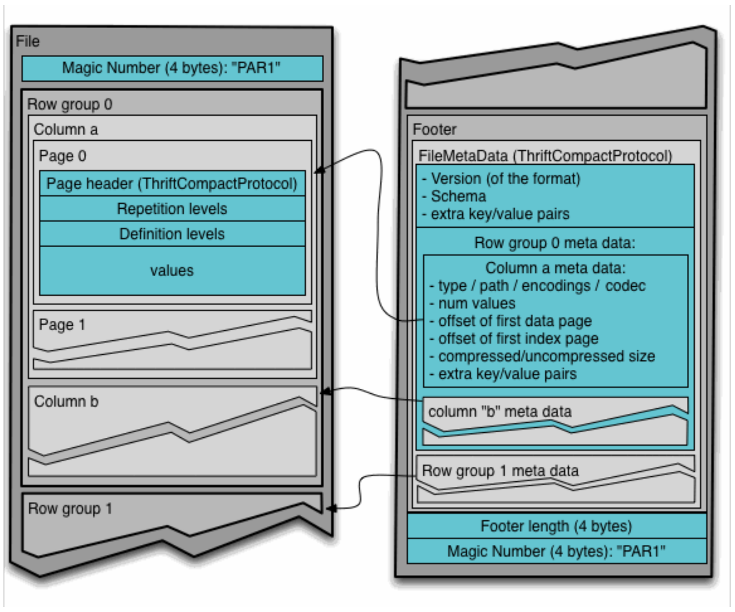

File
Magic Number (4 bytes): "PAR1"
Row group 0
Column a
Page 0
Page header (ThriftCompactProtocol)
Repetition levels
Definition levels
values
Page 1
Column b
Row group 1
Footer
FileMetaData (ThriftCompactProtocol)
- Version (of the format)
- Schema
- extra key/value pairs
Row group 0 meta data:
Column a meta data:
- type / path / encodings / codec
- num values
- offset of first data page
- offset of first index page
- compressed/uncompressed size
- extra key/value pairs
column "b" meta data
Row group 1 meta data
Footer length (4 bytes)
Magic Number (4 bytes): "PAR1"

Figure 2.5: A schema of actual Parquet file structure [2]

3. **Bit packing** – Values are packed into the exact number of bits necessary to express them, rather than into multiples of a byte. For example, eight 3-bit values are packed into three bytes. This encoding is particularly important for definition and repetition levels, as their values typically take 3 bits or less. Parquet also defines a hybrid between RLE and bit packing, to optimize for the important case where most of the level values are equal (0 for sparse columns, max for columns where NULLs are exceptional).

4. **Dictionary encoding** – If the number of unique values in the column in small, write them all to a special dictionary page at the beginning of the column chunk, and then encode the values as indices to the dictionary page.

5. **Delta encodings** – If the difference between neighbouring values is small, what is written is not the values themselves, but the deltas (differences) between them. Deltas are often smaller than the absolute values, meaning they can be bit-packed better. This works for numbers as well as strings (in this case, we reuse the common prefix). Delta encoding is particularly useful for encoding time series, which change continuously, and so have small deltas.

6. **Byte split stream** – If each value has a length of exactly k bytes, list of such values in a memory can be thought of as a table of k columns written row-by-row. This encoding changes that into column-by-column fashion. Although this does not directly compress the data, it makes it easier for other compression algorithms to work on it afterwards, as the uniformity is potentially higher. Used with floating-point data.

On top of its encoding schemes, Parquet uses various general-purpose compression algorithms including `snappy`, `gzip`, `lzo`, `brotli`, `lz4` and `zstd`. They too work on small units (pages), maximizing the potential of choosing different schemes in an adaptive fashion, depending on data at hand.

2.1.5. Apache Arrow

There already exists a functional C++ Parquet library. Originally a small independent codebase, it was absorbed into a project named Apache Arrow about two years ago.

Arrow was originally thought as cross-language in-memory format that would allow easy exchange between different data processing frameworks, eliminating the need to convert one format into another every time. As it grew, other functionalities were added, and it found itself in need of on-disk data format. For that, Parquet was chosen.

Now, Arrow is a development platform spanning hundreds of thousands lines of code. When Parquet library was moved there, effort was put into deduplicating functionality and eliminating transitive dependencies, and the library blended into a part of that big, tightly integrated framework, limiting its reusability.

2.2. ScyllaDB

ScyllaDB is a distributed NoSQL database for big data applications. The 'NoSQL' term does not concern Scylla's query language, but is a blanket term for databases which do not follow the relational model – Scylla indexes records only by their primary key and is unable to perform complex operations, fundamentally important for relational databases (like `JOIN`s), but in exchange it offers performance, scalability and network fault tolerance orders of magnitude higher.

Scylla is a drop-in replacement for the popular Apache Cassandra, reimplemented in Modern C++ (as opposed to Cassandra's Java) and highly optimized for modern server hardware using techniques inspired by high-performance computing. Thanks to this, Scylla achieves efficiency 1-2 orders of magnitude better (both in terms of latency and in terms of throughput) than its predecessor.

CQL

Because one of the ScyllaDB goals is to serve as a drop-in replacement for Apache Cassandra, it operates with the same query language – CQL. Its syntax and semantics are very similar to SQL, but are limited in comparison. In particular, the language allows `INSERT`s, `SELECT`s, `UPDATE`s and `DELETE`s, and a limited use of `WHERE` clauses.

Memtables, SSTables, flushing and compaction

Due to its distributed nature and focus on fast writes, Scylla primary data structure is the LSM (log-structured merge) tree: when new inserts, deletes or updates arrive, Scylla adds them to an in-memory ordered tree (called Memtables). When this tree grows too big, it is written (in order) to a new file on disk (in the SSTables format), which is append-only, in the process called 'flushing' (this is the point where our Parquet module is plugged in). A consequence of this append-only approach, is that deletes and updates are written as new values (in case of deletes, special values called 'tombstones' are used) with attached timestamps, instead of overwriting the values stored in older files. During reads, the database has to look at all files of a given table, in chronological order. A separate process (compaction) merges smaller files into new, bigger files in the background. This consolidates multiple consecutive changes to a particular row into a single bigger change, which improves read performance.

2.2.1. Seastar

ScyllaDB is written in Seastar, a free and open-source asynchronous programming framework for C++14 (and higher), purpose-built for Scylla. Although it was designed specifically for use in the database, it is now an independent project, ready to be used for creating other high-performance server-side applications. Example third-party users of Seastar include Pedis (a Redis replacement) and Redpanda (a Kafka replacement).

Seastar's key features are:

- low-overhead, future-based concurrency model

```
// the usual C++ for loop, blocks for the whole duration
for (int i = 0; i < N; ++i) {
    func(i);
}

// Seastar-style loop, potentially yielding control every iteration
seastar::do_for_each(
    boost::counting_iterator<int>(0),
    boost::counting_iterator<int>(N),
    [] (int i) {
        func(i);
    }
);
```

Listing 2.4: Loop style comparison

- sharded (every OS thread of Seastar is independent, and tied to it's own physical CPU core for optimal scheduling and cache usage patterns), shared-nothing, lock-free architecture. Seastar's design eliminates the need for thread synchronization, increasing performance and simplifying server design
- a custom, DPDK based TCP/IP stack in userspace, allowing zero-copy networking
- direct (not buffered by the OS), nonblocking disk I/O

Reactor stalls

The price of those features is invasiveness: most software components not written in Seastar can not be easily reused in Seastar-based applications, and the other way around. For example: any library which creates threads or uses blocking system calls (like the ubiquitous `open()`, `read()` and `write()`) is not viable. (When a system call blocks, the entire Seastar process is blocked, which destroys Seastar's low-latency guarantees and messes with its scheduler). For a similar reason, any library performing complex computations (longer than a few milliseconds) can't be easily used – Seastar's scheduler is (by design) not preemptive, so long loops steal control from the scheduler and starve other tasks, causing latency spikes.
Such incidents are called 'reactor stalls' in Seastar terminology and avoiding them requires a specific programming style, care and discipline. Existing libraries usually need to be modified or rewritten to be used with Seastar, which was the case with our Parquet library. Consider a comparison between a normal C++ loop, and a stall-proofed Seastar equivalent (Listing 2.4).

