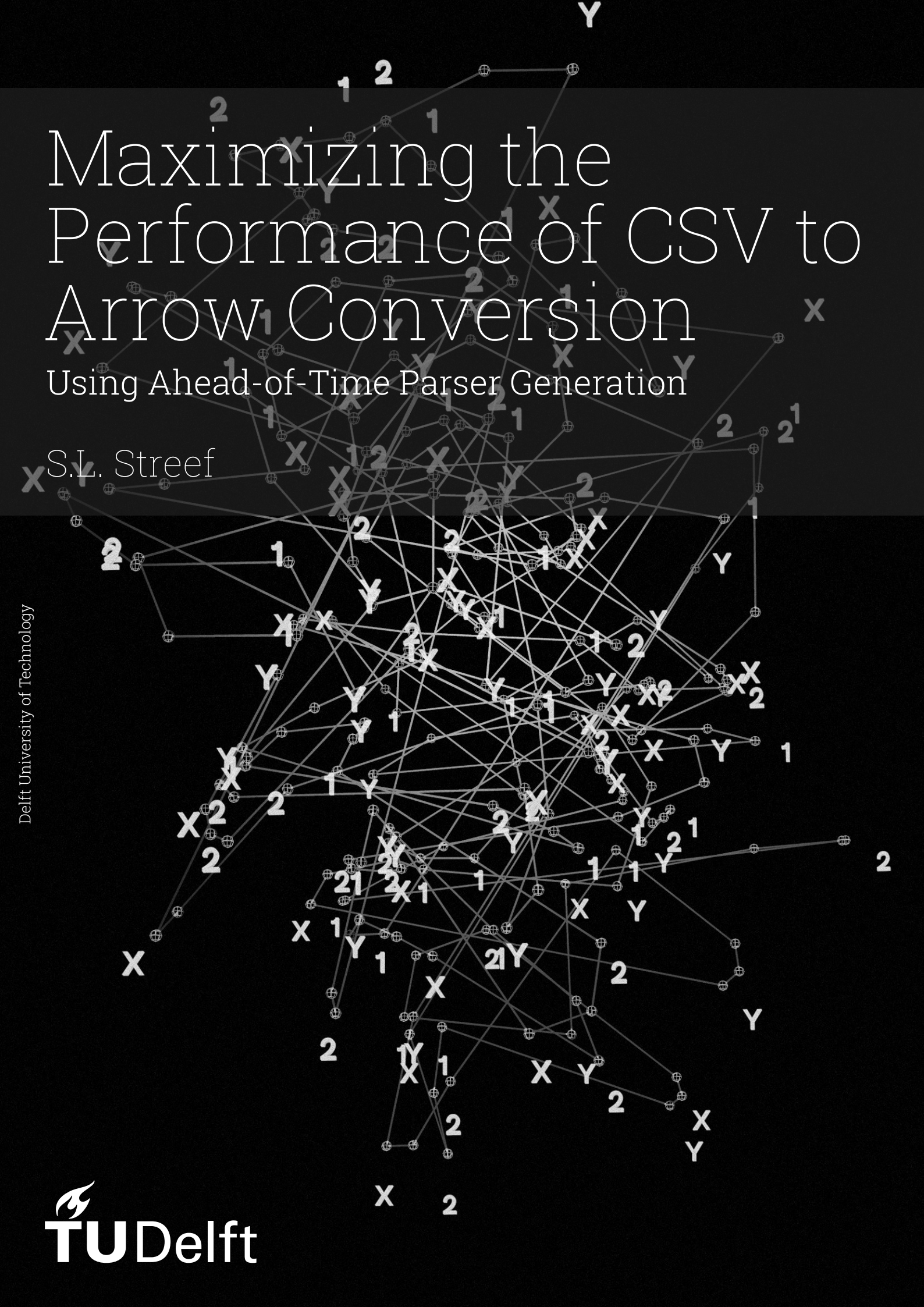
Maximizing the Performance of CSV to Arrow Conversion

Using Ahead-of-Time Parser Generation

S.L. Streef



Delft University of Technology

TU Delft

Maximizing the Performance of CSV to Arrow Conversion

Using Ahead-of-Time Parser Generation

by

S.L. Streef

Project duration: February 27, 2024 – January 30, 2026

Thesis committee:	Prof. dr. H.P. Hofstee	TU Delft
	Dr. ir. Z. Al-Ars	Trinilytics
	Dr. ir. K. Atasu	TU Delft
	Dr. ir. J. Peltenburg	Voltron Data

Cover: Resource Database on Unsplash

TU Delft

Abstract

This thesis explores the application of ahead-of-time parser generation to improve the throughput of big data ingestion. To investigate parser generation, this work produced several libraries related to the conversion of CSV to Apache Arrow. The code that this work has produced is available on GitHub[37].

First, an ahead-of-time parser generator was developed in the form of a Rust derive macro and a supporting library. Using a derive macro, a schema can be defined by a Rust struct definition for which a CSV to Arrow reader is derived. With the knowledge of the schema at compile-time, extra optimizations were possible. For instance, when the types in a schema are known to be of fixed size, estimates for the number of records in an input buffer could be made. With this principle, input bound checks could be reduced.

Experiments showed that ahead-of-time generated parsers outperformed state-of-the-art frameworks such as Apache Arrow[40], Polars[25], and DuckDB[5]. Benchmarks for single types revealed that for integers, unbuffered and buffered generated parsers achieved a throughput of at least 1.5x compared to Apache Arrow, with the reduction of size bound checks sometimes even reaching a throughput of 3x. For floating point numbers, the buffered parser performed slightly better than Arrow. However, the unbuffered, and buffered with reduced size bound checking, parser achieved a throughput of at least 1.5x. For strings, the buffered parser performed similar to Apache Arrow since it uses the same efficient buffered string parsing. The unbuffered parser only achieved half the performance. Furthermore, benchmarks for parsing the TPC-H[46] and TPC-DS[45] datasets showed that the buffered parser generated ahead-of-time performed better for real-world datasets compared to the frameworks mentioned above. The unbuffered parser was only able to achieve a higher throughput for datasets larger than 100 MB. For the datasets, the throughput varied based on the distribution of types.

Additionally, this work explored, but did not integrate, the use of multi-threaded CSV parsing. However, experiments revealed that the performance of parallelization depends on how fast CSV can be scanned for record positions whilst correctly checking character escaping. An experiment with a custom multi-threaded parser implementation showed that when scaling the number of parse threads, the throughput is limited by scanning rather than parsing. This characteristic was also found for Polars and DuckDB, which support multi-threading. The scanning was shown to be possibly improved by using SIMD[18], which allowed scanning record delimiters at 2.8 GB/s using AVX2. This is approximately 1.5x more than the maximum throughput that Polars or DuckDB achieved.

Contents

1 Introduction

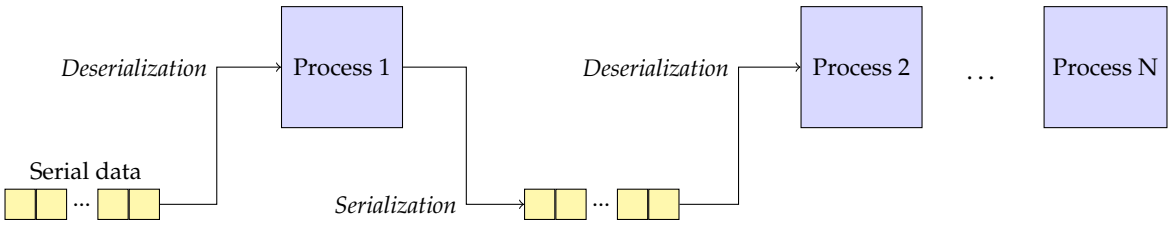
1.1. Context

Since the introduction of the internet, the amount of data generated and processed has been increasing steadily. This is compounded by recent developments with the increasing demand for AI services. In 2024, the ITU (International Telecommunication Union) [47] estimated that there would be approximately 6 ZB of data in fixed broadband traffic, which is a 20% increase in data traffic compared to the previous year. This represents data that needs to be processed at some point in time. Whilst the data traffic is growing rapidly, Moore's law seems to come to an end, making it more difficult to handle the throughput needed for processing this data. For this reason, it is essential to explore new areas in compute optimizations to keep up with demand.

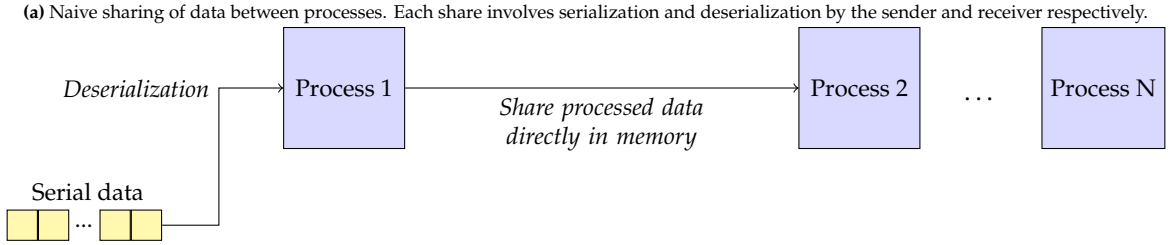
A critical part in processing data is serializing and deserializing data interchange formats, often received from, or sent to, network or storage devices. Such formats are useful for sequential data transfers or stores. For instance, JSON is often used for network communication between a client and a server. This format represents data textually, however, some values might need to be parsed because they do not represent text (i.e. a format such as JSON has to be parsed to a native representation for data to be correctly available). Conversely, native data might have to be converted to JSON. This is known as deserialization and serialization, respectively. It allows data to be converted from, and to, a sequential format such as JSON to, and from, a native data format accessible in memory. When a large amount of data needs to be processed, deserialization and serialization might account for a significant time in compute. Not only because it has to process all the data, but also because it happens every time the data is read. In fact, in the current age of big data analytics, the total data processing time is bottlenecked by reading the data rather than processing it [2, 4, 48, 23]. Consequently, reducing the number of (de)serializations, or increasing their throughput, could significantly help reduce compute costs.

A solution to reduce repeated (de)serializations is Apache Arrow [40], which is a standardized data format and eco system that allows for efficient processing of data. Its main goal is to eliminate the serialization and deserialization overhead that arises when multiple processes want to share their data. Traditionally, these processes, with their programs possibly written in different languages, would have to serialize and deserialize data if it is to be shared between two programs as seen in Figure 1.1a. Data would first have to be converted from an in-memory representation to a sequential data format so that it can be sent to a receiver. This receiver would then have to deserialize the data back to an in-memory representation. This serialization and deserialization would have to occur every time data is shared. By introducing the Apache Arrow format as a common interface for different programming languages, data can be simply shared in-memory. Figure 1.1b shows how Apache Arrow removes the need for serialization and deserialization beyond the initial serialization. Consequently, unnecessary processing is avoided, allowing for a higher overall processing throughput in a data pipeline. Moreover, the Apache Arrow format makes use of column-oriented data storage as opposed to row-oriented data storage (i.e. the data are stored contiguously in a memory buffer for each column). This unlocks high performance processing of fields, as operations on a column can be highly parallelized using SIMD.

The contiguous data can be directly loaded into SIMD registers or offloaded to a GPU, which makes it particularly useful for scanning or mapping records in large datasets. This would not be possible in a row-oriented storage format, because the fields are not stored contiguously. Additionally, Apache Arrow has libraries available in multiple programming languages that implement its format and convenient utilities. Because of this and its performance benefits, several data analytics frameworks use this format internally. Some popular frameworks include Pandas, DataFusion and Polars.



(a) Naive sharing of data between processes. Each share involves serialization and deserialization by the sender and receiver respectively.



(b) Sharing data using the Apache Arrow format. Data is deserialized once, data can then be passed in memory without (de)serialization.

Figure 1.1: The difference between naive data sharing (a) and using the Apache Arrow format (b).

Two of the most widely used formats for storing big data in the field are JSON and CSV. Despite their simple text based format and large size, they remain extensively used in practice for data communication, transport and storage.

JSON format

In their pursuit to accelerate deserializing JSON to Apache Arrow, Peltenburg et al. [24] found that a handwritten parser could outperform the standard Arrow implementation. The difference between their handwritten parser and the default parser was the grammar they accept. The default JSON parser has to accept the complete JSON grammar whilst the tailored handwritten parser only had to accept a grammar instance for its use-case (i.e. the handwritten parser is stricter and could reject valid JSON instances). Even though its strictness might be undesirable for general purpose processing, it could perform well for processing consistent data. Data for which their values can change but their schema, or structure, cannot. For example, Figure 1.2 shows two different JSON documents which both contain logs, each from a different system. Figure 1.2a and 1.2b show the measured temperature and relative humidity values respectively. Both of these documents are structurally equivalent[1] but have different values. Consequently, writing a parser that accepts one will also accept the other. For such applications, there is no need for a general purpose parser that recognizes the complete grammar, which is something a handwritten parser could do, thereby making it potentially faster.

Whilst writing a parser by hand might improve performance, it introduces extra development costs. Whenever the schema of a document changes, a developer has to update their parser to accept this schema, which adds to development time but also adds to the risk of introducing bugs. Hence, it would be more convenient to have a program or framework that helps generate the parser with minimal effort and low risk. Several parsing frameworks exist that provide building blocks for defining a custom parser (see Section 2.2). Nevertheless, they do not provide a way to store the parsed values directly into the Apache Arrow format. As a result, a developer would still need to adapt the parser.

Peltenburg et al. [24] showed that a custom parser performs well for JSON [3]. JSON, or JavaScript object notation, is a text-based, human-readable and object-oriented format. It is commonly used in IoT and web applications as a communication protocol between different programs. The format can be difficult

[1] Note that in Figure 1.2 the number of elements in the array does not change its structure or schema.

```json
[
    {
        "time": "2025-01-03 17:45",
        "value": 20.1
    },
    {
        "time": "2025-01-03 18:00",
        "value": 19.95
    }
]
```

(a) A JSON log containing temperature values.

```json
[
    {
        "time": "2025-03-14 11:58",
        "value": 55.1
    },
    {
        "time": "2025-03-14 12:03",
        "value": 60.2
    }
]
```

(b) A JSON log containing relative humidity values.

Figure 1.2: Two different JSON documents each containing different values but both adhering to the same data schema or structure.

to parse because of its grammar, which allows for dynamic types such as nested objects and arrays, which require more sophisticated checks when parsing, such as matching the delimiters. Additionally, they make the grammar that a parser (generator) needs to support significantly larger.

CSV format

CSV [32] is a text-based, human-readable and tabular format. Its intended use case was sharing tabular data between spreadsheet programs. However, because of its simplicity it is also commonly used as a data storage format. For the European data repository [6], 19% of the datasets are CSV and for the American government repository [1] this is 8%. The data in a CSV [32] document is formatted as a table, it has rows and columns. Each row describes a record and each column describes a field of such a record. Columns, or fields, are separated by a comma. Rows, or records, are separated by a newline character. However, different delimiters could be used in practice, since the RFC-4180 [32] is not a single fixed standard. For instance, in some regions of the world, a semicolon character is used for field delimiters rather than a comma. Moreover, any value in CSV is stored as plain text, often as UTF-8 or UTF-16 characters. The value can be optionally enclosed by double quotes, allowing for characters to be escaped that are otherwise illegal. For example, storing a comma or newline character when these are field and record delimiter respectively. Furthermore, CSV allows the use of a double quote in a value, given that the value is enclosed by double quotes and that the double quote is escaped. A double quote can be escaped by preceding it with another double quote. An example, representing the same data in JSON from Figure 1.2, can be represented in CSV as seen in Figure 1.3. The figure shows that the data can have a header which is defined in the top-most row. Furthermore, it shows that values can be quoted, which does not have to change their representation or schema. Furthermore, it is important to note that the values in a CSV document assume no type. The format only considers textual values, which have to be interpreted by the reader (i.e. parser) of the document.

```csv
time,value
2025-01-03 17:45,20.1
2025-01-03 18:00,19.95
```

(a) A CSV log containing temperature values.

```csv
time,value
"2025-03-14 11:58","55.1"
"2025-03-14 12:03","60.2"
```

(b) A CSV log containing relative humidity values.

Figure 1.3: Two different CSV documents each containing different values but both adhering to the same data schema or structure.

In practice many variations of the CSV format according to the RFC-4180 [32] exist. This can vary from different delimiter characters and support for a header or comments. To focus on finding the potential of a parser generator, and not trying to support every variation, it is important to limit the scope of the CSV format. For this reason, the generated parser should support the general rules of the RFC-4180 with several assumptions. First, the field and record delimiters are assumed to be the comma and newline character respectively. Second, the character set will be limited to UTF-8 rather than UTF-16. Third, no comments are allowed. Finally, headers might be supported, but they would not provide any benefit in terms of performance. These parameters, or assumptions, are well supported by major frameworks that

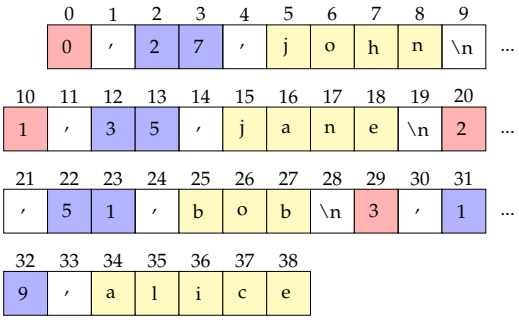
parse CSV (e.g. Pandas, Arrow or Polars).

1.2. Challenges

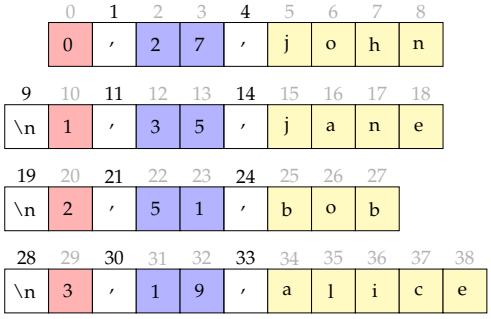
There are several challenges that need to be resolved in order to reduce deserialization overhead by means of a parser generator. First, there is the main challenge of transforming textual CSV data to the Apache Arrow format. Second, is the code generation that needs to adhere to certain standards or rules for it to be viable. Finally, there is the challenge of parallelizing CSV parsing which, when solved, could provide a solution to deserialization overhead. The following sections elaborate on each of these challenges.

1.2.1. Transforming CSV to Arrow

For CSV to be transformed or parsed into the Apache Arrow format, it has to follow a set of steps. This process can be split into three distinct steps, for which each step imposes their own challenge. The first step is parsing the CSV, for which data is purely text-based. For a human, this data may seem structured but for a machine it is a sequence of bytes (Figure 1.4a). Consequently, the CSV data has to be parsed, to identify the field and record positions (Figure 1.4b). The process in Figure 1.4 shows the CSV characters as a sequence. In order to parse this sequence, we need to find a structure for the data. The result allows data from the original text blob to be split into segments that represent fields or records.



(a) A CSV document when read into memory, interpreted as an arbitrary sequence of characters.



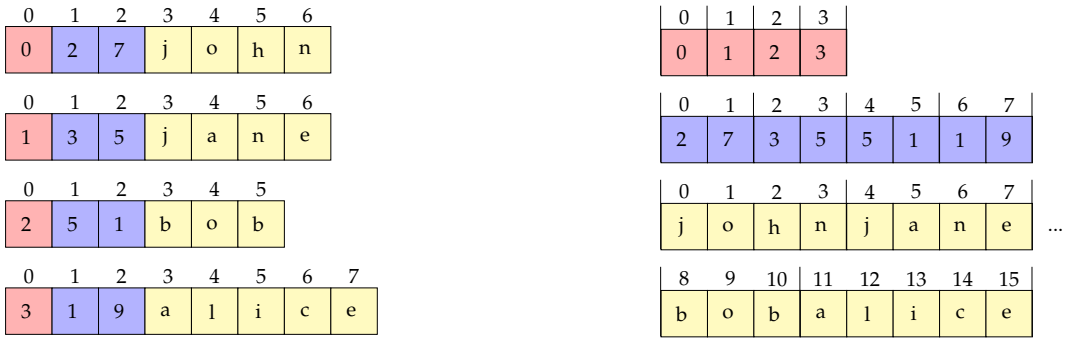
(b) A CSV document read into memory, then parsed to identify the end of fields and records (indicated by the highlighted index).

Figure 1.4: The process of parsing raw CSV data (a) to identify field and record positions (b). Essentially providing a structure for the previously unstructured sequence of bytes.

When CSV data is structured, it has to be restructured to fit the Apache Arrow format. CSV is naturally a row-oriented data format since it stores the values of a record as a row (i.e. the columns of a record instance are always stored together). Figure 1.5a shows this row-oriented storage, where the different colors indicate the different fields or columns. Conversely, the Apache Arrow format is a column-oriented data format. The value of a column for each record is stored in a buffer dedicated for that column. Figure 1.5b displays column-oriented storage, where the different colors also indicate the different fields or columns. Converting between the two storage orientations in Figure 1.5 means that the data must move significantly. Nevertheless, column- and row-oriented data storage can be mapped to each other.

Another step required to convert from CSV to Arrow is parsing the textual values to native values. For example, a number is textually represented, often in decimals. Consequently, for a program to perform calculations, this number has to be parsed and stored as a native number rather than a string. A challenge in parsing these values is not knowing their type. This is solved by either providing a schema to the parser or by inferring the schema through an analysis.

Finally, it is worth noting that the conversion of data orientation and the parsing of the text values can be swapped. The order of these steps is a subject to design choice and each order has its own advantages, and disadvantages, and is discussed later in this work.



(a) Row-oriented storage; each column in a record is stored sequentially. Each color indicates a unique field.

(b) Column-oriented storage; each column per record is stored in a dedicated column buffer. Each color indicates a unique field.

Figure 1.5: Two different orientations for storing records, row-oriented (a) and column-oriented (b) storage. The sequential numbers mean sequential in memory.

1.2.2. Generating parsers

A parser generator needs to construct a program that takes CSV data as input and produces an Apache Arrow data structure as output. It should replace the role of a developer that would write a parser by hand. Consequently, a developer has little insight into how the generated code came to be. This moves the risk of introducing bugs into a parser to the parser generator. For this reason, the design of tools used for a parser generator should try to minimize or reduce this risk.

1.2.3. Parallel parsing

The textual characteristic of CSV makes parallel parsing difficult. The standard approach to parallelize processing is by dividing the work into multiple pieces and run them in parallel. Consequently, a naive solution to parsing CSV would be to split the CSV document into arbitrarily sized chunks and start parsing these chunks in parallel. The result of each chunk could then be accumulated into a single Apache Arrow record batch. Unfortunately, it is not that trivial for CSV. As mentioned above, CSV allows for escaping a value, which may then contain delimiters or other special characters. This escaped value needs to be enclosed by double quotes. This characteristic makes parallelization using a simple divide and conquer approach impossible. The reason for this is as follows.

Consider that each chunk of CSV data follows some other chunk except for the first chunk. Whilst parsing each chunk, a state is needed to track its progress. First, it needs to maintain whether it is in a field or not. This is to correctly validate a delimiter such as a field or record delimiter. Second, it needs to maintain whether the current field is escaped. This is to assert if special characters are allowed to be present in the current value. Finally, in case of an escaped field, the presence of a preceding double quote needs to be stored, to validate a correct ending of an escaped field, or to validate an escaped double quote.

For the first CSV chunk, an initial state is used but any subsequent chunks require the resulting state of the previous chunk. Alternatively, if every chunk would start with a blank state, there could be instances in which they start in the middle of an escaped field. This is because a CSV row/record is not a fixed size and splitting CSV into arbitrary sized chunks could therefore result in chunks starting at random positions in a record. Without knowing that a chunk starts in an escaped field, it will result in undefined behavior when parsing. For instance, the ending double quote of the escaped field is encountered, for which it detects the start of an escaped field at that position. Likewise, an escaped double quote could be encountered, resulting in either an empty string if it is followed by a delimiter or an error if it is not. Both examples are only two of many possible variations in which parsing random positions without any context results in undefined behavior.

A naive solution to the problem of not knowing the previous state is to skip input characters until a record delimiter is encountered. A record delimiter indicates that the following character is the first field of a record. However, remember that an escaped value is allowed to contain record or field delimiters. Consequently, when starting to parse in an escaped field, a record delimiter could be encountered that does not represent the end of a record. This would result in the same problem as above.

1.3. Research questions

The goal of this research is to evaluate the performance of CSV to Arrow parsers generated ahead-of-time. In order to achieve this goal, a parser generator framework will be developed. For which, several of the previously mentioned challenges should be solved. This work will try to answer the following research questions:

1. What challenges are there in parsing CSV or CSV to Arrow?
2. How does the performance of a generated parser compare to other parsers?
3. What unique optimizations can a parser generator leverage?
4. What are the pitfalls of a generated parser?

These questions will be answered by two methods. First, by the process of implementing the parser generator. Second, by running the generated parser against different types of datasets.

1.4. Outline

This paper tries to answer the research questions step-by-step, doing so in the following order. First, Chapter 2 will describe the background for parsing and parser generation. This includes state-of-the-art CSV parsing techniques, popular frameworks for constructing parsers and popular data analytics framework. In doing so, concepts can be taken for inspiration and comparison in future chapters. Chapter 3 will then discuss the different solutions for the parser generator and the generated parser. Furthermore, Chapter 4 will describe the implementation of the solution for a CSV to Arrow parser generator. Chapter 5 will describe the setup and experiments carried out on the state-of-the-art CSV parsers and the generated parsers. Additionally, it will provide results for other possible optimizations such as using SIMD. Finally, Chapter 6 will answer the research questions, summarize the work and propose future work.

2

Background

2.1. State-of-the-art CSV parsers

To generate a performant CSV to Arrow parser, it is important to explore state-of-the-art CSV parsers. However, there are few innovative parser implementations due to the simplicity of the CSV format. Exploring the implementations of CSV parsers inside data analytics frameworks (from Section 2.3) revealed that most parsers are simple code defined state machines (i.e. CSV is parsed by simply looking for the delimiters). Whilst this is a valid approach, it is neither state-of-the-art nor complex. Consequently, this work will not explore these parsers because they are trivial.

2.1.1. Finite automata

A finite automaton is a computational model that is useful for recognizing patterns [33]. It is a model that takes a string as input and either accepts or rejects it. This functionality is what makes a finite automaton applicable for parsing, since it can decide if an input adheres to a format. But before applying the finite automaton model, it is important to understand its definition and workings. A finite automaton is formally defined by a 5-tuple as seen in Equation 2.1. Each of the elements in this tuple is explained in Table 2.1.

$$M = (Q, \Sigma, \delta, q_0, F) \tag{2.1}$$

Symbol	Definition
Q	The finite set of all possible states.
Σ	The finite set of all possible input characters, also known as the alphabet.
δ	The transition function, formally defined as $\delta : Q \times \Sigma \rightarrow Q$. This function describes how the automaton can transition from one state to another, given an input character from the alphabet.
q_0	The initial state, or start state. Formally defined as $q_0 \in Q$, meaning that the start state is in the set of all possible states.
F	The set of final states, or accept states. Formally defined as $F \subseteq Q$, meaning that there can be multiple final states from the set of all possible states.

Table 2.1: The definitions of each item in the formal definition of a finite automaton (see Equation 2.1).

To illustrate how a finite automaton works and how it is formally defined, two visual examples are provided in Figure 2.1. Consider the example in Figure 2.1a. It depicts a finite automaton with a binary string input that only accepts strings containing an odd number of ones (e.g. it will accept the string *1101* but not the string *0110*). It can be formally described using Equation 2.1. Observing Figure 2.1a, the set of all possible states contains three states $\{q_0, q_1, q_2\}$. Of these states, q_0 is marked as the initial state and $q2$ as the only final state, therefore $q_0 = q_0$ and $F = \{q_2\}$. Moreover, the input type is only binary, resulting in an alphabet of $0, 1$. Finally, the transition function can be expressed by a transition table

as seen in Table 2.2a. The left most column indicates the current state and the top most row the input character. Each of these combinations describes the next state, which represents the formal transition function $\delta : Q \times \Sigma \rightarrow Q$. In summary, the finite automaton that recognizes binary strings containing an odd number of ones can be formally defined by Equation 2.2.[1]

$$M_{odd\_ones} = \{\{q_0, q_1, q_2\}, \delta_{table2.2a}, \{0, 1\}, q_0, \{q_2\}\} \tag{2.2}$$

The same process can be applied to the other, more simple, finite automaton that recognizes even strings (Figure 2.1b). Using its transition table (Table 2.2b), it can be formally defined by Equation 2.3.

$$M_{even\_string} = \{\{q_0, q_1\}, \delta_{table2.2b}, \{0, 1\}, q_0, \{q_0\}\} \tag{2.3}$$

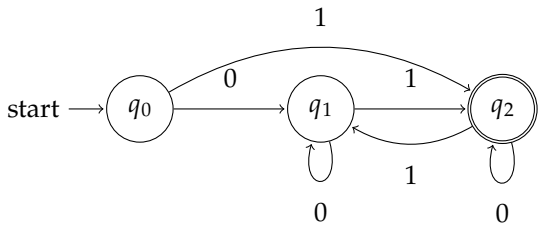
	0	1
q_0	q_1	q_2
q_1	q_1	q_2
q_2	q_2	q_1

(a) The transitions for the finite automaton defined in Figure 2.1a.

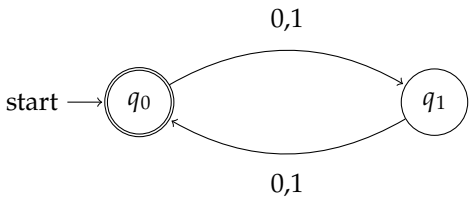
	0	1
q_0	q_1	q_1
q_1	q_0	q_0

(b) The transitions for the finite automaton defined in Figure 2.1b.

Table 2.2: The transition function expressed as a transition table for each of the finite automata defined in Figure 2.1.



(a) A visual representation of a finite automaton that only accepts binary strings containing an odd number of 1s.



(b) A visual representation of a finite automaton that only accepts binary strings of even length.

Figure 2.1: Two visual examples of finite automata recognizing a pattern.