Green threads

All computations running inside Seastar tend to be boxed in `seastar::future` – Seastar's implementation of the promise-future model. However, futures tend to be cumbersome to the programmer, so Seastar offers an alternative: a custom implementation of green threads called `seastar::thread` which makes it possible to write asynchronous code in direct style, making porting existing code to Seastar straightforward – example in Listing 2.5.

Unfortunately, the runtime performance of this convenient feature was found to be worse than that of futures. For this reason, users of Seastar are discouraged from using `seastar::thread`

```
seastar::future<uint64_t> log2_file_size(sstring name) {
    return seastar::async([name] () {
        file f = seastar::open_file_dma(name).get0(); // yields
        auto size = f.size().get0(); // yields
        return seastar::log2ceil(size);
    });
};
```

Listing 2.5: Direct style using seastar::async

```
seastar::future<uint64_t> log2_file_size(sstring name) {
        return seastar::open_file_dma(name).then([] (auto file) {
            return file.size().then([] (auto size) {
                return seastar::log2ceil(size);
            };
        };
    });
};
```

Listing 2.6: CPS version not using seastar::async

in performance-sensitive contexts, and asked to use futures, writing in continuation-passing style (Listing 2.6) instead, whenever possible. Consequently, we haven't used `seastar::thread` in our project.

Chapter 3

Our project

As mentioned before, Scylla's developers were interested in Parquet for two reasons: they wanted to help their users migrate data directly between Parquet and Scylla, and they viewed Parquet as a potential secondary storage format for internal use in certain workloads in Scylla. For those tasks, a Seastar-based Parquet library was needed, and we were requested to create that library. We were also asked to provide a proof-of-concept implementation of translating Scylla's internal data storage format to Parquet.
At the end, our delivered products include:

- A Parquet library for Seastar named `parquet4seastar`.
- A tool for Parquet-to-Scylla data import named `parquet2cql`.
- A proof-of-concept Scylla feature for writing internal data to Parquet.

3.1. Overview of the architecture

A high-level diagram of our product is shown in Figure 3.1. The functionality can be split into two paths: importing (reading) and exporting (writing) the data:

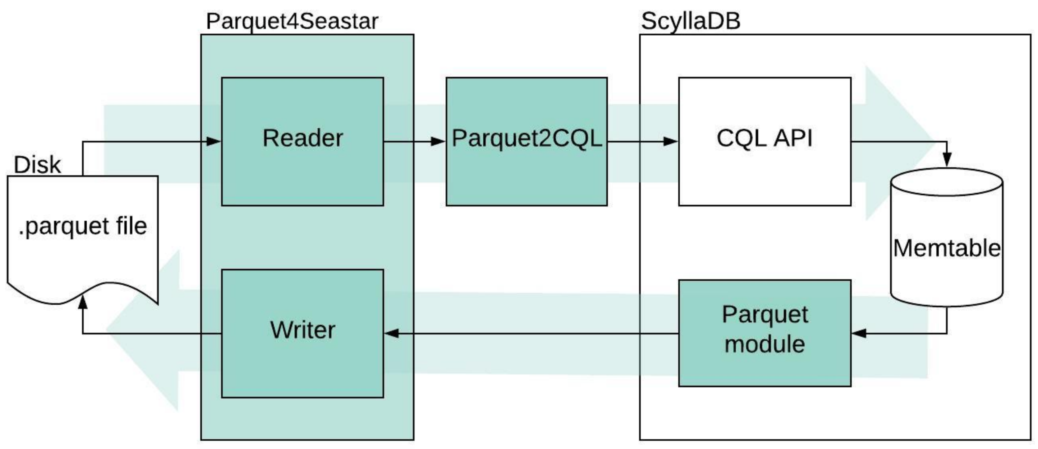


Figure 3.1: Architecture diagram (delivered components in green).

1. The reader half of `parquet4seastar` library is used by `parquet2cql`. It is a standalone tool that feeds the contents of the input `.parquet` file into ScyllaDB, by outputting a series of CQL statements, creating the appropriate types and tables, and inserting the data. ScyllaDB then processes these statements and actually puts the data into a memtable. This whole process is visualized in Figure 3.2 and explained in the sections below.

2. Inside ScyllaDB, in parallel to existing code responsible for flushing Memtables into SSTables, a new module is created for the purpose of directly flushing Memtables into `.parquet` files, using the writer half of the `parquet4seastar` library (visualized in Figure 3.3).

3.2. parquet4seastar

The Parquet library for Seastar was the main deliverable of our project. It was written from scratch (except for some optimized bit-packing routines borrowed from Apache Arrow[1]) in C++17.

3.2.1. Design

The hierarchic design of our library mirrors the structure of the data it's handling. As shown in Figure 3.2:

1. The entry point to the library is the `file_reader` class. It opens a `.parquet` file, and deserializes its metadata using a parser generated automatically from the Parquet Thrift definition (published by the Parquet developer team). Then, the schema is preprocessed to a useful form (for example, the definition and repetition levels of columns need to be computed). After that, users of the `file_reader` can view the schema and spawn `column_chunk_reader` for each of the columns which interest them.

2. Then, the user can read batches of data from the `column_chunk_reader`s, which, under the hood, read pages from corresponding column chunks sequentially (using Seastar's `input_stream`), deserialize the page headers, decompress and decode page bodies, and eventually output a stream of (value, definition level, repetition level) tuples. Some key pieces of code in this part, specifically the ones used for decoding, were borrowed from Apache Arrow. The `column_chunk_reader` is the basic low-level, low-overhead interface to Parquet – it does not interpret the data, only decodes it and exposes it to higher layers.

3. Parquet defines the interpretation of data using logical type annotations. For example, a column of `BYTE_ARRAY` might represent arbitrary-precision integers, UTF-8 text, keys to a dictionary, etc. `reader_schema` decodes and validates those annotations, and exposes them to the user.

4. Finally, the library provides a high level interface, the `record_reader`, which aggregates multiple `column_chunk_reader`s using the `reader_schema` and outputs complex records using Dremel's record assembly algorithm. We decided against outputting a concrete representation of the record, because this would require additional allocations, serializations (to aggregate the fields into a single object in memory) and deserializations (so

[1]Which in turn borrowed it from D. Lemire, who in turn borrowed the idea from [4].