The automata given in the examples are known as deterministic finite automata (DFAs). They are finite because they have a finite set of states. Furthermore, they are deterministic because each transition uniquely maps two states given an input symbol. Alternatively, a non-deterministic finite automaton (NFA) exists that has a set of finite states but non-deterministic transitions. An NFA has the unique property of allowing transitions, or edges, between states that do not consume an input symbol. These are known as epsilon transitions. The result of this property is that when an NFA is evaluated, an epsilon edge will introduce multiple possible states at the same time. This is what its non-determinism refers to. Because an NFA can contain epsilon edges, its 5-tuple remains the same except for the definition of its transition function. Its transition function becomes $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$ where $\mathcal{P}(Q)$ is the power set of the possible states. This power set models the fact that with epsilon transitions, all the combination of states could be reached rather than a single state.

NFAs and DFAs can recognize the same languages, however one might be preferable over the other in different applications. For example, an NFA is easier to construct than a DFA, as it can reduce the number of states and transitions. Alternatively, a DFA can be useful for hardware implementations, as its transitions can easily be encoded as a table. An NFA can be converted to a DFA by using the powerset construction.

Parsing CSV

A CSV parser that makes use of a finite automaton is the `rust-csv` [8] crate. The general idea of this CSV parser is to construct and use a DFA that recognizes the CSV format. In his work, Gallant [8] notes

[1] Note that δ_{table} can be found in Table 2.2a.

that an NFA is easier to construct. For this reason, the CSV parser is first constructed as an NFA after which it is transformed into a DFA. This transformation is possible because an NFA is equivalent to a DFA [33]. I.e. they recognize the same language. As this transformation is a trivial process, only the DFA will be discussed. The DFA constructed by this library can be described using the formal definition of a finite automaton (see Equation 2.1), where the possible states ($|Q| = 10$) are defined as follows.

1. Start of a record
2. Start of a field
3. In a field
4. In a quoted field
5. In an escaped quote
6. In a double escaped quote
7. In a comment
8. End of a field
9. End of a record
10. End of a line

The transition function of the DFA is implemented by a transition table. This allows parsing to be simple, since each state transition becomes a look up. Each transition is the combination of the states and the alphabet of the DFA. A naive approach would be to consider the alphabet of this parser as the range of a byte (i.e. values from 0 to 255 or formally $\Sigma_{byte} = \{0, 1, \ldots, 255\}$ where $|\Sigma_{byte}| = 256$). However, Gallant [8] notes the combination of this alphabet and the states would lead to a large transition table. Specifically, it would lead to $|Q \times \Sigma_{byte}| = 2560$ transitions, translating to a lookup table of 2560 bytes in size. Fitting this in cache for modern CPUs might not be a problem. However, for older or embedded CPUs this number can still be relatively large. Consequently, Gallant [8] decided to map the byte values to classes in order to reduce the number of transitions.

In CSV grammar, not all bytes share the same meaning. For instance, there is no distinction between numeric characters such as 0 to 9, they are just values. However, there is a distinction between a comma and a numeric character. More specifically, the first indicates the start or end of a field. For this reason, characters can be divided into classes to indicate their meaning. This means that some characters can share a class and are equivalent in their meaning. This classification is nothing more than a mapping between the alphabet Σ_{byte} and a new alphabet of classes Σ_{class}. The mapping can be described by a transition function $\delta_0 : \Sigma_{byte} \rightarrow \Sigma_{class}$ where $|\Sigma_{class}| = 7$ and the classes are defined as follows.

1. The field delimiter
2. The first record terminator
3. The second record terminator (if the record terminator is CRLF, then CR and LF are distinct equivalence classes)
4. The quote byte
5. The escape byte
6. The comment byte
7. Everything else

The result of this extra transition is an extra lookup into a 256 byte transition table and a lookup into a $|Q \times \Sigma_1| = 70$ byte transition table. Additionally, a third table of 70 bytes is needed to determine whether an input byte is to be stored or not. All of these mechanisms are constructed at run-time when this parser is called.

Since this research focuses on developing a parser generator, it is worth noting that `rust-csv` can be considered as one. More specifically, the DFA that it uses to parse CSV is constructed every time a user uses this parser. It is constructed at run-time, based on configurations passed to the parser. An example of such a configuration is the delimiters used for fields or records. This allows the parser to generate a different class and transition table. Nevertheless, for this research it does not qualify as a complete

parser generator. First, it only parses the CSV values to a native Rust representation and not to the Arrow format. Second, it is only able to parse CSV to string records, where all values are strings (i.e. this DFA is only able to parse the syntax of CSV and not its field types).

2.1.2. SIMD

Langdale and Lemire [18] showed that it is possible to parse a format such as JSON efficiently using SIMD. One of the challenges they solve in parsing textual formats such as JSON is parsing delimiters. More specifically, the challenge discussed in Section 1.2.3 allows for escaped values that are delimited by double quotes. These values may contain special characters such as delimiters, and thus double quotes. For this reason, division of data into chunks and parsing them individually could lead to errors. For instance, when the parsing of a chunk starts in the middle of an escaped value, an escaped delimiter might be interpreted as non-escaped leading to incorrect parsing.

Langdale and Lemire [18] solve this problem for JSON by using SIMD to generate bitmaps that indicate when a character is escaped or not. A character is escaped if it is delimited by a preceding and a following double quote. Between each delimiter and character can be many characters, as well as escaped characters such as the double quote. In JSON, a double quote is escaped by using the backslash delimiter. They classify the problem of determining escaped characters by means of propagation. Assume a sequence of arbitrary characters. The first double quote character that is encountered will make all succeeding characters be escaped until a second double quote is encountered. That is, a second double quote that is not escaped by an escape character. Thus, the first double quote character starts a propagation and every character propagates the escape except for a second unescaped double quote character.

For JSON, a double quote needs to be escaped by a back slash, which adds complexity to propagation. However, for CSV the escape delimiter for a double quote is defined to be a double quote. This makes escape propagation easier, as having an escaped double quote in an escaped value means that the first double quote ends propagation and the second double quote starts propagation [19]. The final result is that all characters that are not double quotes can still be detected as escaped using propagation. To illustrate this propagation for CSV, consider the examples in Figure 2.2. Before propagation, a bitmap of the double quotes is needed. Then a running xor needs to be performed on this bitmap, for this example the input bit is considered zero (i.e. on every bit an xor operation is performed with the result of the previous bit). The result of the running xor is a bitmap with escaped characters. As seen in Figure 2.2b this also works for an escaped value containing an escaped double quote.

```
12,"Joe  Doe",56    CHARACTERS
___1________1___    BITMAP
___111111111____    RUNNING XOR
```

(a) Finding the bitmap for escaped characters using a quote bitmap and XOR propagation.

```
12,"Joe""Doe",56    CHARACTERS
___1___11___1___    BITMAP
___1111_1111____    RUNNING XOR
```

(b) Finding the bitmap for escaped characters using a quote bitmap and XOR propagation where an escaped double quote is present.

Figure 2.2: Two examples for finding the bitmap for escaped characters using a quote bitmap and XOR propagation.

Langdale and Lemire [18] discovered that this binary propagation by means of a running xor could be performed using a special AVX instruction. This instruction is the clmul, or carry-less multiplication, instruction. Using this instruction with the text characters as first argument and supplying a bit mask with all bits set to one as the second argument, it performs a running xor. A system running on a modern architecture with support for AVX512 could compute four 128-bit bitmaps at the time using the `_mm512_clmulepi64_epi128` instruction [14]. This allows computing bitmaps for 512 characters per operation. However, an escape might be propagated on the edge of a bitmap, which needs to be taken as input by the next bitmap. Consequently, a bit has to be propagated to subsequent bitmaps. Fortunately, this is a simple operation as either a 0 or a 1 might be propagated. A 0 will not change the result and a 1 will simply negate the whole bitmap. For this reason, it is a case of negating a bitmap based on whether the last bit of the previous bitmap is 1. Figure 2.3 shows the two possibilities of propagation between bitmaps. Figure 2.3a shows a case where, without propagation, invalid escape bitmaps are generated.

Although the algorithm to detect escaped characters in CSV might not parse every aspect of CSV, it does provide useful context. Using the bitmaps of the escaped characters, it is easy to navigate through the

```
aaa,"bbb bbb",aaa    CHARACTERS
____1___ ___1____    BITMAP
____1111 ___11111    RUNNING XOR
____1111 111_____    BIT PROPAGATION
```

(a) Propagation of one from the last bit in the first 8-bit bitmap to the second 8-bit bitmap.

```
aaa,"bb" ,aaaaaaa    CHARACTERS
____1__1 ________    BITMAP
____111_ ________    RUNNING XOR
____111_ ________    BIT PROPAGATION
```

(b) Propagation of zero from the last bit in the first 8-bit bitmap to the second 8-bit bitmap.

Figure 2.3: Two examples for finding two 8-bit bitmaps for escaped characters using propagation. After each computed 8-bit bitmap, the last bit of the first bitmap is propagated to the second bitmap.

CSV text and detect the positions of records and fields. Consequently, it can be used to speed up and parallelize CSV parsing.

2.2. Parser generator frameworks

A parser generator framework provides abstractions for creating a parser. Common parser generators allow users to call functions, sometimes macros, that perform certain parsing steps. For instance, a user might call a function that parses a delimiter and throws an error if the input does not match. Even though these frameworks significantly reduce the boilerplate code needed when writing a parser, they still require significant effort from a developer.

A popular parser generator framework in C++ is the Boost Spirit X3 library. Boost spirit X3 [10] is a recursive-descent parser constructed by defining a grammar in C++. It is one of the frameworks used by Peltenburg et al. [24] to compare parse performance with. Boost spirit allows a user to define a grammar at compile-time by using inline grammar specifications. From this grammar, a parser is made available that allows for parsing the grammar. Moreover, a hierarchical data structure for the parsed data can be constructed and managed.

An alternative framework for generating parsers in Rust is the Nom crate. Nom [21] is a parser combinator framework used to construct recursive descent parsers. A parser combinator framework is a framework for constructing a parser by providing and combining a set of parsing functions. This allows for modular construction of a parser where individual functions can be tested and repeated.

Both frameworks operate with the concept of compile-time available context. That is, a user can define their parser with some sort of macro or inlined functions. Using this approach, a compiler has access to the code that will run at run-time. Other parsing frameworks only decide their parsing functions at run-time usually based on run-time schema inference. Consequently, parser generator frameworks can optimize more parser code ahead-of-time. Nevertheless, both frameworks still require a user to manually specify their parse instructions. Consequently, when a grammar has changed, their parser has to be manually updated.

2.3. Popular data analytics frameworks

Data analytics frameworks allow users to ingest and process data efficiently. To perform efficient operations, these frameworks often rely on the Apache Arrow format or some derivative of their concept. In addition, they often provide support for many different data sources such as CSV. Consequently, most data analytics frameworks have CSV to Arrow conversion built in. For this reason, they can be used to compare with a parser generator. Additionally, their implementation details might provide ideas for improving or solving challenges in converting CSV to Arrow.

Many data analytics frameworks exist, for which some of the most popular, state-of-the-art and open-source frameworks are Polars [25] and DuckDB [5]. Additionally, Apache Arrow [40] supports performing data analysis through their libraries. For this research, their underlying implementations for CSV parsing is particularly interesting.

2.3.1. Apache Arrow

In addition to its format, the Apache Arrow [42] framework provides libraries to interact with its format. Most notably, it provides functionality for reading different data sources into the Apache Arrow format.

Because these libraries also define the format and how it is stored, Apache Arrow has the advantage of having access to the internal implementation. For instance, its implementation has more direct access to buffers for storing data such as strings. For this reason, reading CSV to Arrow should achieve the highest performance using this framework. Consequently, it will serve as the baseline for performance in this project.

To convert a CSV data source into Arrow record batches, the Rust Arrow CSV reader performs several steps. First, the underlying `rust-csv` [8] parser (see Section 2.1.1) is used to produce string records. The string fields of these records are stored in dedicated buffers for each field. Second, when the target number of records for a record batch is reached, all the string fields are parsed to their respective data types. Finally, a record batch is produced from the field buffers.

When converting from string records to records with types, Arrow makes use of several different type parsers. Knowing the parser implementations allows for setting a fair comparison between a parser generator and Apache Arrow. Using the same parsers in the parser generator allows the CSV parsing performance difference to be measured more accurately. Table 2.3 shows the parsers for some of the most used types. Since Arrow parses CSV directly to string records, no additional parsing is needed for strings. For integers it makes use of the atoi [16] crate, which provides safe parsing functions to protect from cases such as numbers that do not fit in the number type. For floating-point numbers it makes use of the lexical core [22] crate, similar to atoi it provides safe parsing functions.

Category	Type	Variants	Implementation
Number	Unsigned integer	8-, 16-, 32-, 64-bit	atoi [16]
	Signed integer	8-, 16-, 32-, 64-bit	atoi [16]
	Floating-point	32-, 64-bit	lexical core [22]
	String	escaped, unescaped	N.A.

Table 2.3: The parsers used by Arrow for some of the most common data types.

As described in Section 1.2.3, parallelizing CSV parsing can be difficult due to its characteristics. At the time of writing, the Rust Apache Arrow implementation does not provide a multi-threaded solution.

2.3.2. Polars

Polars [25] is a state-of-the-art data analytics framework written in Rust, designed for fast data processing on a single machine. It allows many different data sources to be ingested and stored as a dataframe including CSV. A dataframe is similar to an Apache Arrow record batch as it makes use of the Apache Arrow format for storage. Consequently, it can be converted to Apache Arrow data structures with little to no overhead. Furthermore, Polars achieves high performance because of its query engine. Its query engine optimizes query plans for more efficient computations.

To convert CSV to dataframes, Polars uses a different approach to Apache Arrow. Polars defines a multi-threaded CSV parsing model in their `polars-io` crate for which it requires different steps than Apache Arrow. First, the data is split into chunks. For chunks they try to minimize the size of a chunk to 16 MB to still fit in L3 cache. However, when a file contains many columns the number of chunks is adjusted such that the data is split in less chunks. This is to reduce the buffer allocation overhead per chunk. The data is split into chunks by parsing the CSV line delimiters to have an exact number of records per chunk. Second, a thread is spawned from a thread pool for each chunk. For each chunk, each line is split into fields by parsing column delimiters. Each field value is then parsed to the type of a field and stored in its respective buffer. Finally, the buffers can be used to construct dataframes.

Polars makes use of different type parsers, for which the parsers for the most common types can be found in Table 2.4 When converting from string records to records with types, Arrow makes use of several different type parsers. For integers it makes use of the atoi-simd [31] crate, which is similar to atoi, but provides SIMD support if the target machine supports it. For floating-point numbers it makes use of the fast_float2 [11] crate, which is a port of a state-of-the-art C++ number parser developed by Lemire [20]. Furthermore, for strings, possibly escaped characters are removed using a built-in string parser.

Category	Type	Variants	Implementation
Number	Unsigned integer	8-, 16-, 32-, 64-bit	atoi-simd [31]
	Signed integer	8-, 16-, 32-, 64-bit	atoi-simd [31]
	Floating-point	32-, 64-bit	fast_float2 [11]
String		escaped, unescaped	built-in

Table 2.4: The parsers used by Polars for some of the most common data types.

2.3.3. DuckDB

DuckDB [5] is a state-of-the-art database management system (DMBS) that supports SQL. DuckDB tries to achieve higher performance by providing a columnar-vectorized query execution engine. Compared to traditional database systems such as MySQL, processing occurs on vectors of column values rather than processing row by row. As such, it uses a similar model to Apache Arrow.

Whilst DuckDB is able to convert CSV to Apache Arrow record batches, its open-source code is extensive and abstract due to its query engine. Consequently, it is hard to pin point the model of CSV to Arrow conversion and the parsers used for the types. Nevertheless, DuckDB can still act as a good comparison in performance benchmarks.

3

Alternative solutions

The implementation for CSV to Arrow conversion can be split in two distinct problems. The first problem is the generation of parser code. Second, is the problem of parsing CSV, which translates to the implementation details of the generated parser. For both problems, different solutions or design choices exist that need to be considered. The following sections will explain and compare the alternative solutions for these problems.

3.1. Programming language

Both the generator and the resulting parser are programs that are defined using a programming language. In general, a programming language provides abstractions to interact with the hardware of a machine. Languages are often categorized for this property in terms of a level. Higher-level programming languages introduce more abstractions as opposed to lower-level languages. An example of one of the lowest level language is assembly. A programmer has to write a significant amount of boiler plate code to achieve certain operations. For instance, printing in assembly requires several assembly instructions, but in a higher level language such as Java only a single print statement is required. Whilst this makes a higher-level language seem more useful as it reduces complexity, there are benefits to lower level languages. For example, a lower level language allows for fine grain control over hardware. Micro optimizations can be applied by the developer rather than relying on a compiler. An optimization could be to use a specialized instruction but also using a different algorithm. In higher level languages, the implementation details are often defined by the programming language or compiler rather than the user.

The most popular low-level programming languages used for performance applications are C and C++. They provide low-level control with little overhead introduced by abstractions. C++ is an extension to C, where additional features were added . Moreover, it is used as the language for the default Apache Arrow implementation. In more recent years Rust has been added to the popular high performance languages. Just like C++ it offers high performance with features such as generics, iterators and lambda expressions. They both offer these abstractions with the zero-cost principle, these features will only cost what is used. For instance, if an iterator over a list first limits the list to half the size and then performs a map operation, only half of the items are mapped.

Additionally, Rust adds the promise of additional memory safety. Both C and C++ are known to be vulnerable to memory bugs introduced by programmers. For example, having a dangling-pointer, a buffer overflow or a use-after-free. Although C++ supports smart pointers that can automatically manage pointers, it is up to individual developers to enforce. Alternatively, Rust tries to reduce memory bugs by enforcing a set of rules at compile-time. These rules limit the use of a value, such that the problems previously mentioned cannot occur. If one of the rules is violated, a Rust program will not compile and will not introduce a memory bug. Alternatively, C and C++ would compile and possibly introduce undefined run-time behavior.

To be able to use Apache Arrow, it provides a C data interface which allows any language to use its

format over a foreign function interface (FFI) [43]. However, this requires a program in a non-C language to adjust their code so that their data adheres to the C FFI convention. More specifically, any data going in, or out, over this FFI has to be transformed from, or to, a C data representation. In turn, this introduces conversion overhead and adds complexity when using the Apache Arrow format. Fortunately, Apache Arrow also provides an interface in the form of a library for several popular programming languages [41]. This allows for an easy Apache Arrow integration for C, C++, and Rust.

3.1.1. Generator

The target programming language of a generator is important as it defines its usability and complexity. Usability is important for any users or developers that would use a parser and the complexity needs to be considered for the maintenance of the parser generator.

First, integrating a parser generator into a project should not be hard. Integration can happen in different ways, depending on the language. The most common way of integrating code is by defining and using a library. At compile time the library is compiled and linked to the program of a user. Another option is to generate code using macros. C++ and Rust allow the use of macros and thus code generation directly in the user program. An advantage of using macros is that the generated parser is available in the user program. This can allow for better optimizations when compiling or doing link-time optimizations. These optimizations include inlining.

Second, the implementation of a parser generator should not be difficult to maintain or debug. This means that code generation should not be too hard nor very complex. The macro ecosystem in Rust contains several libraries that allow for easy code manipulation. Additionally, procedural macros are implemented as crates, allowing the code to be checked by the Rust compiler. This supplies useful warnings and errors during the development of a parser generator. C++ or C on the other hand, do not provide much information when compiling other than syntactic errors.

Rust fits well as the target language of the generated parser on both criteria. It provides a compiler with useful messages and a well supported ecosystem for defining macros. Additionally, its macros allow for easy integration in a user program. Consequently, Rust is the choice of target language for the parser generated.

3.1.2. Parser

The target programming language of a parser is important for several reasons. First, it needs to perform well once compiled, since the goal of the parser is to parse CSV as fast as possible to Arrow. Second, it needs to be usable, so that it should be easy to integrate into another program. Finally, it should be safe to generate, allowing bugs in code generation to be detected.

Generating code is an automated process that might reduce bugs because it removes the need for a human to (re)write a parser. However, it can also introduce systematic bugs that a user might not catch, since the user does not have fine-level control or understanding of the code. This risk could be reduced by extensive testing and even software verification but it would be desirable to have a target language that could provide additional safety out-of-the-box. For instance, a lower-level language such as C or C++ could be considered unsafe because it does not provide a safety mechanism in the case of out-of-bound errors. The responsibility lies with the developer, or code generator, to ensure such a bug is not introduced. Failing to do so can lead to undefined behavior and possibly malicious exploitation. Alternatively, a language such as Rust can provide memory safety, as it may warn a user with a compiler error in the case of such memory bugs. Consequently, when Rust code is generated, it might not compile if some illegal memory behavior is produced by the generator.

Another important criterion is the performance of the generated parser. Different metrics can be used to define the performance of a program. For example, the throughput it can achieve with regard to its input. Alternatively, the memory characteristics of the program throughout execution. In general, lower level programming languages perform best for both these metrics. They allow for more fine-grained optimizations, such as using low-level instructions that better fit a problem. In addition, they are compiled ahead-of-time, allowing for little runtime overhead. The most popular low-level languages, C and C++, are therefore often considered for high performance applications. In more recent years, Rust is also considered, as it has matured as a low-level language with higher-level functionality. Other

language alternatives are interpreted or just-in-time (JIT) compiled languages. Sometimes the two are even combined, allowing for source code to be compiled to byte code and the interpreted into machine code at run time.

Another important criterion is the usability for parsing and integration with Apache Arrow. Apache Arrow provides a C data interface which allows any language to use its format over a foreign function interface (FFI) [43]. However, this requires a program in a non-C language to adjust their code so that their data adheres to the C FFI convention. More specifically, any data going in, or out, over this FFI has to be transformed from, or to, a C data representation. In turn, this introduces conversion overhead and adds complexity when using the Apache Arrow format. Fortunately, Apache Arrow also provides an interface in the form of a library for several popular programming languages [41]. This allows for an easy Apache Arrow integration for languages such as Rust, Java, Python and others.

Combining the criteria, Rust fits well as the target language of the generated parser. It can provide the performance of a low-level language, whilst safeguarding users for possibly generated memory bugs. Additionally, it is well supported by Apache Arrow for integration and use of its format. Consequently, Rust is the choice of target language for the generated parser.

3.2. Parser generation

Whilst a parser will execute at run-time, its generation could be either at run- or compile-time. Both provide advantages and disadvantages in terms of performance, complexity, flexibility and usability of the parser generator.

3.2.1. Ahead-of-time

Ahead-of-time, or static, parser generation refers to generating a parser at compile-time. It allows a parser to be compiled and statically linked to a program. A parser is constructed by generating the code for a program that takes CSV as input and generates Arrow record batches as output. The advantage of constructing it at compile time is two-fold. First, the program can be compiled, which can optimize choices made by the parser constructor. Additionally, compiler flags can be added to tweak the performance of the program even more. For instance, the compiler could parallelize certain code into SIMD instructions if the target computer architecture supports it. Second, the user, or programmer, utilizing the parser generator can supply context to the generator. For instance, the user supplies it with the schema of the CSV records. Consequently, the parser generator is tailored to only accept a grammar instance of CSV, rather than the complete CSV grammar. This allows for replacing generic parsing of CSV to specific parsing a set of types. I.e. every field in a CSV record is not parsed the same, but rather parsed depending on the type. However, this makes the generated parser inflexible at run-time, allowing for only the specified schema to be parsed[1]. Nevertheless, for a data pipeline, flexibility is often not needed since the files are often repetitions of a schema.

A compile-time parser generator can work much like a library, in that it provides a parser implementation to the user. More specifically, a parser can be generated as if generated by the user. In Rust, a method for generating code is using macros. The Rust ecosystem has three types of procedural macros that allow for manipulation of Rust code at compile-time. For instance, the procedural derive macro allows for automatically deriving a trait implementation (i.e. code is generated in the form of an implementation for a trait). A common use is automatic derivation of the Debug trait, which allows a data type such as a struct to be formatted for debug printing. In addition, a derive macro provides the ability of supplying helper attributes or arguments. In general this is useful when a user of a macro library would like to customize an implementation. This means that a derive macro has the potential for customizations of the parser generated by a parser generator derive macro.

In Rust, for a procedural derive macro to generate a parser it needs a structure for which it can implement such a parser. Several structures or data types are allowed to be derived, namely structs, enums or unions [27]. A struct is a type that can accurately represent tabular data, making it suitable for representing the schema of a CSV record. For instance, the columns for each row in a CSV file can be represented by a struct as shown in Listing 3.1. Each field defines the name and the type of a column. Using an unnamed struct definition, field names are omitted which indicates the absence of a header in a CSV

[1] If input data does not match the schema, the parser will simply throw an error.

file. Defining a struct means defining a schema for CSV, but also defining concrete types for the string values. Consequently, it also provides a good representation, or schema, for an Apache Arrow record.

```
id,name,birth_year
0,Bob,1992
1,Alice,1974
2,John,2001
3,Jane,1953
```

(a) A CSV file

```
struct Named {
    id: u64,
    name: String,
    birth_year: u16
}
```

(b) A struct with named fields

```
struct Unnamed (
    u64,
    String,
    u16
);
```

(c) A struct with unnamed fields

Figure 3.1: Two different Rust struct representations for the record schema of a CSV file. The struct fields represent the columns of a CSV file, one struct represents a single row in a CSV file.

The type of field can be any Rust type, allowing for extensive type support with custom struct types and enums. However, field naming is more limited since structs cannot contain both unnamed and named fields[2]. It is possible to exclusively use unnamed fields as seen in Figure 3.1c. Furthermore, both CSV and Apache Arrow formats, allow types to be null (i.e. the value can be undefined). Describing a nullable type is different in Rust compared to other languages such as C, since Rust uses a type abstraction. Any generic type `T` that is nullable is described by an enum `Option<T>`, where `None` represents a null value and `Some(T)` represents a value. Consequently, it provides a descriptive and intuitive definition for nullable fields in schemas for CSV.

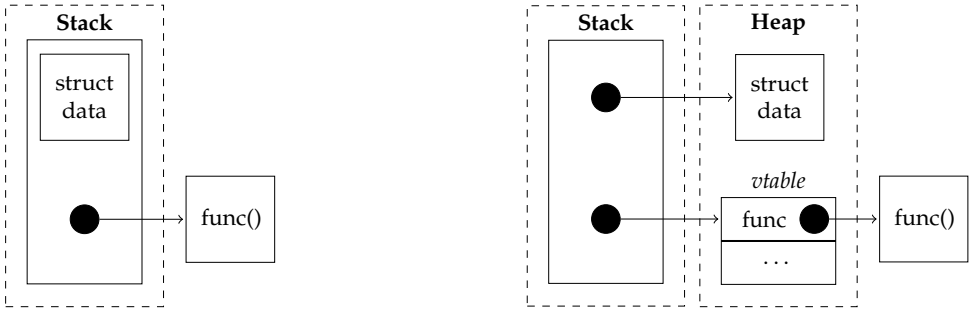
3.2.2. Just-in-time

Just-in-time (JIT), or dynamic, parser generation refers to generating a parser on demand at run-time. This approach is also called just-in-time (JIT) compilation. Two solutions exist for generating a parser at run-time. Either a parser is provided to a user by code generation or by run-time interpretation.

First, parser code can be generated in the form of a Rust program and loaded in at run-time. In Rust, a program can be compiled with the `--crate-type=dylib` argument to generate a dynamic library as output [26]. The result is a shared object (`.so`) file that can be dynamically linked to a program. Dynamic linking allows a program or library to be linked at run-time rather than compile-time. This allows a library to be shared by multiple programs, saving space and making updates more easy. Alternatively, it can be used to introduce new logic into a running program. For instance, the *libloading* [15] crate in Rust allows dynamic libraries to be loaded in at run-time. The functions in this library can then be invoked. Consequently, this would allow for a dynamic parser library to be generated and compiled at run-time. This could then be loaded in using the *libloading* crate and invoked by the user.

A second solution to run-time parser generation is in-memory parser construction using a data structure. The data structure can hold the information required to parse and store fields of a record. For instance, by defining a struct data type containing a buffer, or builder, for each supported type. Furthermore, a parser trait can be defined with a function that takes bytes as input and returns the remaining bytes after parsing whilst storing new values in the buffer. For each data type struct, this trait can then be implemented and optimized per type. Next, instances of the data type structs should be stored when the parser generator is invoked for a schema. When different types are present, Rust does not allow these different structs to be stored together in a vector or array because they have different sizes. Consequently, to store them together, the structs need to be transformed to dynamic trait objects (e.g. `&dyn Trait`) [29, 28]. This transformation results in trait objects that have the same memory layout and size, allowing them to be stored together in a vector. The difference in memory layout between a struct and its resulting dynamic trait object can be seen in Figure 3.2. Converting to a trait object adds extra indirection and requires the struct data to be moved to the heap. Additionally, trait method implementations need to be looked up by using dynamic dispatch. Dynamic dispatch allows a program to determine the implementation at run-time by using a virtual table (vtable). A vtable holds pointers to the method implementations for a type and a vtable is constructed for each type that implements a trait.

[2]It is possible to add field names using macro attributes, allowing for a mix of named and unnamed fields. However, this implementation is left open, as it is only a quality of life feature.



(a) The memory layout for the struct.

(b) The memory layout for the struct stored as a dynamic trait object.

Figure 3.2: Two examples of the memory representation of a struct, which implements a trait, when stored normally (a) and as a dynamic trait object (b). The dynamic trait object representation adds extra indirection by adding data to the heap and requires resolving method implementations at run-time rather than compile-time.

3.2.3. Ahead-of-time vs. just-in-time

In terms of performance, compile-time code generation allows for applying compiler optimizations to the code and its integration into a user program. Modern compilers are good at optimizing programs and could, for instance, vectorize parts of code. These optimizations become easier to be applied when code is more simple. That is, in the case of tailoring parsers to a schema, this less generic code can be easier to optimize. Additionally, compile-time schema knowledge can be used to apply parsing strategies. Both compile- and run-time code generation solutions could benefit from this. However, optimizations come at the cost of an increased compile time, which for run-time means an increased run-time cost. For run-time code generation code needs to be generated, compiled, and linked at run-time, introducing a significant overhead. It is possible to mitigate some of this cost by caching the shared object files when a parser is requested multiple times for the same schema. However, this will add complexity to the parser generation. Alternatively, generating a parser at run-time in memory using dynamic dispatch could provide less overhead as no storage, compilation and linking happens.