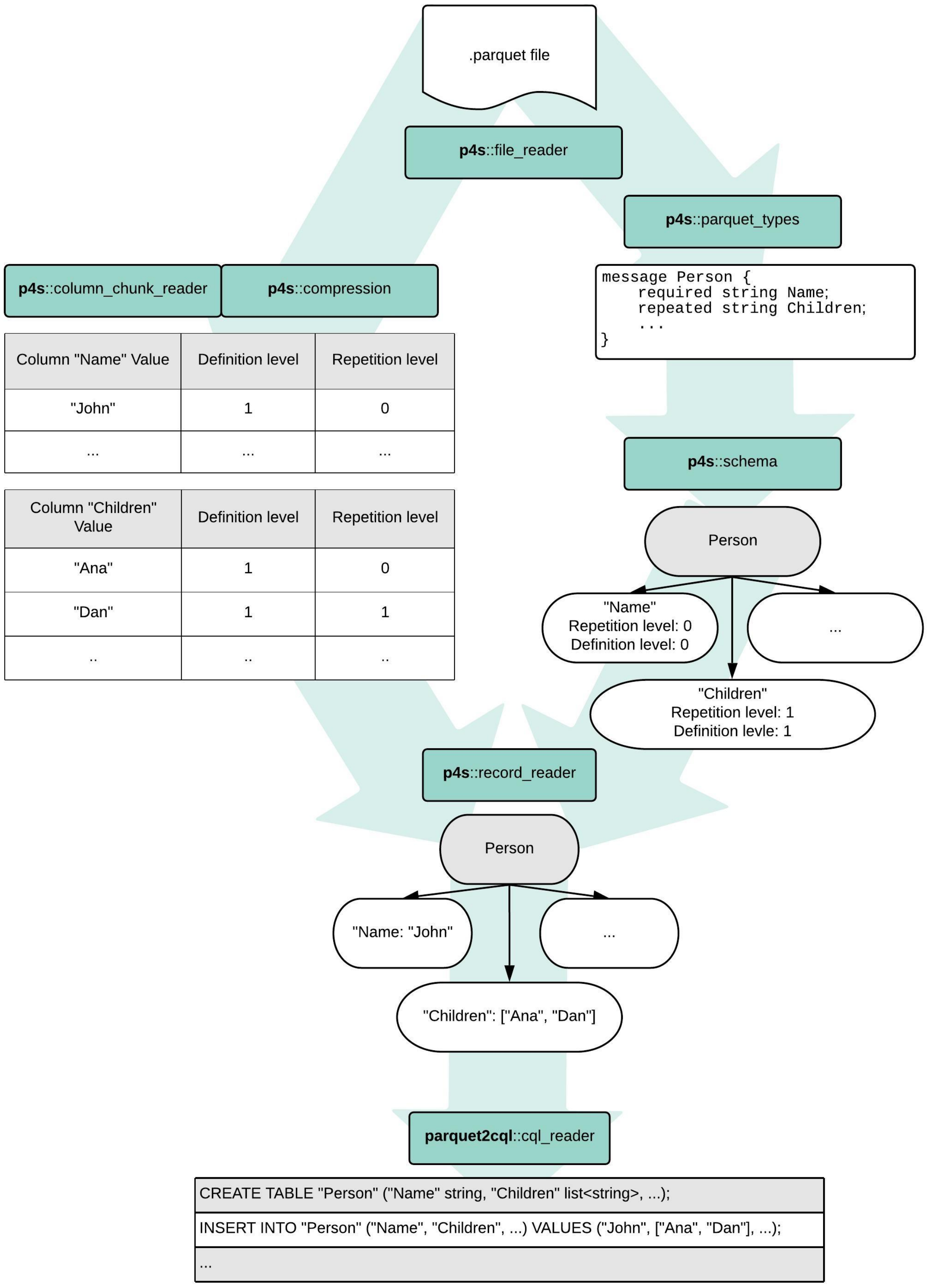
that the consumer can iterate over the fields). Instead, the user can consume the record in parts, by providing a `Consumer` object which implements appropriate callbacks, such as `start_list`, `end_list`, `append_value`, etc.

The design of the writer half of the library mirrors the design of the reader half, providing the same operations, but in the reverse direction. However, we did not provide a high-level writer counterpart to the `record_reader`. Implementing a `record_writer` would either require settling on a particular representation of an assembled record in memory, or require the user to provide input fields in exact order and both solutions were unsatisfactory. For this reason, we implemented the record shredding needed for our Scylla module in the module itself.
When we were starting the project, an optional page-level encryption feature was added to Parquet. We have decided not to support this feature. It would complicate the library quite significantly, and we couldn't imagine a usage scenario for our library which involves it. Therefore, our library can read all Parquet files, except encrypted ones.

3.2.2. Implementation notes

1. **Seastar compatibility:** the implementation is fully Seastar-compliant, including I/O, memory management and concurrency. To prevent reactor stalls (Section 2.2.1), loops which can run for arbitrarily long amounts of time are written using Seastar primitives. Unfortunately, we found it impossible to prevent reactor stalls in general. The theoretical page size limit in Parquet is 4GB. This means that, since not all compression algorithms used in Parquet provide a streaming interface, in the general case a page decompression might cause a reactor stall. Preventing this behaviour would require modifying the compression libraries to be Seastar-friendly, which exceeded the scope of our project.
   That said, this should never occur during normal usage, because pages in Parquet are supposed to be several KB in size. Only malicious or guideline-incompliant files can cause the reactor stall, so when maximal latency is of concern, the library user can put a fixed size limit on pages which should be considered valid.

2. **Memory management:** the memory is owned at the lowest possible level of the reader stack, and only non-owning pointers are propagated up. This way, memory is reallocated and copied only when strictly necessary. For example, when the Parquet file has compression or decoding turned off, parts of the stream responsible for decoding and decompression simply pass pointers up.
   We have also decided to output reference-counted strings from the reader instead of copying strings to the output, to maximize the effectiveness of dictionary encoding. In other words, if multiple copies of the same string are read from a dictionary-encoded page, the user will receive multiple pointers to the same string in the dictionary table, instead of several copies of the string. Ultimately, we found that our optimizations have a minor effect overall, except when processing long strings, which is not a typical workload for Parquet files. This went against our preconceptions about the cost of memory management.



```
message Person {
    required string Name;
    repeated string Children;
    ...
}
```

Column "Name" Value	Definition level	Repetition level
"John"	1	0
...	...	...

Column "Children" Value	Definition level	Repetition level
"Ana"	1	0
"Dan"	1	1
..	..	..

```
CREATE TABLE "Person" ("Name" string, "Children" list<string>, ...);
INSERT INTO "Person" ("Name", "Children", ...) VALUES ("John", ["Ana", "Dan"], ...);
...
```

Figure 3.2: Read path diagram (code components in green, produced data in grey/white).

3.3. parquet2cql

On top of the `record_reader` we have implemented a command-line interface tool which converts Parquet files to `CREATE` and `INSERT` statements in CQL, Scylla's query language (actually: 'Cassandra Query Language'). The tool maps all logical types defined by Parquet to most closely matching logical types defined by CQL, defines a CQL UDT (User Defined Type) for every **group** node in the Parquet schema, creates the table and then prints an appropriate `INSERT` for every row in the file (example output is shown in Listing 3.2). The design of `record_reader` lends itself well to this task.

parquet2cql is a starting point for the direct Parquet-to-Scylla import path. While it can be used for importing `.parquet` to Scylla, it's not suitable for big migrations – printing the `INSERT` statements as text is slow. To be useful for real-world migrations, the tool should be modified to output the records to an optimized Scylla driver, rather than to text. This was deemed outside of the scope of our project.

The purpose of `parquet2cql` is thus mainly to serve for testing and validation of the library, as well as an example documenting its usage.

Still, we found the tool useful for viewing Parquet data in a human-readable form. Surprisingly to us, other existing tools (like `parquet-cat` from the `parquet-tools` package or `parquet-reader` from Apache Arrow) we wanted to use for this task (for example, to look at `.parquet` generated by our Scylla module), were either lacking in features (that is, they could not understand the current version of Parquet) or were plainly incorrect (for example, `parquet-reader` erroneously confuses empty **group** nodes in the schema with atomic nodes). Eventually, we ended up using our own tool.