The complexity of a solution is important for maintaining and debugging the parser generator. A more complex parser generator becomes harder to debug and might be more difficult to ensure properties such as safety or functionality. For the compile-time solution, constructing and maintaining a Rust macro crate is easy. The macro ecosystem is well documented and multiple crates exist that allow for easy manipulation or generation of Rust code. Moreover, a macro is executed at compile-time, allowing to display errors or info messages for debugging. Alternatively, a run-time solution code generation solution requires code generation without the help of the macro ecosystem. This makes it more difficult to generate correct code and requires more manual labor. Finally, a run-time in-memory solution would provide the least complex solution as it involves only writing a library.

The flexibility of a parser generator defines how it can be easily used to generate different parsers. For instance, the ahead-of-time parser generator does not allow parsing a different schema during run-time (i.e. only parsers generated at compile-time are available for use). Defining multiple schemas and thus generating multiple parsers is possible however. Alternatively, the just-in-time parser generator solutions do allow for generating and using a parser at run-time. Nevertheless, as mentioned in 1.1, in practice a schema is often fixed at run-time. For instance, in a system that processes logs, which can contain different values but it schema does not change. Consequently, ahead-of-time parser generation might not have to be flexible.

A parser generator has to be used by a developer, as such some solutions might be easier to use. Ahead-of-time parser generation by providing a macro crate is easy to use, as it uses a common approach for deriving functionality. Derive macros are often used by Rust developers to derive functionality automatically. For this reason, most users are acquainted with them. Alternatively the just-in-time parser generator solutions can be implemented as a crate which provides a reading function much like Arrow.

Ultimately, the performance of a parser generator is expected to be best with static generation and dynamic in-memory generation. Dynamic code generation can be complex, as it involves extra steps

that need to occur at run-time. With the overhead that compilation and linking provides, it should not be able to achieve higher performance than static code generation. It only provides the advantage of having a more flexible parser generator, as it can generate parsers at run-time. However, this can also be achieved by the just-in-time in-memory parser or might not be needed for some applications. The remaining solutions both provide a relatively easy usability and implementation complexity. For this reason, both approaches are explored.

3.3. Generated parser

In addition to the process of parser generation, different solutions exist for the parser itself. The most significant difference in solution for the parser is its strategy in reading the data source (i.e. whether to buffer the input data or not).

Buffered vs unbuffered

A parser, or reader, parses bytes from an input source. Such sources include bytes, files and streams. In some cases, it might be desirable to read a data source in increments. For instance, when reading a stream, bytes might arrive at different times. Consequently, bytes can be buffered to accumulate and process once the buffer is full. Moreover, when reading a file, the file may be sufficiently large that reading it completely into memory is inefficient. For this reason, the file might be better suited to be read in smaller chunks. For example, by repeatedly storing a chunk in a buffer and reading it.

A buffered reading approach can provide better performance in terms of memory for larger data sizes. Additionally, it is required for some data sources to be read correctly. Nevertheless, it adds some complexity to reading or parsing. Since bytes are read in increments, parsing might have to be stalled because some bytes are not available yet. Consequently, state has to be managed with buffered parsing. Alternatively, unbuffered parsing assumes that the complete data source is available. Therefore, it cannot be stalled and does not have to maintain state.

Unbuffered reading allows for a significantly less complex parser implementation, since no state has to be managed and no edge cases need to be handled. However, some data sources simply need buffered reading, as their data is not always immediately available. The Apache Arrow CSV reader seems to support both types, however, the underlying implementation of the unbuffered reader uses the buffered reader. Therefore, it only makes use of a buffered reader. For this reason, the main solution for a CSV reader will be a buffered implementation, so that it can be compared correctly against the Apache Arrow implementation in terms of performance. Nevertheless, since unbuffered reading introduces control overhead, it can provide a good upper bound to what the performance can be. Consequently, unbuffered reading is also implemented and compared.

4

Implementation

4.1. Ahead-of-time parser generation

The solution for ahead-of-time parser generation is to create a procedural derive macro that generates a parser. As described in Section 3.2.1, a procedural derive macro is often used to generate code that implements some functionality. There are two requirements to being able to generate code using a procedural derive macro. First, a data structure is needed for which the procedural macro can be invoked. Second, a trait for which to derive is required.

Before defining a trait for which a procedural macro can derive, it is important to outline the setup required. A procedural derive macro is a type of library that can only export its derive implementation by default. Consequently, to also provide a trait for which it can be invoked, an additional library should be defined. Hence, the ahead-of-time parser generator consists of two modules, namely the `arrow_csv_reader` [34] and `arrow_csv_reader_derive` [36] modules. The first contains traits and implementations needed for deriving a CSV to Arrow reader. The second module is the procedural derive macro that generates the parsing code.

4.1.1. Parser derivation

To generate a parser, a derive macro needs a trait to implement and a data structure to implement the trait for. As described in Section 3.2.1, a struct can act as a schema for a CSV row and an Apache Arrow record. To generate the code for a parser using a derive macro, the `ArrowCsvRecord` trait is defined in the `arrow_csv_reader` crate as seen in Figure 4.1a. Deriving the `ArrowCsvRecord` trait for a struct means that this struct will be interpreted as the Arrow and CSV schema/record to generate code. The functionality of the initial trait is limited. It provides users with a static definition of the Arrow schema and is used to implement other traits that define the actual CSV to Arrow reader functionality. The derive macro can then be invoked using the `#[derive(ArrowCsvRecord)]` attribute as seen in Figure 4.1b.

```rust
// arrow_csv_reader/lib.rs
use arrow::datatypes::Schema;

pub trait ArrowCsvRecord {
    fn schema() -> Arc<Schema>;
}
```

(a) The initial trait used to derive a CSV parser.

```rust
use arrow_csv_reader::ArrowCsvRecord;
use arrow_csv_reader_derive::ArrowCsvRecord;

#[derive(ArrowCsvRecord)]
struct Example(u8, i16, String);
```

(b) An example of deriving the `ArrowCsvRecord` trait on an arbitrary struct that acts as a schema. The struct will have access to the `schema` function at compile-time.

Figure 4.1: The initial trait (a) for which the parser implementation should be derived (b).

This definition assumes that a schema is known ahead-of-time. However, sometimes the schema, or data types of values, of a CSV file might not be known by a user. For this reason, popular frameworks

allow for inferring a schema by scanning part of a CSV file before parsing it. This inference occurs at run-time, inducing an extra overhead. Since in this section the parser generation occurs ahead-of-time, the only way for inference to work is by providing an example file at compile-time. Whilst this would be possible, it would have no difference from providing the schema using the discussed struct definition. Therefore, schema inference is left open, as it would only be a quality of life improvement. Nevertheless, for a fair comparison in results, every framework is provided with a schema at compile-time. For the discussed frameworks, these schema can be passed to the parser.

Simplifying schema field definitions

In theory, there is no limit to the number of columns that a CSV file can have. Fortunately, this is also true for the number of fields a struct can have in Rust. For real-world datasets it is not unusual to have a large number of fields. For instance, the 'web sales' dataset from the TPC-H [46] benchmark contains 34 fields. Defining the schema for such datasets using a struct can introduce many lines of code. To aid in reducing code complexity, a feature for repeating fields through a helper attribute was implemented. Using the repeat helper attribute as seen in Figure 4.2 a field is repeated multiple times, reducing the size of a schema definition. This attribute can only be applied to unnamed structs, because repeating a named field would create multiple fields with the same identifier or name.

```rust
#[derive(ArrowCsvRecord)]
struct Example (
    String,
    String,
    String,
    String
);
```

(a) Traditional field repetition

```rust
#[derive(ArrowCsvRecord)]
struct Example (
    #[repeat(4)]
    String
);
```

(b) Field repetition using the helper attribute.

Figure 4.2: The default method for repeating fields (a) and using the repeat helper method (b).

4.1.2. Generating a reader

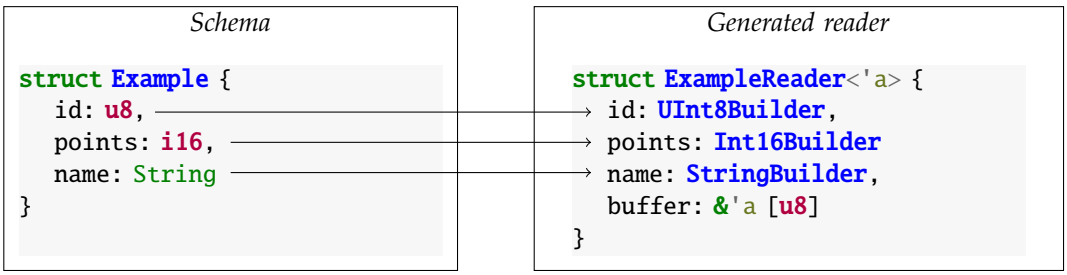
The derivation of the `ArrowCsvRecord` trait allows the derive macro to modify the target struct or generate additional code. Any procedural macro starts with a token stream as input, for which it can add, remove or change tokens to manipulate Rust code. A token stream is a sequence of token trees that represents the Rust syntax. Using a library such as syn [44], a token stream can be transformed to a Rust code syntax tree. This allows the input token stream to be inspected and manipulated with Rust data structures. In the case of a struct, the fields can be inspected to retrieve and store their name and type. This information provides the basis for generating a CSV to Arrow reader.

The `ArrowCsvRecord` trait contains a static method for constructing an Apache Arrow schema. This method is implemented by the derive macro after inspecting the type and name of each field. An Apache Arrow schema consists of a vector of fields with each field requiring three properties. First, the name of a field that can be directly taken from the struct field identifier. Alternatively, in the case of a tuple struct without field identifiers, a name can be generated from the index of the field. Second, the type of the field, which has to be mapped from a Rust type to an Apache Arrow type. Fortunately, both Rust and Apache Arrow support similar types, allowing for simple mapping. The final property is the nullability of a field, which can also be extracted from the type of a field. More specifically, the composite `Option<T>` type in Rust indicates that a type can be null. Using these properties, the code for constructing an Arrow schema can be inlined and implemented for the static schema method.

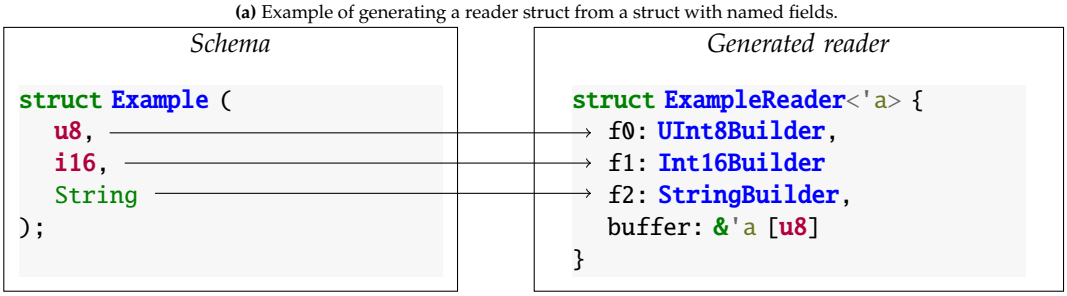
Struct generation

After the initial `ArrowCsvRecord` trait is derived, a reader struct must be generated. This is because a trait can only be implemented for a data type. Additionally, state needs to be managed whilst parsing. For example, parsed values need to be stored in a buffer and the input data should be managed so that the input data can be traversed incrementally. Therefore, when the `ArrowCsvRecord` trait is derived, a new struct definition is generated using its schema. The new reader struct consists of two key components. First, a field is defined to manage the input data. This field stores the slice, which

is a reference to bytes. Second, for each field in the schema, a field is added to the reader that holds an Arrow array builder. This builder is provided by Apache Arrow and provides an abstraction for storing values in a contiguous buffer. Two examples of how a reader struct can be generated from an input schema struct are found in Figure 4.3. The figures show how each schema field is mapped to a builder field in the reader struct. In the case of a tuple struct that has no field names, the field names are generated according to their field index.



(a) Example of generating a reader struct from a struct with named fields.



(b) Example of generating a reader struct from a tuple struct by generating field names.

Figure 4.3: Two examples of generating the code of a reader struct from an input schema struct. Each type is mapped to a specific Arrow builder. In the case of a tuple struct (b), the field names are automatically generated by using the field index.

Reader functionality

With a reader struct present, the derive macro can add reader functionality by implementing two additional traits. First, the `CsvReader` trait is defined as seen in Figure 4.4a, which provides a method to build a CSV to Arrow reader from bytes. This reader is required to have implemented the `ArrowCsvReader` trait, which is the second defined trait. As seen in Figure 4.4b, this trait has no methods but enforces that the `From` and `Iterator` trait are implemented for types that implement the trait. Consequently, a reader constructed by the `CsvReader` trait will always have the `From` and `Iterator` trait implemented. These traits provide the construction and reading functionality required for parsing CSV to Arrow. Furthermore, the `Iterator` trait will be used to read record batches, exactly like the Apache Arrow implementation. An overview of the functionality and use of traits for ahead-of-time parser generation can be found in Table 4.1.

```
pub trait CsvReader: ArrowCsvRecord {
    fn reader<'a>(bytes: &'a [u8]) -> impl ArrowCsvReader<'a>;
}
```

(a) The trait for constructing an instance of an unbuffered reader from input bytes. This trait is implemented on an `ArrowCsvRecord` implementation. I.e. it is implemented for the schema struct.

```
pub trait ArrowCsvReader<'a>:
    Iterator<Item = Result<RecordBatch, ArrowError>> + From<&'a [u8]>
{
}
```

(b) The trait that defines the functionality of an unbuffered reader. That is, it must implement a record batch iterator and be able to be constructed from bytes.

Figure 4.4: The traits used to define the functionality of an unbuffered CSV reader.

Trait	Functionality	Description
`ArrowCsvRecord`	A type can be used as an Arrow schema.	The initial trait for which the derive macro needs to be invoked.
`CsvReader`	A type can construct a CSV to Arrow reader.	The trait defining the construction of a CSV to Arrow reader.
`ArrowCsvReader`	A type can be used as a CSV to Arrow reader.	The trait defining the traits required to be implemented for parsing CSV to Arrow.
`From<T>`	A type can be converted from a generic type T.	Trait required for constructing a reader.
`Iterator<T>`	A type can be iterated over whilst producing a generic type T.	Trait required for progressing a reader. I.e. parse the CSV to Arrow.

Table 4.1: The traits used to implement a CSV to Arrow parser using a derive macro.

The `From<&'a [u8]>` trait defines that a reader can be statically constructed from a byte slice. The byte slice should be a reference to the bytes of a CSV file that needs to be read[1]. The implementation for this trait is generated like the struct definition. For each field, the field name and the Arrow builder are used to initialize the respective fields. Additionally, bytes are taken from the input argument of the function and stored in the buffer field. Figure 4.5 shows an example of generating the implementation for the `From` trait. Using this implementation, the `CsvReader` trait can be implemented for the schema struct. This implementation can now simply call the static `from` method from the generated reader struct.

Schema

```
struct Example (
   u8,
   i16,
   String
);
```

Generated implementation

```
impl<'a> From<&'a [u8]> for ExampleReader<'a> {
   fn from(value: &[u8]) {
      Self {
         f0: UInt8Builder::default(),
         f1: Int16Builder::default(),
         f2: StringBuilder::default(),
         buffer: value
      }
   }
}
```

Figure 4.5: An example of generating the code for the `From` trait implementation from a tuple struct. This implementation follows the same mapping technique as the struct generation from Figure 4.3.

The `Iterator` trait allows a reader to be used as an iterator with an Apache Arrow record batch as the result item. That is, for each iteration, the reader could produce a record batch. The Apache Arrow implementation for parsing CSV implements the same type of iterator. Consequently, a generated reader will have a similar user interface as the Apache Arrow reader. The implementation of an iterator consists of one method called `next`, which optionally yields an item. When no item is returned, the iterator is exhausted. The generated implementation for the `Iterator` trait can be divided into several steps.

1. Yield nothing if input is empty.
2. Parse records whilst input is not empty, or the maximum number of records per record batch is not yet reached.
3. Convert field buffers to arrays.
4. Construct a record batch from schema and field arrays.

Following the steps, the first step is a simple size check on the input buffer. The second step is the parse step. It consists of a loop that checks if the input bytes are non-empty and the number of records parsed

[1]Since the byte slice is a reference, it is accompanied by a lifetime. Lifetimes in Rust are used to check if references are valid and if a reference does not outlive the data it points to. As such, the lifetime `'a` will be passed from the byte slice to the reader since it will store the reference. Consequently, the reader must have a lifetime that does not outlive `'a`.

is not exceeding the maximum record batch size. By default, the maximum number of records per record batch is set to 1024 records, just like the Apache Arrow default. Each successful loop iteration corresponds to one record parsed. Since the schema of the record is known, the parsing code for each type can be inlined. The idea is that by doing this, the compiler has access to all parsing instructions. This allows the compiler to optimize every part of the complete record parsing step. This is different compared to the approach such as that of Arrow, where the actual parsing from text to type is known only at run-time. The parsing segment of each field either stores a value to the buffer and advances the input buffer, or halts the reader by returning an error. The third step builds arrays from the field builders, which are needed to construct a record batch. In this step, the calls to the finish method of the builders are inlined into a vector. Finally, a record batch is attempted to be constructed using the static schema definition from the `ArrowCsvRecord` trait and the vector of field value arrays. The control flow diagram for this process can be found in Figure 4.6.

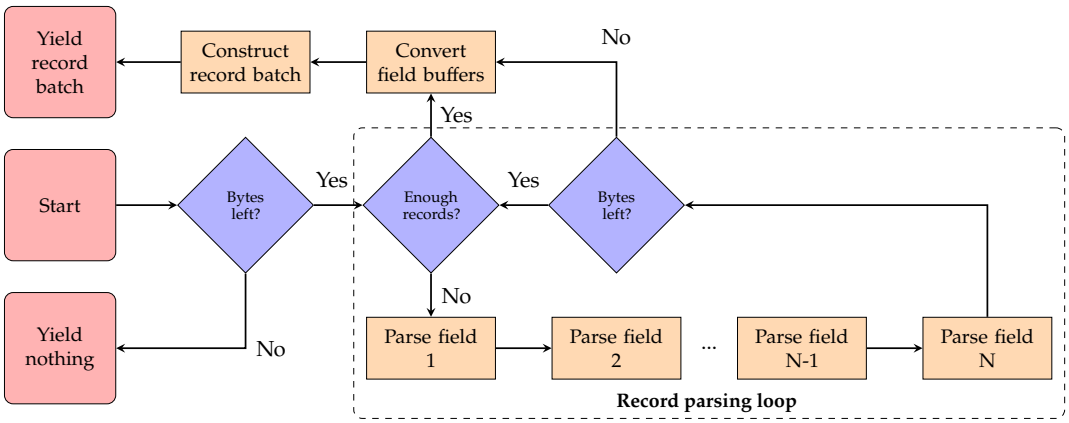


Figure 4.6: The control flow diagram of the `next` method implementation for an unbuffered CSV to Arrow reader. The highlighted area shows the record parsing loop where the type parsers for each field are inlined after each other (see Figure 4.8.

4.1.3. Parsing types

Knowing the type of a field by the schema struct definition, the generated parser requires a parser for this type. A simple solution to this is to use the same parsers as Arrow (see Section 2.2). For integer types, Arrow uses the atoi crate to safely parse a number from bytes. Equivalently, for floating-point numbers the lexical core crate is used. The parsers used can be found in Table 4.2.

Category	Type	Variants	Implementation
Number	Unsigned integer	8-, 16-, 32-, 64-bit	atoi [16]
	Signed integer	8-, 16-, 32-, 64-bit	atoi [16]
	Floating-point	32-, 64-bit	lexical core [22]
String		escaped, unescaped	custom (rust-csv [8] based)

Table 4.2: The parser for each type used in generating unbuffered parsers.

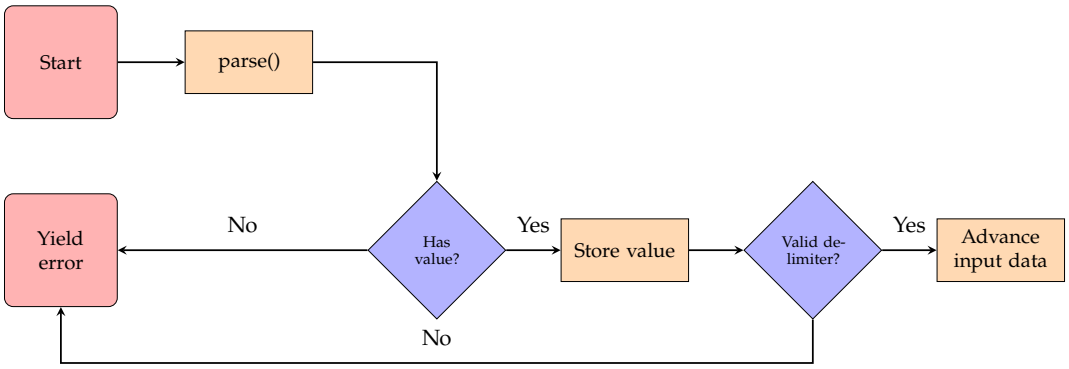
Since some of the parsers are implemented by different libraries, their input and output can be different. When trying to use these parsers it can be inconvenient to have to deal with different results when generating the complete CSV to Arrow reader. For this reason, the `Parser` trait is introduced as seen in Figure 4.7. Any type that implements this trait can be parsed from bytes, returning a tuple of two elements as a result. The resulting tuple consists of a possible value and the number of bytes that were consumed when parsing. For the types in Table 4.2 with parsers from external libraries, the trait can be implemented by calling the parser and mapping their result. Additionally, the trait can be directly implemented for custom parsers. For example, it is implemented for the string type using a custom version of the rust-csv [8] approach discussed in Section 2.1.1. Its implementation can be found in the `arrow_csv_reader` library.

Using the `Parser` trait, inlining the parsing code of each field into the record parsing loop becomes easy. For each field the `parse` method is first inserted, after which control logic needs to be added

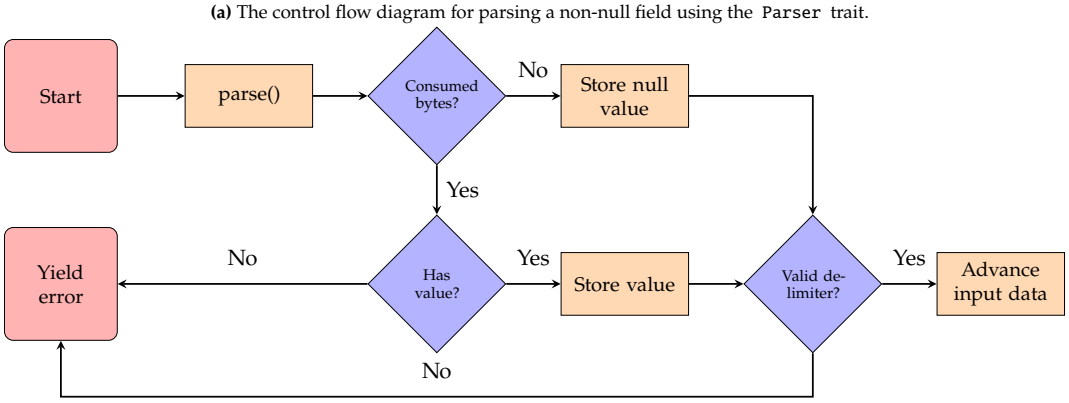
```rust
pub trait Parser: Sized {
    fn parse(bytes: &[u8]) -> (Option<Self>, usize);
}
```

Figure 4.7: The `parser` trait used to let all type parsers have the same input and output.

to handle the result. Depending on whether a field allows for null values, different control logic is generated. For non-null fields the control logic is simple as seen in Figure 4.8a. If a value is returned by the parse method, store the value and check if the next character is a valid delimiter. When this is the case, the input data can be advanced by the number of bytes that were consumed. In any other case, an error is thrown, stopping the CSV reader. Alternatively, a field can be null for which an extra check needs to be added to the control logic as seen in Figure 4.8b. The extra check is to identify if a parser could not parse any bytes. In this case, the value should be null.



(a) The control flow diagram for parsing a non-null field using the `Parser` trait.



(b) The control flow diagram for parsing a possibly null field using the `Parser` trait.

Figure 4.8: Two control flow diagrams for parsing a non-null or possibly null field respectively.

4.1.4. Generating a buffered reader

The generation of a buffered reader follows similar steps to an unbuffered reader, however, several changes have to be made. First, the struct definition of a buffered reader has to be updated in order to support interrupting and resuming parsing when input bytes are temporarily depleted. Second, traits similar to the `CsvReader` trait and `ArrowCsvReader` trait, the `CsvBufReader` and `ArrowCsvBufReader` traits respectively, are added for buffered reading as seen in Figure 4.9 and Table 4.3.

Struct generation

A buffered reader struct is generated similar to the unbuffered reader struct but with different fields and identifier. An example of a buffered reader struct generated from a schema is found in Figure 4.10. To allow sources to be read in increments, the input source type is changed to a type that implements the `Read` trait. This trait defines the functionality for bytes to be retrieved incrementally from a source. As seen in Figure 4.10, this type is stored wrapped in a `BufReader` which provides helpful methods to

Trait	Functionality	Description
`CsvBufReader`	A type can construct a buffered CSV to Arrow reader.	The trait defining the construction of a buffered CSV to Arrow reader.
`ArrowCsvBufReader`	A type can be used as a buffered CSV to Arrow reader.	The trait defining the traits required to be implemented for buffered CSV to Arrow parsing.
`BufferedTypeBuilder`	A type can be used to parse and store values to construct Apache Arrow arrays.	The trait defining the buffered parsing functionality on type-level.

Table 4.3: Three extra traits that are used in addition to the traits from Table 4.1 to implement a buffered CSV to Arrow parser using a derive macro.

```rust
pub trait CsvBufReader: ArrowCsvRecord {
    fn buffered_reader<R: Read>(reader: R) -> impl ArrowCsvBufReader<R>;
}
```

(a) The trait for constructing an instance of a buffered reader from a reader. This trait is implemented on an `ArrowCsvRecord` implementation.

```rust
pub enum BufferedParserState {
    Delayed,  // Parsing is delayed, more bytes are needed and are possibly coming.
    Result,   // Parsing is finished, bytes are left for other field parsers.
    Finished, // Parsing is completely finished, no more bytes are left.
}
```

(b) An enum that defines the three possible states after parsing a value.

```rust
pub trait ArrowCsvBufReader<R: Read>:
    Iterator<Item = Result<RecordBatch, ArrowError>> + From<R>
{
    fn parse_record(&mut self) -> Result<BufferedParserState, ArrowError>;

    fn finish(&mut self) -> Result<RecordBatch, ArrowError>;

    fn read(&mut self) -> Result<Option<RecordBatch>, ArrowError>;
}
```

(c) The trait that defines the functionality of a buffered reader. That is, it must implement a record batch iterator and be able to be constructed from bytes. Additionally, it defines methods for progressing the reader incrementally whilst maintaining state. The state is defined by the enum in b.

Figure 4.9: The traits used for defining the functionality buffered CSV reader.

perform buffered reading.

For a buffered reader, the field builders have to be stored in an array as opposed to dedicated struct fields. The reason for this is that unlike unbuffered reading, the reader is allowed to stall parsing in the middle of a record if no bytes are left. More bytes might arrive at a later time which resumes parsing. Consequently, it would be hard to resume record parsing at any arbitrary field if the field parsers are inlined like the unbuffered reader. Hence, all builders are stored in an array and by maintaining the index of the last field that is parsed, parsing can be resumed with a simple array lookup.

Buffered builders

Since each builder could have a different type, the builders are stored as dynamic trait objects. Consequently, they are stored according to Figure 3.2, i.e. the builder structs should be stored on the heap by boxing them. Each builder implements the `BufferedTypeBuilder` trait, a custom trait used to combine an Apache Arrow array builder with a custom type parser implementation. As seen in Figure 4.11, it defines four methods to parse types and construct an Apache Arrow array. The `parse` method is defined to parse a value from input bytes where parsing could be delayed. It can produce either a result containing the state of parsing and the number of bytes consumed, or an error indicating that a value is invalid to parse. The parse state is defined by the `BufferedParserState` enum in Figure 4.9b, it defines three types of state representing the following situations.

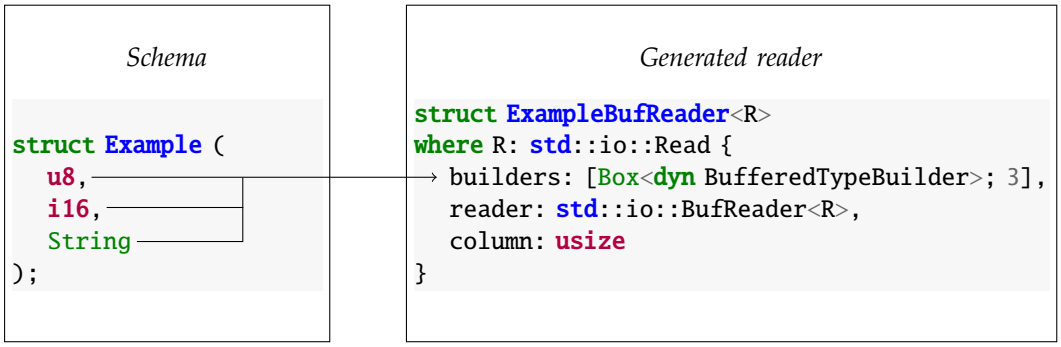


Figure 4.10: An example of generating the code of a buffered reader struct from an input schema struct. All types are stored as dynamic trait objects.

1. Parsing can be delayed if more bytes are needed or possibly coming.
2. Parsing can yield a result, and bytes are left for other field parsers.
3. Parsing can be finished if there are no more bytes left.