A usage example is shown in Listing 3.1. User has to specify the input file, as well as the destination table name (`--table` parameter) and primary key name (`--pk` parameter). The result – a sequence of of CQL statements – is then printed to the standard output.

```
parquet2cql --file ./example.parquet --table "table1" --pk "primarykey1" > output.cql
```

Listing 3.1: parquet2cql usage

```
CREATE TYPE "parquet_udt_0" ("floatField" float, "intField" int);

CREATE TABLE "parquet"("primarykey1" bigint PRIMARY KEY, "mapField" frozen<map<text, int>>,
    "listField" frozen<list<"parquet_udt_0">>);

INSERT INTO "parquet"("primarykey1", "mapField", "listField") VALUES(1, {'key1': 1, 'key2':
    1}, [null, {"floatField": 1.000000e+00, "intField": 3}]);
```

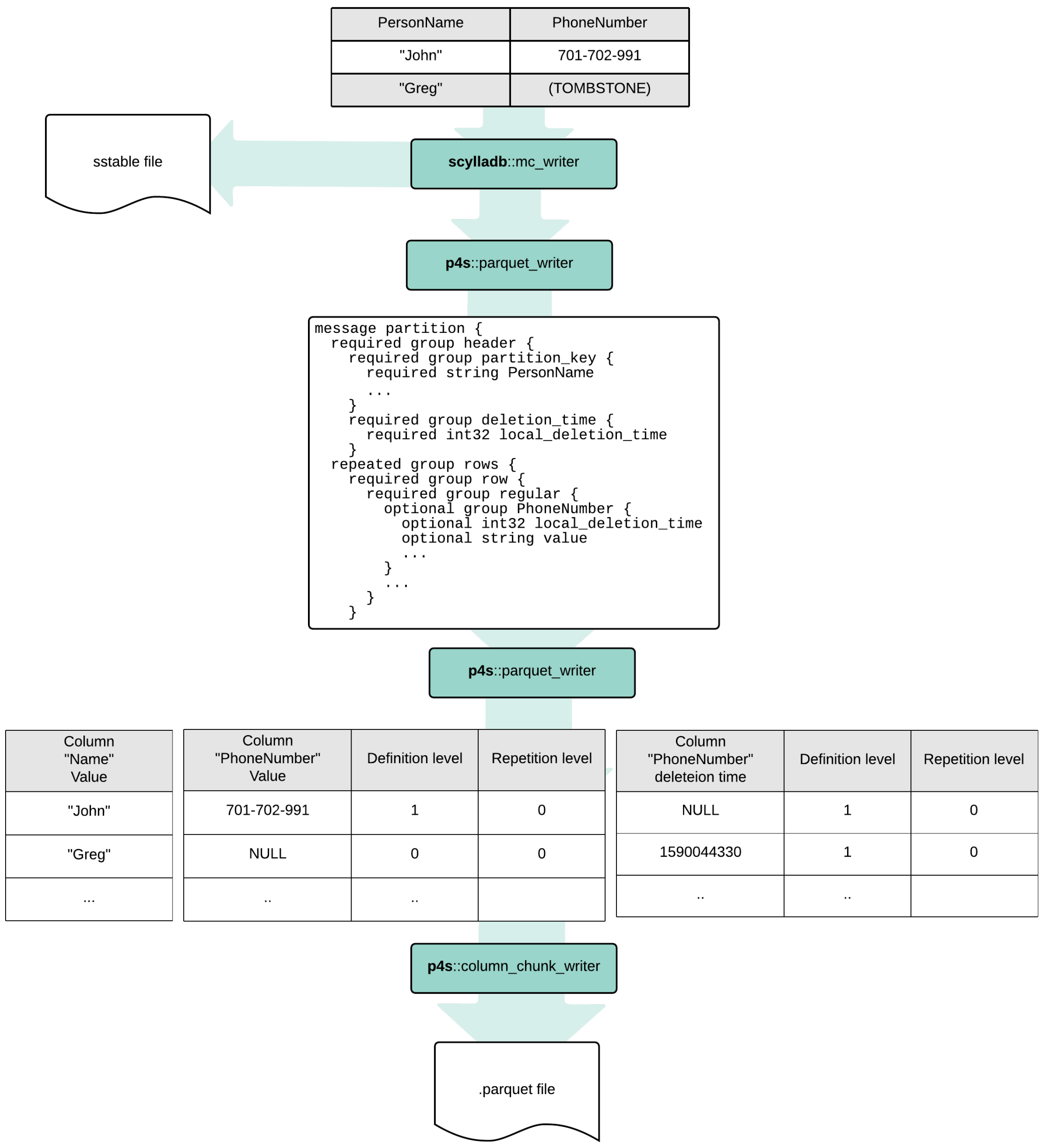
Listing 3.2: parquet2cql example output

3.4. ScyllaDB module

The final part of our project involved the design and implementation of a proof-of-concept Scylla module, which would demonstrate our library and it's potential usage in Scylla, by writing Scylla's data to Parquet, alongside Scylla's default storage format, SSTables [5]. We found that the schema of SSTables is well representable as a Parquet schema (Appendix B).

One notable difference between SSTables and our corresponding Parquet is that Scylla and Cassandra do not write full values of timestamps (which are associated with every value in the database, so its important to store them efficiently), but write deltas (differences between a value, and the preceding value) instead, encoded as VLQ (variable-length quantity) to save space. Since parquet does not support a VLQ encoding, writing deltas would be pointless. Instead, Parquet provides its own `DELTA_BINARY_PACKED` encoding, which achieves the same goal of space reduction, except by using bit-packing instead of VLQ for the deltas.

Thanks to the thought-out design of Scylla, our module achieved its goal while requiring minimal modifications to Scylla (only 50 lines of changes in existing code, not counting the module itself).



PersonName	PhoneNumber
"John"	701-702-991
"Greg"	(TOMBSTONE)

```
message partition {
  required group header {
    required group partition_key {
      required string PersonName
      ...
    }
    required group deletion_time {
      required int32 local_deletion_time
    }
  repeated group rows {
    required group row {
      required group regular {
        optional group PhoneNumber {
          optional int32 local_deletion_time
          optional string value
          ...
        }
        ...
      }
    }
```

Column "Name" Value
"John"
"Greg"
...

Column "PhoneNumber" Value	Definition level	Repetition level
701-702-991	1	0
NULL	0	0
..	..	

Column "PhoneNumber" deleteion time	Definition level	Repetition level
NULL	1	0
1590044330	1	0
..	..	

Figure 3.3: Write path diagram (code components in green, produced data in grey/white).

Chapter 4

Execution

Although we found the final outcome satisfactory, the road to it was long and bumpy. This section tells what the problems that we faced were and how they were overcome. As we worked mostly in a sequential fashion, on one product at a time, the project naturally partitioned into iterations. We did not, however, formally realize any of the software development methodologies usually associated with this notion.

4.1. Part I: Planning

Since the beginning, our clients – the Scylla developer team – made it clear that the main goal of the project was to produce a working Parquet library for Seastar. Everything else would be a bonus, and would depend on the library anyways. So, naturally, we had to start with the library, but weren't sure what approach to take. As we mentioned before, there already exists a Parquet library for C++, inside the Apache Arrow project. So, the natural question arose: should we aim to develop a new library from scratch, or attempt to adapt the existing one?

Since the library was supposed to be suitable for use in Scylla, the Arrow implementation would have to be modified to be Seastar-compatible. Unfortunately, futures are invasive: any function which calls an asynchronous function, has to be an asynchronous function. Therefore, all functions which indirectly called I/O procedures would have to be rewritten to be asynchronous functions. If the rewrite of IO code succeeded painlessly, memory management and concurrency could also prove hard to replace. On top of that, controlling reactor stalls meant that we would have to review and potentially correct every loop in the code. And even if all of that was taken care of, Apache Arrow contains about 300k lines of C++ – for comparison, the entire Seastar codebase is about a third of that. Forking Arrow and modifying only the I/O parts of it would be unacceptable to the Scylla team for maintenance reasons. The only way we could use the existing library would be to extract the relevant pieces.

However, we felt compelled to read and understand the existing solution first, even if we were to write our own. Perhaps the additional effort of modifying some parts of it would not be so big in proportion? Additionally, Arrow had a convincing test suite, so we thought it was feasible to port the library to Seastar in a continuous fashion, by getting feedback from the tests after every change. Most importantly, reinventing an existing library from scratch went against traditional programmer wisdom.

These seemed to be major arguments for reuse. It sure seemed less fun, and perhaps we were importing some unknown technical debt, but existing testing and optimism about changing the code as we read it convinced us to do it. Our advisors, Scylla developers, also found it to be a reasonable approach.

4.2. Part II: Parquet library reuse

Before starting, we had another decision to make – how should we extract the Parquet parts from Arrow? We could either try to do it top-down, or bottom-up. Top-down, meaning: start with the entire Arrow repository, identify the parts which interest us, and then meticulously cut out the pieces which the interesting parts don't indirectly depend on. Bottom-up, meaning: start with a clean repository, and keep copying in reusable parts from Arrow, modifying them to be Seastar-compatible, until we are able to combine them into the features we want.

We figured that a bottom-up approach should result in better code – if we could extract only the bits that were needed, we would end up with no cruft. And so, we started with the bottom-up approach. Unfortunately, we soon understood that the code was less reusable than we hoped for – the internal dependencies were too numerous for us to handle. Specifically, we found ourselves moving high volumes of code around without a way to build it, and get any sense of progress – because some dependencies were always missing.

That's why we soon dropped that approach, and decided that we have to do it top-down. This meant we would very likely end up with lots of code we are not comfortable with, but at least we would get some feedback from the compiler and Arrow's test suite about what we were doing.

The idea worked – we started to cut out non-Parquet code from Arrow meticulously, and transition the I/O layer to Seastar. After several weeks, we managed to get down to around 30k lines of code, and found reducing that number further impossible without fundamentally rewriting the library. We have also successfully rewritten all I/O routines to be Seastar-based, and got a viable product.

Unfortunately, the result was unsatisfactory from a software engineering point of view. Not only was 30k lines of code still too much, we were also stuck with a weird hybrid of Arrow's and Seastar's memory management, testing and build systems. Also, the code quality was low, because many conventions used in Arrow translate badly to Seastar. For example, potentially-blocking destructors (e.g. closing a file) are very hard to port – every call to a destructor (and those calls are invisible/implicit in C++) has to be located and redone manually. Another example is using the return value of functions to pass error codes, while returning the actual results through an argument. This was the error handling convention used in Arrow, but Seastar discourages it heavily. In asynchronous functions, output arguments are ugly and error prone, because they invite bugs related to pointers' lifetimes – the output argument has to be manually kept alive beyond the function it is passed to. Avoiding memory corruption in this situation requires great care and C++ has little facilities to help the programmer with it.

Overall, the resulting product was messy and it was clear it was created with different goals and conventions than our clients had in mind. Because of those problems, we eventually decided to discard our work and rewrite the library from scratch.

While the first iteration of the library was unsatisfactory, it was functional, and we were asked to use it for a proof-of-concept Parquet module for Scylla. We then split our efforts, and started working on the new library and on the database module in parallel.

4.3. Part III: parquet2cql

Using the knowledge gained earlier, we made quick progress on the new library, and reimplemented most of Arrow's Parquet functionality. Admittedly, some features were lost (for example, writing min/max statistics for data pages), but we have also added some features not provided by Arrow, like the record assembly algorithm (`record_reader`). Ultimately, we arrived at the result described in chapter 3, which was smaller, more independent and more Seastar-idiomatic, and we were satisfied with the result. We have tried to beat the Arrow implementation in benchmarks, too, but it proved difficult. In the end, both libraries ended up similar in performance, we summarize the comparison in chapter 6.

Between writing the reader half and the writer half of the library, we have created the parquet2cql utility, to serve as a demonstration and test of our reader. We later found it a useful tool for inspecting Parquet files in general as we worked on the writer and the Scylla module.

4.4. Part IV: ScyllaDB to Parquet

In parallel with the rewrite, we started working on the Scylla module, finding and investigating relevant parts of Scylla's source code, determining the points where our module should intercept the data, and working out the translation between SSTables and Parquet.

The first iteration of the module was written using the first iteration of the library. It supported only a small subset of possible data types and column types, and did not handle metadata (like timestamps).

We happened to finish the library first, and then we unified our efforts on completing the module. We didn't spend much time on its second iteration, but since we didn't run into any new problems (other than the ones outlined in chapter 5), we managed to deliver a prototype which covered most of the cases. (More specifically, it ignored some kinds of collection types and tombstones. They wouldn't be particularly hard to implement, but they are rarely used features of Scylla and were not important for the prototype). We then performed some tests to observe how effective the transition from SSTables to Parquet was.

4.5. Part V: Benchmarking, polishing

Although some preliminary benchmarks were done along the way, it was the final part of the project when they started to be developed extensively. There were a number of difficulties performing them, stemming from the peculiarities of Seastar-based application architecture, such as the fact that the framework does active waiting, thus limiting the option of seeing what time is actually spent doing useful computations, as opposed to waiting for IO. Full description of benchmarks results is given in chapter 6.

The remaining time was spent on polishing the code. We added some missing encodings and support for compression standards (using external libraries) as described earlier – brotli [6], lz4 [7] and zstd [8], zlib [9] and snappy [10]. Tests for the ScyllaDB integration were also added.

Chapter 5

Tooling

Our work described in this thesis concerned big C++ projects: Seastar, Scylla and Apache Arrow. We have found the language and its programming environment frustrating to work with.

The most aggravating problem was long compile and link times, which have severely limited our productivity and work enjoyment. On our – reasonably modern – laptops a full compilation of Scylla (in the fastest available build configuration, using all CPU threads) took above an hour. An incremental recompilation of Scylla after any change took more than 30 seconds per modified translation unit. This would often result in a workflow where we would write a piece of code, wait 30 seconds for some of it to compile, fix the reported typo or type error in 5 seconds, then wait another 30 seconds for the next error report, repeating the cycle until all the minor errors were ironed out. Long build times also meant that we had to wait a long time to view the effect of any tweaks we made to our program, for example changing a constant parameter, or adding some message to the logs. This all made testing and debugging frustrating, as we found ourselves spending an unreasonably big fraction of the edit-compile-debug cycle on waiting idly for the compiler.

One useful tool we used to partially amend this problem was `ccache`, a wrapper around the C++ compiler that robustly caches the results of compilation (including compiler warnings) for given translation units. This shortens the compilation time to a fraction of second, provided the exact same file has already been compiled once before. This allowed us to use version control systems comfortably, without worrying about the drastic cost of recompiling the entire repository when switching between versions.

Another tool that slightly enhanced the build times was `ninja` (replacing the traditional GNU `make`). Fortunately, Seastar, ScyllaDB and Apache Arrow all provided it as an option.

The choice of a linker also happened to be significant. We found `lld` to be faster than `ld` and `gold`.

The aforementioned problem of waiting 30 seconds for a type check to happen should normally be solved by an IDE and/or a language server. Unfortunately, C++ is an exceptionally difficult language to provide tooling for (thanks to its usage of C preprocessor, and the intractable semantics of C++ templates, among others), and indeed we found the available tools inadequate. The best C++ viewing and editing tools we could find (`clion` and `clangd`), were slow, unstable and consumed prohibitive amount of computing resources when faced with the Scylla repository on our laptops. Ultimately, we have resigned ourselves to plaintext-based tools (like `vim`, `fzf` and `ripgrep`), and found them more productive.

It's worth noting that the upcoming C++ revision – C++20 – will bring tools supposed to alleviate some of problems we have complained about. In particular, C++ modules should lower compile times significantly and C++ concepts should give C++ tools a better understanding of template semantics. Unfortunately, C++20 is not supported by any major compiler yet.

Another (much lesser than the above) issue in our project was dependency management. C++ does not have a standard build system, which in practice means that developers have to deal with dependency conflicts manually, and that every project big enough has its own build conventions, usually incompatible with others. Such was the case here.

The problem of dependency conflicts can be solved with containerization (i.e. with `docker`). Those of us, who had problems with the library versions provided by their operating system's package manager, have used this tool.