When the state indicates that either a result was yielded or parsing has finished, a value is stored to the underlying buffer. In the case of parsing being delayed but no more bytes are available, the builder might still hold data that can be evaluated to a result value. For example, in the case of integers, if a parser is delayed when it has already parsed the digits 1 and 0, it can store the digits. When no more bytes are available, the state could be evaluated to the number 10. The `flush` method is used to try and evaluate the state to a value and store it. It is only used in the case all bytes are exhausted, hence it returns either nothing or an error. To check that a parser did not start parsing yet, the `empty_state` method is provided. Finally, the `finish` method is defined to construct an Apache Arrow array from the underlying value buffer which can be used to construct a record batch.

```
pub trait BufferedTypeBuilder {
    fn parse(&mut self, input: &[u8]) ->
    ↪  Result<(BufferedParserState,usize),ArrowError>;

    fn flush(&mut self) -> Result<(), ArrowError>;

    fn empty_state(&self) -> bool;

    fn finish(&mut self) -> ArrayRef;
}
```

Figure 4.11: The trait that defines the functionality for parsing and storing a type where parsing can be delayed by managing a state.

To simplify selecting a buffered builder for a type, a generic struct definition can be used for which the `BufferedTypeBuilder` trait can be implemented. For this reason, the `BufferedBuilder` struct is defined with with a builder and a state field that are both defined by a generic type as seen in Figure 4.12. Using generic types, multiple implementations can be provided for the struct. The generic builder type is bound so that it implements the Apache Arrow `ArrayBuilder` trait that Apache Arrow uses to implement their type builders. The generic state type is only bound to being able to construct a default value. A generic boolean argument is provided so that a buffered builder with the same builder and state can be defined with different implementations for nullable and non-nullable types. Using these generic arguments, a buffered builder for the `u8` type can be defined as `BufferedBuilder<UInt8Builder, Option<u8>, false>` for which the `BufferedTypeBuilder` trait can then be implemented. By specifying different arguments to the struct, different implementations can be selected in the code generation process.

```rust
pub struct BufferedBuilder<B, S, const NULLABLE: bool>
where B: ArrayBuilder + Default, S: Sized + Default,
{
    builder: B,
    state: S,
}
```

Figure 4.12: The `BufferedBuilder` struct definition used for implementing the `BufferedTypeBuilder` trait for different types.

Buffered reading functionality

Second, two traits are used to define the functionality for buffered reading. The traits for buffered reading are defined as seen in Figure 4.9. Furthermore, an overview of the traits used to implement the buffered reader can be found in Table 4.3.

The two traits, `CsvBufReader` and `ArrowCsvBufReader`, are similar to the unbuffered reading traits, but have two main differences. First, the `CsvBufReader` trait defines the construction of a buffered reader by providing an input source that implements the Rust `Read` trait. Second, the `ArrowCsvBufReader` trait defines additional functionality compared to the `ArrowCsvReader` trait. Figure 4.9c shows that the `ArrowCsvBufReader` trait defines methods that return a state. These methods are used to manage state and resume parsing when it could not be completed. For example, when not all bytes of a record are yet present in the input data. The state that parsing a record can return is the same `BufferedParserState` enum (see Figure 4.9b) as the state used by the `BufferedTypeBuilder` trait.

The `From<R: Read>` trait defines that a reader can be statically constructed from any type that implements the `Read` trait. The implementation for this trait is generated differently compared to the unbuffered implementation for two reasons. First, the field builders are stored in an array as opposed to dedicated fields. Second, they need to be boxed so that they can be stored as dynamic trait objects. As such, each builder is initialized, boxed and stored in an array. The builders are custom builders that implement the `BufferedTypeBuilder`. Additionally, a reader source is taken from the input argument of the function and stored in the buffer field by initializing a `BufReader`. Figure 4.13 shows an example of generating the implementation for the `From` trait. Using this implementation, the `CsvBufReader` trait can be implemented for the schema struct. This implementation can now simply call the static `from` method from the generated reader struct.

Schema

```rust
struct Example (
   u8,
   i16,
   String
);
```

Generated implementation

```rust
impl<R> From<R> for ExampleBufReader<R>
where R: std::io::Read {
   fn from(value: R) {
      Self {
         builders: [
            Box::new(BufferedUInt8Builder::default()),
            Box::new(BufferedInt16Builder::default()),
            Box::new(BufferedStringBuilder::default()),
         ],
         reader: value,
         column: 0
      }
   }
}
```

Figure 4.13: An example of generating the code for the `From` trait implementation from a tuple struct. The builders are custom builders that each implement the `BufferedTypeBuilder` trait allowing them to be stored together as dynamic trait objects.

The `Iterator` trait for the buffered reader is implemented by delegating reading to the `read` method. By default this method is implemented as seen in the control flow diagram in Figure 4.14. Records are parsed using the `parse_record` method which returns one of the three enum states (see Figure 4.9b). In case of the the delayed state, it will do nothing and try parse the record again. For the result state the reader will move to the next record. Alternatively, the finished state will halt record parsing after which the finish method will be called to construct a record batch. The finish method is similar to that of the unbuffered reader, since it will first convert the buffers from the field builders to arrays and then construct a record batch.

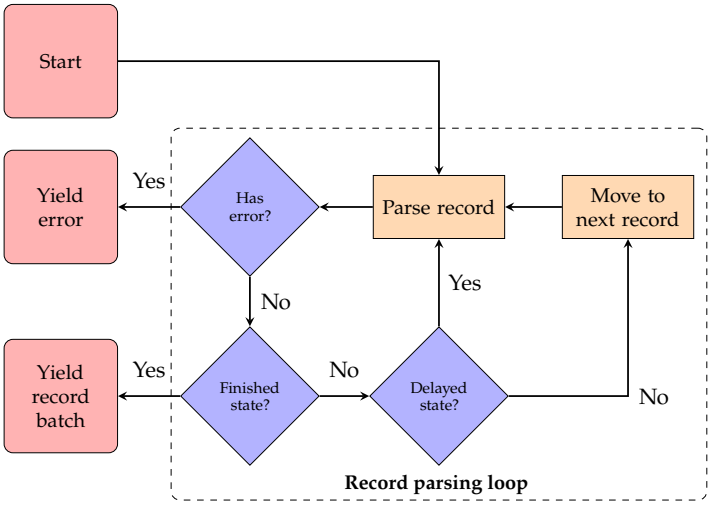


Figure 4.14: The control flow diagram of the `read` method implementation for a buffered CSV to Arrow reader. See figure 4.15 for more details on the parse record process.

The `parse_record` method parses a record by parsing fields in sequence using the parse implementation of their builders. It starts by filling the input buffer after which it will parse the fields in sequence using a loop. Each field can delay parsing when more bytes are needed, for which the buffer will be filled and parsing is attempted again. When all fields are parsed, the result state will be yielded, or a finished state in case no more bytes are left. A control flow diagram for the `parse_record` method can be found in Figure 4.15.

4.1.5. Buffered type parsing

For buffered reading, parsing becomes more difficult as not all the data might be available at any time. For example, a reader might end up in a situation where only half of the bytes of a record are available for parsing. These type of situations require that parsing can be delayed and resume when more bytes are available. As such, additional control logic is needed for maintaining the state whilst parsing.

The control logic needed for buffered parsing can be defined by the `BufferedTypeParser` trait as seen in Figure 4.16. It needs to be implemented for any type that needs to be parsed from bytes. It defines a parse and a flush method that handle parsing using a state, allowing a type to be parsed incrementally. Whilst the `BufferedTypeParser` trait needs to be implemented for a concrete type, it also defines a generic value type. This type is defined so that the trait can also be implemented for option types and still have access to the concrete inner type of the option. For example, the trait can be implemented for a type such as `Option<u8>` whilst setting the Value type to `u8`. Additionally, a generic state type is defined which is used when delaying parsing in the parse method.

The parse method takes bytes and a state as input and produces either a `TypeParserState` result or an error. The `TypeParserState` is an enum that indicates the state of parsing with two options. First, parsing can be delayed, returning only the state needed for future continuation. This state is represented by the `TypeParserState::Delayed(S)` variant, where S is the generic state type. Second, parsing can yield a result, returning only the parsed value. This state is represented by the `TypeParserState::Finished(V)` variant, where V is the generic value type. Using the `TypeParserState`, the parsing state can be fed to the parse method when bytes are available and parsing was not finished. Otherwise, when it yielded a

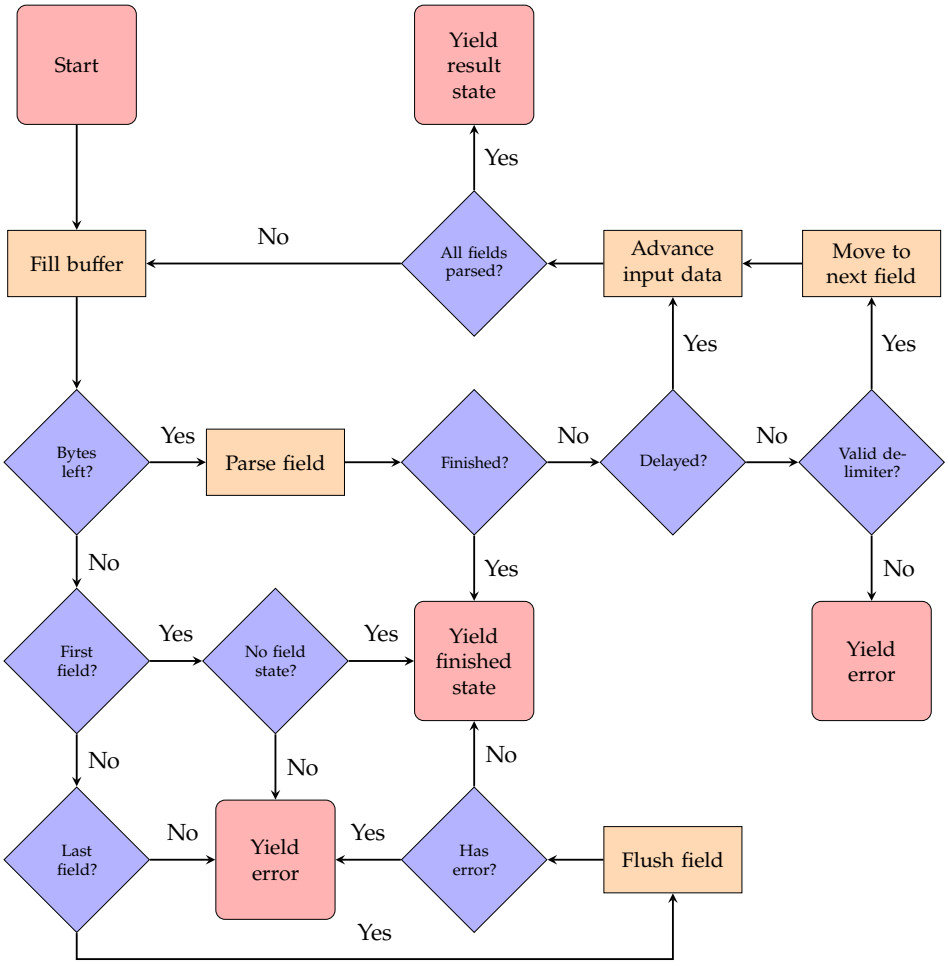


Figure 4.15: The control flow diagram of the `parse_record` method implementation for a buffered CSV to Arrow reader.

value, parsing is finished. The flush method is provided for when parsing is delayed but all bytes are exhausted. It has the same functionality and purpose of the flush method in the `BufferedTypeBuilder` trait. A control flow diagram of how these methods are used to achieve buffered parsing for a type can be found in Figure 4.17.

The `BufferedTypeParser` trait is implemented for the types in Table 4.4 to allow for buffered type parsing. Since the type implementations in Table 4.2 do not support managing state by default, custom implementations for the unsigned and signed integer types and the string type are provided. Unsigned integer parsing is carried out by first checking characters to be digits and trying to add these digits with the correct significance[2]. The parser will throw an error when an overflow would occur and when parsing is delayed, either no value or intermediate value is returned. Signed integer parsing uses a similar approach except that it has to keep track of a sign. Consequently, the state returned is either no value or an enum that tracks the sign. The state holds either a negative value, which is possibly null if only the sign has been parsed, or a positive value. The string parser is implemented using the same approach as the unbuffered parser, thus it uses a custom boiled down version of the `rust-csv` [8] library. When delayed, it will return a number that indicates the state used in the DFA. For the floating point number, the `lexical core` [22] library is used since creating a custom floating-point parser is complex. For this reason, the parser is implemented by first finding the end of the field and then calling the `lexical core` parser. In order to do so, a state is defined to hold tuple of a 64 byte buffer and an offset that are used to store bytes of previous parsing attempts. No state is returned when the parser is delayed without having processed byte and the tuple is returned otherwise. The implementations for these types can be found in the `arrow-csv-reader` [37] module.

[2]Numbers in CSV use decimal digits and therefore require multiplying by 10 when moving to the next digit.

```rust
pub enum TypeParserState<S: Sized, V: Sized> {
    Delayed(S), // indicates parsing was delayed and maintained state `S`
    Finished(V), // indicates parsing was completed and yielded a value `V`
}

pub trait BufferedTypeParser: Sized {
    /// The state that is maintained during several parse iterations.
    type State: Default;
    /// The type of a value that is to be constructed.
    type Value;

    fn parse(input: &[u8], state: Self::State,) ->
    ↪  Result<(TypeParserState<Self::State, Option<Self>>, usize), ArrowError>;

    fn flush(_state: Self::State) -> Result<Option<Self::Value>, ArrowError>;
}
```

Figure 4.16: The `BufferedTypeParser` trait that defines the functionality for parsing a type where parsing can be delayed using state results.

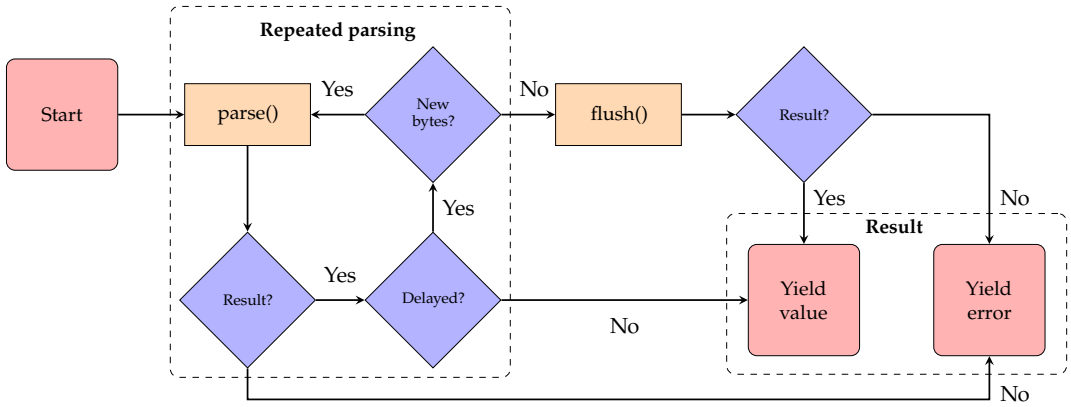


Figure 4.17: The control flow diagram of buffered parsing for a type using the `BufferedTypeParser` trait methods.

4.1.6. Optimize using size-bounds

Knowing the maximum byte representation of a schema in CSV allows for reducing bounds checking on the input bytes during parsing. This can be particularly useful for buffered reading, as buffered reading introduces more bound checks. A reason for this is that in buffered reading, the input source has to be managed by checking if it is empty and then filling it with bytes. Reducing these checks could help lower the overhead and perform much like the unbuffered reader that does not have to manage a buffer.