As for the incompatible build systems, Arrow and Seastar fortunately both use `cmake`, which comes the closest to being a de facto standard for C++, which made working with their dependencies reasonably easy, thanks to available CMake documentation. Scylla however, for historical reasons, uses a custom, unstructured Python script as its build generator. Because of that, integrating our library with Scylla turned out to be a much harder task than it ought to be.

We have extensively used `git` for version control, `github.com` for repository hosting, Facebook Messenger for messaging, Google Meet for meetings, Lucidchart for creating diagrams and Overleaf for writing this thesis in LaTeX.

Chapter 6

Benchmarking

We have measured two unrelated things:

1. The run time performance of parquet4seastar (our Parquet library) versus Apache Arrow (which is the only other existing C++ implementation of Parquet that we are aware of). We have performed this measurement to assess the quality of our implementation.
2. The disk usage of Parquet (written by our Scylla module) relative to SSTables (Scylla's native format). Note that this is a property of the formats, and not the implementations. We were asked to perform this measurement by our clients.

6.1. Performance of parquet4seastar relative to Arrow C++

6.1.1. Scope of the benchmark

We have benchmarked the core (and most performance-sensitive) functionality of both libraries: sequential batch I/O for a given column. That is, we have measured how long it takes for both libraries to sequentially transfer (in either direction) a Parquet column (with a particular combination of parameters) between a user-provided buffer in memory and a Parquet file on disk, batch by batch.

For the purpose of this discussion, the relevant parameters of a Parquet column are its type, encoding, and compression. The type is inherent to the data and describes its physical form. It can be, for example, `INT32`, `FLOAT` or `BYTE_ARRAY`. The encoding is one of the several Parquet-specific simple data compression algorithms. See Section 2.1.4 for examples. The compression is one of the external, general-purpose compression algorithms employed in Parquet, like `GZIP` or `SNAPPY`.

Each possible (type, encoding) combination is handled by a different piece of library code and ought to be benchmarked separately. In contrast, the code responsible for compression is shared between parquet4seastar and Arrow, because it is provided by external libraries. For this reason, one could assume that compression should be turned off for the benchmarks. However, as compression is a relatively expensive operation, it changes disk access patterns, and we expected that it could uncover some issues which would otherwise stay unnoticed.

We decided to test all combinations of types `INT32`, `INT64`, `BYTE_ARRAY`, encodings `PLAIN`, `RLE_DICTIONARY`, and compressions `UNCOMPRESSED`, `SNAPPY`. The chosen types are representative of all types (for example, `INT32` is handled exactly the same as `FLOAT` in both libraries), except for BOOLEAN (which we decided not to measure, because we have copied the relevant code for that type from Arrow). The chosen encodings were selected solely because they are the only ones supported by Arrow. SNAPPY was chosen due to its speed – a slower compression (like GZIP) would have dominated the run time, hiding the differences between the two tested programs.

6.1.2. Benchmark details

Each of the tests consisted of writing/reading one Parquet file containing one column, with a single encoding and a single compression algorithm. We have written two small tools, one using Arrow and one using parquet4seastar, for this task. The times presented in the result section were obtained by measuring the full run time of those programs, and subtracting their startup times (loading dynamic libraries and – in case of Seastar – setting up its CPU and memory pool), which we confirmed to be constant. Before each reader test, page cache was dropped. After each writer test a disk sync on the output file was performed (length of the sync was added to the overall run time). We set the Parquet page size to 64KB and row group size to a value between 1MB and 10MB (varying those parameters didn't visibly affect the results). The number of rows in each file was chosen so that every test would take about 8 seconds (long enough to trump minor variations in run time, short enough to keep the file sizes maintainable on our machines). The benchmark was repeated 5 times for each case. By 'short strings' we mean strings of length 8, and by 'long strings' we mean strings of length 80.

6.1.3. Test environment

- CPU: Intel Core i5-5200U
- Motherboard: ThinkPad X250
- Storage (SSD): Samsung MZ7LN128 (SATA III)
- Linux version: 5.6.15
- Filesystem: ext4
- Arrow[11] version: git commit dc8cd4fd4d60018e3d195b784c562af04d30bf66
- parquet4seastar[12] version: git commit 8a04e7367ea6e88d2a2e3816eedb603650ec44d6
- Seastar[13] version: git commit 92365e7b87a8cc9506b57a9c6ea4db675dbbc64d

All mentioned software versions can be found in the GitHub repositories of each project. The tools and scripts used to perform the benchmark can also be found in the mentioned parquet4seastar commit.

6.1.4. Results

The following graphs show the relative performance of parquet4seastar against Arrow. The value on the horizontal axis is given by $\frac{\text{parquet4seastar run time}}{\text{Arrow run time}}$. The bars show the averages of results, while the black ticks represent the results of individual test runs.

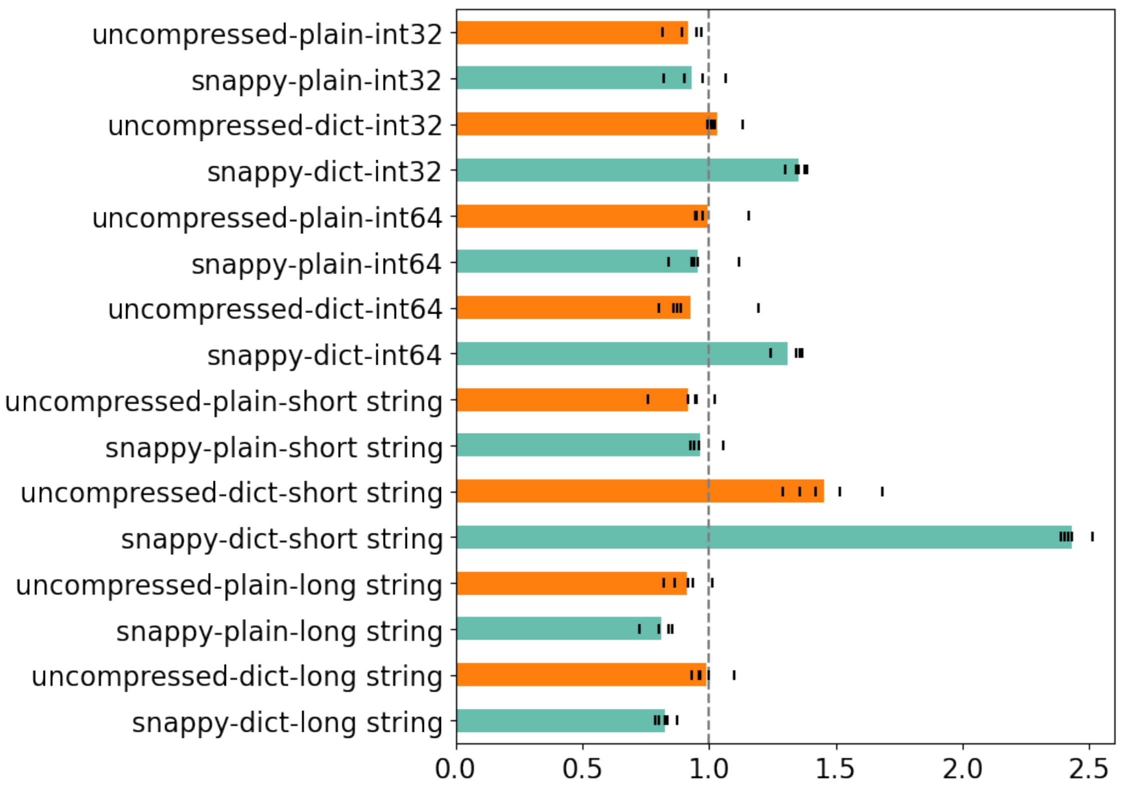


Figure 6.1: Writing time of parquet4seastar relative to Apache Arrow (less means that parquet4seastar was faster).

6.1.5. Interpretation and conclusions

In most of the cases parquet4seastar performs on par or slightly faster than Arrow. This was the expected result. Only one test case showed significant difference in performance: dictionary-encoded short strings. In this case, both reading and writing was more than 50% slower with parquet4seastar. Interestingly, the causes of this difference are unrelated for reading and for writing.

On reads, Arrow outputs strings as raw pointers, which are only valid until the next read from the column. If the user wants to store the strings for a longer time, they need to copy them manually. In contrast, parquet4seastar outputs strings as reference-counted pointers. That is: every string in the output is a pointer to the underlying buffer, which holds a counter. When a pointer is destroyed, the counter is decreased, and when the counter reaches 0, the associated buffer is freed. This allows the user to store the results without manual copying, but incurs a performance cost of managing the counter. This was our deliberate design choice, and some performance difference was expected. However, the magnitude of this difference exceeded our expectations. This functionality will likely be redesigned in the future due to this result.

The difference in write times has a different origin. Dictionary encoding is implemented very similarly in both libraries, using a hash table. However, Arrow uses a custom hash table and the xxHash hash function, while parquet4seastar uses `std::unordered_map` and the `std::hash` hash function from the C++ standard library. The result of the benchmark is a direct consequence of the performance difference between the two hash tables. Apparently the C++

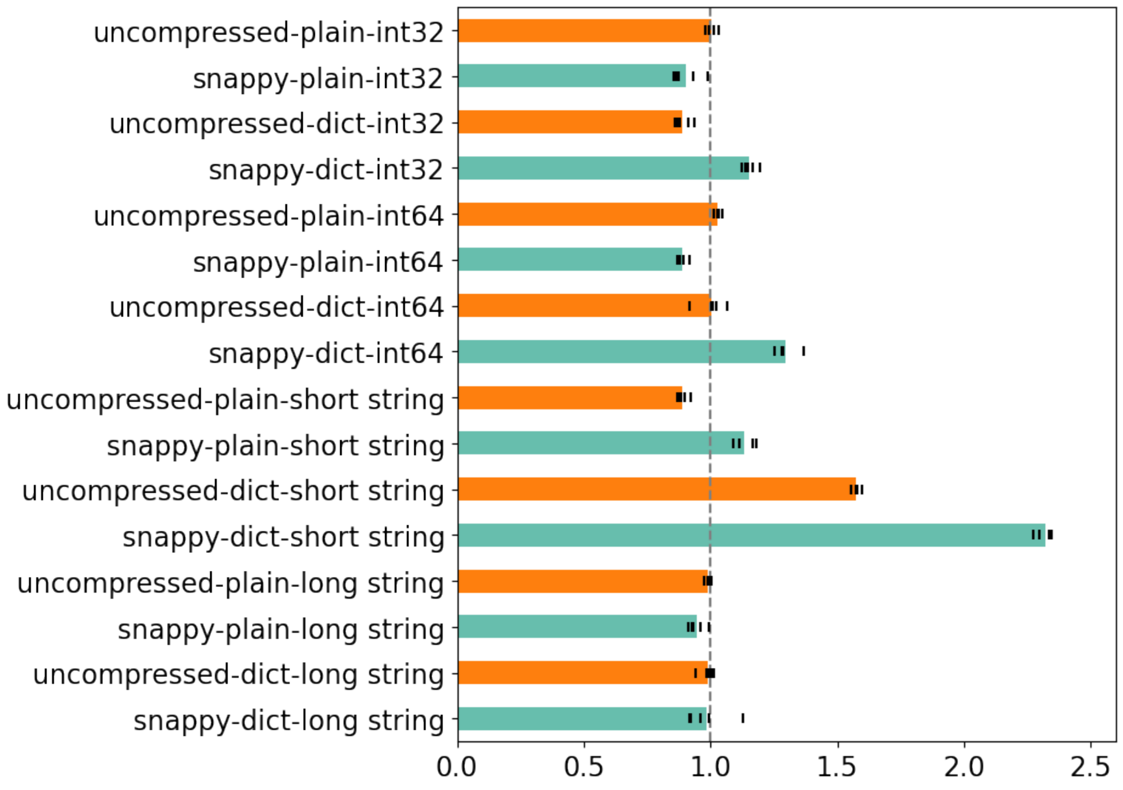


Figure 6.2: Reading time of parquet4seastar relative to Apache Arrow (less means that parquet4seastar was faster).

standard hash table is a poor fit for this use case and is likely to be replaced in a future iteration.

There is one other noteworthy result: parquet4seastar performs significantly worse for a combination of dictionary encoding and Snappy compression, in both reads and writes, for all types except long strings. This is surprising – compression should take the same amount of time in both libraries, as it is performed by the same external library, and so the time difference for the entire program should be no greater than for the uncompressed version of the same test case. We have not been able to explain this behaviour. Perhaps it can be attributed to an unfortunate change in disk usage patterns; further investigation is needed.

6.2. Disk utilization of Parquet relative to SSTables

6.2.1. Introduction

One of the expected advantages of Parquet over SSTables is file size. We were asked to use our experimental module in Scylla to find out how much disk space can be saved by storing Scylla's data in Parquet instead of SSTables.
We have tested three scenarios:

1. A two-column table, where the first column is a unique integer (serving as primary key) and the second column is of 32-bit `integer` type. The integers in the second column are randomly generated. We vary the number of copies – i.e. number of times each integer

appears in the second column of the table – as this is the most important factor in the compressibility of such data.

2. A two-column table, where the first column is a unique integer (serving as the primary key), and the second column is of `string` type. The strings in the second column are alphanumeric, fixed-size and randomly generated. We vary the size of strings and number of copies – i.e. number of times each string of appears in the table – as these are the most important factors in the compressibility of such data.

3. A 100-column table, where the first column is a unique integer (serving as primary key), and the rest of the columns are of `int` type. For each row, we randomly select (a fixed number of) columns that are to be filled with `NULL` values, filling the rest of the columns with random integers. Across the test cases, we vary this number of `NULL` filled columns.

6.2.2. Benchmark details

In all tests, we used version 3.0 of SSTables (also called 'SSTables mc')[5]. All tests were carried with `DeflateCompressor` set for SSTables and `GZIP` compression set for Parquet. `string` columns were written to Parquet with dictionary encoding, and `int` columns were written with delta encoding. All tables in all tests had 1 million rows.

The description of test environment is not included, since the results are independent from it.

6.2.3. Results

Column 'SSTables' shows the size of resulting SSTables file, column 'Parquet' shows the size of resulting Parquet file, column 'Ratio' is the value of column 'Parquet' divided by the value of column 'SSTables' – lower ratio means that Parquet takes less space.

Copies	SSTables [KiB]	Parquet [KiB]	Ratio
1	9451.53	6235.06	65.97%
10	10397.87	7344.08	70.63%
100	12561.91	10150.82	80.81%
1000	13855.67	10182.99	73.49%
10000	14050.46	10196.17	72.57%

Table 6.1: Results for the first scenario. Column 'Copies' describes the number of copies of every unique value in the non-key column.

Copies	String size	SSTables [KiB]	Parquet [KiB]	Ratio
1	16	10217.27	6243.37	61.11%
10	16	12326.25	6658.52	54.02%
100	16	18228.22	7111.31	39.01%
1000	16	21210.09	11254.82	53.06%
10000	16	21539.72	14921.21	69.27%
1	100	14658.61	6240.5	42.57%
10	100	31736.99	6662.75	20.99%
100	100	62857.07	7155.45	11.38%
1000	100	69532.1	41687.74	59.95%
10000	100	70313.91	54906.48	78.09%

Table 6.2: Results for the second scenario. Column ‘Copies’ describes the number of copies of every unique value in the non-key column.

Non-null columns	SSTables [KiB]	Parquet [KiB]	Ratio
0	9306.79	7438.49	79.93%
20	55204.31	83483.61	151.23%
40	84657.85	93474.25	110.41%
60	104103.2	94733.85	91.0%
80	110125.07	94589.21	85.89%

Table 6.3: Results for the third scenario. ’Non-null columns’ indicates how many randomly selected columns in each row had a (random) value – the rest were filled with NULLs.

6.2.4. Interpretation and conclusions

Parquet presents significant (20%-50% in our tests, on average) potential savings in disk usage over SSTables both for numerical data and for string data. The results for these cases can be extrapolated to other possible types in Scylla. However, it is less efficient when the fraction of non-`NULL` values is small, but not insignificant. The reason is: in our translation of SSTables to Parquet, a `NULL` value takes up to 2 bits in the worst case, while in SSTables 3.0 it always takes 1 bit. This is usually mitigated by RLE encoding (Section 2.1.4) in Parquet, but when the levels vary randomly, RLE encoding is crippled. The less biased the levels, the stronger this effect is (strongest at non-`NULL` fractions around 0.5), but it is only noticeable for small non-`NULL` fractions – otherwise the size of values dominates the size of levels.

Chapter 7

Division of work

7.1. Samvel Abrahamyan

Samvel has performed the analysis of Parquet's dependency tree inside Arrow, did the initial cutout of the Parquet library from Arrow, set up its build system, and cleaned up the resulting first library from unnecessary code. In the second phase of the project, Samvel researched the parts of Scylla relevant for our module, and authored its first iteration. He also performed the Parquet vs SSTables tests.

7.2. Michał Chojnowski

Michał took care of adapting the reader half of the first library and cooperated with Samvel on cutting out non-essential dependencies. In the second phase of the project, he was the main author of the rewritten library, and the second iteration of the Parquet module for Scylla. He also maintained our git repositories and build systems.

7.3. Adam Czajkowski

Adam adapted the writer half of the first library together with Jacek. In the second phase of the project, he was in charge of testing and benchmarking for both iterations or the Parquet library, and documented the results.

7.4. Jacek Karwowski

Jacek adapted the writer half of the first library together with Adam. In the second phase of the project, he worked with Samvel on the design of the Scylla module, and implemented the encoding and compression facilities in the rewritten library. He was also the main editor of this paper.

Chapter 8

Summary

The goal of our project was to bring support for Apache Parquet, a popular columnar data storage format, to ScyllaDB, a high performance NoSQL database, whose specialized asynchronous concurrency framework, Seastar, prevented it from using the existing implementations of Parquet.

First, we tried to adapt the existing library – included in the Apache Arrow project – and modify it to be suitable for use with Scylla. We eventually succeeded in this goal, but the result, while functional, was unacceptable for maintenance and software engineering reasons. We found that a direct translation of many idioms and conventions used by the Arrow project to the equivalent Seastar code was jarring, bug-prone and hard to maintain.

For this reason, we decided to write a new library, `parquet4seastar`. We have successfully reimplemented most of Arrow's C++ Parquet functionality using a more Seastar-friendly design, and achieving similar performance. We have also added a feature not supported by the Arrow library: an implementation of the record assembly algorithm, and we have used it to create a CLI tool `parquet2cql` for printing Parquet files to Scylla's query language.

Additionally, we have designed a translation between SSTables (Scylla's native data storage format) and Parquet by providing a proof-of-concept Scylla module for writing its internal data to Parquet files. We have used it to carry out tests comparing Parquet with SSTables in terms of disk utilization. We have found that in most cases Parquet uses significantly less disk space, although it performed worse than expected when the fraction of non-`NULL` values is small, but not insignificant.

We have determined that the biggest setback we faced in our work was long project rebuild times, exceeding 30 seconds per modified translation unit. This greatly hampered our productivity and motivation, especially during experimentation with adjustable program parameters, testing and debugging. Unstable and poorly performing tooling support for C++ also proved frustrating.

The natural possibilities of future extensions to our project include:

- implementing the remaining unsupported features of Parquet (like page level statistics) in the library,
- modifying `parquet2cql` to use an optimized Scylla driver as its output, to create a tool for fast direct migration of data from Parquet to Scylla,

- extending our Scylla module to support more use cases and more secondary features of Parquet, and optimizing its performance.

Appendix A

Attached disk description

As an attachment to the thesis, we present a CD containing all the relevant software. That includes a copy of the `parquet4seastar` git repository, as well as a ScyllaDB fork containing our experimental module. Additionally, we include the text of this thesis, the presentation slides and the demo video.

The main directory of CD has the following structure:

```
parquet4seastar/ # contains the parquet4seastar git repository
scylla/ # contains our fork of the ScyllaDB git repository
thesis/ # contains the source code of this paper
presentation/ # contains presentation slides and the demo video
thesis.pdf # the text of this thesis
```

Building

- For `parquet4seastar`, instructions are provided in `parquet4seastar/README.md`.
- For the ScyllaDB fork, instructions are provided in `scylla/README_BUILD.md`.
- To compile the thesis PDF, run `pdflatex main.tex` in the `thesis` directory.

Appendix B

Mapping from SSTables to Parquet

```
message partition {
    required group header {
        required group partition_key {
            required $X_TYPE $X // for X in partition_key_columns
            ...
        }
        required group deletion_time {
            required int32 local_deletion_time
            required int64 marked_for_delete_at
        }
        required group shadowable_deletion_time {
            required int32 local_deletion_time
            required int64 marked_for_delete_at
        }
    }
    optional group static_row {
        required int32 flags (UINT8)
        required int32 extended_flags (UINT8)
        required group cells {
            optional group $X { // for X in static_columns
                required int32 flags (UINT8)
                optional int64 timestamp
                optional int32 local_deletion_time
                optional int32 marked_for_delete_at
                optional $X_TYPE value
            }
        }
    }
    repeated group rows {
        required group row {
            required int32 flags (UINT8)
            required int32 extended_flags (UINT8)
            required group clustering_key {
                optional $X_TYPE $X // for X in clustering_key_columns
                ...
            }
            required group regular {
                optional group $X { // for X in regular_columns
                    required int32 flags (UINT8)
                    optional int64 timestamp
                    optional int32 local_deletion_time
                    optional int32 marked_for_delete_at
                    optional $X_TYPE value
                }
                ...
            }
        }
    }
}
```

Bibliography

[1] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, “Dremel: Interactive analysis of web-scale datasets,” *Communications of the ACM*, vol. 54, pp. 114–123, 2011.

[2] Apache Software Foundation, “Apache parquet: File structure.” https://parquet.apache.org/documentation/latest/, 2020.

[3] Apache Software Foundation, “Apache parquet: Encodings.” https://github.com/apache/parquet-format/blob/master/Encodings.md, 2020.

[4] J. Goldstein, R. Ramakrishnan, and U. Shaft, “Compressing relations and indexes,” pp. 370 – 379, 03 1998.

[5] “Sstables 3.0 data file format.” `https://github.com/scylladb/scylla/wiki/SSTables-3.0-Data-File-Format`, 2020.

[6] J. Alakuijala and Z. Szabadka, “Brotli library.” `https://github.com/google/brotli`, 2020.

[7] Y. Collet, “Lz4 library.” `https://github.com/lz4/lz4`, 2020.

[8] Y. Collet, “Zstd library.” `https://github.com/facebook/zstd`, 2020.

[9] J.-l. Gailly and M. Adler, “Zlib library.” `https://github.com/madler/zlib`, 2020.

[10] J. Dean, S. Ghemawat, and S. H. Gunderson, “Snappy library.” `https://github.com/google/snappy`, 2020.

[11] Apache Software Foundation, “Apache arrow repository.” https://github.com/apache/arrow, 2020.

[12] Chojnowski, Michał and others, “parquet4seastar repository.” https://github.com/michoecho/parquet4seastar, 2020.

[13] ScyllaDB, “Seastar repository.” https://github.com/scylladb/seastar, 2020.