In order to lower the number of input checks, several parameters need to be defined. First, define the worst case byte representation of a schema for a CSV record to be $S_{\max}$. $S_{\max}$ is the case where all fields have their maximum textual representation including delimiters. Then define the number of available bytes from the input source to be N. The minimum number of records $R_{\min}$ that are present in N input bytes will be $\lfloor N/S_{\max} \rfloor$. Consequently, if N input bytes are available, $R_{\min}$ records can be parsed without checking if enough bytes are available. This allows for skipping many compare instructions, which reduces the number of branches encountered.

Nevertheless, this technique only works when several criteria are met. First, $S_{\max}$ needs to be known, requiring every field to have a known size. Second, the input data should respect these maximum sizes. For instance, a textual representation of a number in CSV should not have a value that is larger than this number can physically represent. Any illegal value could lead to undefined behavior. In general, CSV data is generated by systems using native types, so for types such as numbers this should not happen. However, for dynamic types such as strings, the safety of this technique relies on the user providing the

Category	Type	Variants	Implementation
Number	Unsigned integer	8-, 16-, 32-, 64-bit	custom
	Signed integer	8-, 16-, 32-, 64-bit	custom
	Floating-point	32-, 64-bit	lexical core [22]
String		escaped, unescaped	custom (rust-csv [8] based)

Table 4.4: The parser for each type used in generating buffered parsers.

right size bounds. For this reason, optimizations using size bounds is turned off by default. It can be turned on by using the `size_inference` helper attribute on a schema, as seen in the following sections.

Bounds of fixed sized types

Some primitive types in Rust are bound by a physical size and thus have a fixed range of possible values. For instance, the `u8` type has a physical representation of 1 byte and a value range from 0 to 255. Since the data in CSV are textual representations of values, these value ranges can be used to infer the physical textual representation of such a type. For `u8` , the smallest textual representation are the numbers 0 to 9, i.e. only 1 byte. The largest textual representation are the numbers 100 to 255, i.e. 3 bytes. An overview of the bounds of all integer primitives can be found in Table 4.5.

	Value range		*Text / digit range*	
Type	**Min**	**Max**	**Min**	**Max**
u8	0	255	1	3
u16	0	65,535	1	5
u32	0	4,294,967,295	1	10
u64	0	18,446,744,073,709,551,615	1	20
i8	-128	127	1	4
i16	-32,768	32,767	1	6
i32	-2,147,483,648	2,147,483,647	1	11
i64	-9,223,372,036,854,775,808	9,223,372,036,854,775,807	1	20

Table 4.5: The value and text / digit bounds for integer primitives.

When a schema is provided with only size-bounded types, a size bound can be inferred and put on that schema. For example, in Figure 4.18 two structs are bounded in size by their fields, allowing for a maximum byte (or textual) representation of these records.

```
// Max byte representation = 3 + 5 + 10 + 20 = 38 bytes
#[derive(ArrowCsvRecord)]
#[infer_size]
struct BoundedInt (u8, u16, u32, u64);

// Max byte representation = 4 + 6 + 11 + 20 = 41 bytes
#[derive(ArrowCsvRecord)]
#[infer_size]
struct BoundedUInt (i8, i16, i32, i64);
```

Figure 4.18: Two size bounded structs with their respective maximum byte (or textual) representation.

Bounds of dynamic sized types

Bounding types is simple for primitive types such as integers and even floating point numbers. However, for dynamic sized types such as byte vectors or strings it is not possible to infer the bounds. Nevertheless, in practical applications strings are often limited in size. For instance, names, flags or comments are often limited to a maximum. GUIDs or timestamps can even be fixed size. Consequently, this parser generator allows for annotating size hints for types. Figure 4.19 shows that annotating a String field to have a maximum size, can complete the maximum bounds of the whole record.

```rust
// Max byte representation = 20 + 5 + 32 = 57 bytes
#[derive(ArrowCsvRecord)]
#[infer_size]
struct Example {
    id: u64,
    birth_year: u16,
    #[size_hint(max=32)]
    name: String
}
```

Figure 4.19: A struct that is completely size bounded by annotating the dynamic String to have a maximum number of characters.

As mentioned earlier, a size hint should be provided with care. Providing a size that is not actually the maximum size could lead to undefined behavior because bounds can be violated. It is especially safer to use for values such as a GUID that has fixed length strings.

Applying size bounds

To apply the knowledge of the maximum byte representation and reduce checks, the implementation of the reader needs to be adjusted. The first change is to add an extra trait that represents the relaxation of bounds for types. As such, the `BufferedTypeParserBounded` trait is defined as seen in Figure 4.20. This trait introduces an unsafe method called `parse_unchecked` which adds an extra argument and has different result type compared to the `BufferedTypeParser` trait. The method is marked as unsafe since an implementation can call unsafe functions such as the `get_unchecked` [3] method for byte slices. Furthermore, it has an extra argument called `upper_bound` . This upper bound is the maximum number of bytes by which the type is represented in CSV. Moreover, the result type is different since no state should have to be managed given that this method is only called when the upper bound number of bytes is present in the input. By default, the `parse_unchecked` method calls the safe `parse` method from the `BufferedTypeParser` that is required to be implemented. The `BufferedTypeParserBounded` implementations for the types found in Table 4.4 are implemented in the `arrow-csv-reader` [37] module.

```rust
pub trait BufferedTypeParserBounded: BufferedTypeParser {
    unsafe fn parse_unchecked(input: &[u8], upper_bound: usize) ->
    ↪ Result<(Option<Self>, usize), ArrowError> {
        let (result, n) = Self::parse(input, Default::default())?;
        match result {
            TypeParserState::Finished(value) => Ok((value, n)),
            TypeParserState::Delayed(_) => unreachable!("Knowing the bounds, this
            ↪ method is only called when enough bytes are available to parse a
            ↪ value."),
        }
    }
}
```

Figure 4.20: The `BufferedTypeParserBounded` trait that adds an extra method to use when the maximum byte representation of a schema is known.

Additionally, the `ArrowCsvBufReader` trait has to be updated to contain a new method for parsing a record. For this reason, the `parse_record_unchecked` method is added, with exactly the same signature as the `parse_record` method. By default it is implemented by inlining the `parse_record` method. Furthermore, the `BufferedTypeBuilderUnchecked` trait is added to extend the functionality of the `BufferedTypeBuilder` trait as seen in Figure 4.21. This trait simply adds an unchecked parse method for types that requires no state management and returns only the number of bytes consumed. Moreover, the `BufferedBoundedBuilder` struct is defined as a wrapper for the `BufferedBuilder` struct. It adds

[3] This `get_unchecked` method is the same as retrieving a value from an array by index, however it does not perform bounds checking. Hence, performing less instructions.

an extra generic argument called `UPPER_BOUND` that provides the maximum byte representation upper bound, which can be used in code generation. As seen in Figure 4.22, it currently only support non null types.

```
pub trait BufferedTypeBuilderUnchecked: BufferedTypeBuilder {
    unsafe fn parse_unchecked(&mut self, input: &[u8]) -> Result<usize, ArrowError>;
}
```

Figure 4.21: A trait that extends the `BufferedTypeBuilder` trait with an unchecked parse method.

```
#[derive(Default)]
pub struct BufferedBoundedBuilder<B, S, const UPPER_BOUND: usize>(
    BufferedBuilder<B, S, false>
) where B: ArrayBuilder + Default, S: Sized + Default;
```

Figure 4.22: A struct that acts as a wrapper for the `BufferedBuilder` struct to supply the maximum byte representation upper bound as an extra generic argument. Only types that can not be null are supported.

Using the new methods, the implementation for the `read` method for the `ArrowCsvBufReader` trait can be generated to incorporate reduced bounds checking. When the size bounds are known, the following control logic is added. Instead of repeatedly trying to parse a record, the minimum number of records is calculated using the maximum size of a record every time the input buffer is filled. Then for this number of records, the `parse_record_unchecked` method is called instead of the `parse_record` method. Hence, these records are parsed with less overhead. When these are parsed, the buffer is filled again and the process is repeated until either enough records are parsed or the input buffer is exhausted. If the minimum number of records in the input buffer is zero, the `parse_record` method is called to parse the remaining records.

4.2. Just-in-time parser generation

The solution for just-in-time parser generation is to create a library that can construct a parser in memory. Since the focus of this work is on ahead-of-time parser generation, just-in-time parser generation is briefly explored for performance comparison (see Section 5.2). More specifically, only an unbuffered reader is implemented and compared. The implementation for the just-in-time parser generator can be found in the `arrow-csv-reader-jit` [38] module.

4.2.1. Generating a reader

The generation process of a just-in-time parser is similar to that of the ahead-of-time buffered parser but executed at run-time. The library defines an `ArrowCsvParser` struct, as seen in Figure 4.24, that can be constructed by providing it an Apache Arrow schema and an input data byte slice. The struct holds an array of dynamic trait objects for the fields, the schema, a reference to the input bytes and the maximum number of records to parse. In order to construct it, each field in the schema instantiates a dynamic `CsvColumnParser` trait object for its type implementation. This trait also requires the `ArrowTypeBuilder` trait to be implemented, allowing it to construct an Apache Arrow array. Figure 4.23 shows the definition of both traits. Similar the ahead-of-time buffered reader, these objects can be accessed by using dynamic dispatch to call their methods.

The reader is executed by using it as an iterator like the ahead-of-time implementations. Each call to the `next` method will try to read enough records to construct an Apache Arrow record batch. Each record is parsed by calling the parse method of the dynamic `CsvColumnParser` objects in the order of fields. When no more records can be parsed, the `build_array` method is called on the fields and used to construct record batch.

4.2.2. Parsing types

For the limited exploration of the just-in-time solution, only signed and unsigned integer types are supported. For both, the `atoi` [16] library is used to parse the values, which the unbuffered ahead-of-time reader does as well.

```rust
pub trait ArrowTypeBuilder {
    fn build_array(&mut self) -> ArrayRef;
}

pub trait CsvColumnParser: ArrowTypeBuilder {
    fn parse<'a>(&mut self, bytes: &'a [u8]) -> Option<&'a [u8]>;
}
```

Figure 4.23: The two traits used to define the builder and parser of a field. The `CsvColumnParser` trait is used as the dynamic trait object type to store the builders.

```rust
pub struct ArrowCsvParser<'a> {
    schema: Arc<Schema>,
    parsers: Vec<Box<dyn CsvColumnParser>>,
    bytes: &'a [u8],
    max_batch_size: usize,
}
```

Figure 4.24: The struct that acts as the reader for parsing CSV to Arrow.

4.3. Multi-threaded parsing

As explained in Section 1.2.3, parsing CSV in parallel is challenging. CSV is difficult to parse since splitting it in chunks requires knowing the starting context of each chunk. I.e. whether the first character is escaped or not. Solving this challenge means that the throughput of CSV parsing could be scaled by the number of threads.

To identify the potential of multi-threaded CSV parsing and its limitations, a custom multi-threaded reader is implemented that works as follows. The reader consists of a single scanner thread and multiple worker threads. The scanner identifies where CSV records start and will provision worker threads with chunks of data which they can safely process. Each worker thread will act as a CSV to Arrow reader, and when the works have gathered enough records the scanner thread will construct a record batch. By benchmarking this reader for a different number of threads, and datasets of increasing size, its bottleneck and characteristics are identified. Section 5.3 reveals that the scanner bottlenecks the other threads by not supplying their contexts fast enough. Consequently, scanning should be fast to allow complete utilization of each thread.

SIMD can be used to quickly scan delimiters using several steps. First, bytes need to be converted to double quote bitmaps that indicate whether a value is a double quote or not. Second, the double quote bitmaps can be used to calculate bitmaps for escaped characters as described in Section 2.1.2. These bitmaps can be inverted to get bitmaps for unescaped characters. Finally, using a bitwise AND operation, bitmaps created for record delimiters from the input bytes can be overlayed with the unescaped bitmaps to produce bitmaps of unescaped record delimiters. Each step in this process can be executed by using AVX or AVX2 instructions, except for bit propagation between the bitmaps of adjacent input data as discussed in Section 2.2. Nevertheless, every other step can therefore be highly parallelized and the result can be used to quickly scan the positions of records. The implementation and experiments for this process can be found in the `simd` [39] module for experiments. Furthermore, the results can be seen in Section 5.3.

Since multi-threading and SIMD add an extra layer of complexity to parser generation, these features were not able to be incorporated due to the limited time of this research. Nevertheless, they could provide work for the future as described in Section 6.2.2.

5

Experimental results

5.1. Experimental setup

5.1.1. System specifications

The experimental results were obtained on a desktop PC. The specifications for the CPU and memory can be found in Table 5.1. Every benchmark performed makes use of the criterion benchmark framework. This framework performs repeated benchmark executions and performs a warm up for each benchmark. In practice, the datasets used in the benchmarks got cached by the operating system during a warm up. Consequently, the storage device used by this setup did not impact the performance and its details are omitted. This was tested by performing a simple benchmark that only reads the bytes of an input file. The result was a throughput that matched the memory read bandwidth found in Table 5.1b.

Name	**Clock**		**Cores**	**Threads**	**Cache**		
	Max. Boost	*Base*			*L1*	*L2*	*L3*
AMD Ryzen 9 7900	5.4 GHz	3.7 GHz	12	24	768 KB	12 MB	64 MB

(a) CPU

Name	**Type**	**Size**	**Configuration**	**Speed**	**CAS Latency**	**Bandwidth**[a]	
						Read	*Write*
Corsair Vengeance	DDR5	2x16GB	Dual-channel	6000 MT/s	30	50 GB/s	27 GB/s

(b) Memory

[a]The read and write bandwidth were measured using `sysbench` [17].

Table 5.1: The CPU and memory used in carrying out the experiments.

The operating system used for experiments is Ubuntu 24.04.3 LTS. Moreover, other versions for frameworks and tools used during experiments can be found in Table 5.2.

Tool	**Version**
Rust	1.89.0-nightly
Criterion	0.5.1

(a) The versions for tools and frameworks used for performing benchmarks.

Framework	**Version**
Apache Arrow	55.0.0
Polars	0.51.0
DuckDB	1.4.1

(b) The versions for data analytic frameworks/libraries to benchmark.

Table 5.2: The versions of tools and frameworks used in the experiments.

5.1.2. Metrics

To analyze and compare the CSV to Arrow parsers, benchmarks are constructed to measure their throughput over different datasets. These benchmarks vary in their size and type distribution. The

throughput over different sizes can show whether a parser has a start-up overhead. Furthermore, it can display memory hierarchy characteristics that it might depend on. I.e. for different sizes the CPU cache might provide a benefit for some implementations. Alternatively, the type distribution of a dataset can display the characteristics and performance of the different type parser implementations. For this purpose, homogeneous typed synthetic datasets are used to analyze the performance per parser per type. Moreover, real-world datasets are used to compare parsers in practical applications. Often these datasets have more variety in types which displays the overall performance and characteristics of a parser.

Using these input data metrics, the performance of parsers can be measured using benchmarks. In this project, the performance of a parser is defined as the number of bytes a parser can process each second. This throughput makes it easy to compare the performance between benchmarks of different size, type, and parser.

5.1.3. Synthetic datasets

Performing benchmarks using synthetic datasets allows identifying the type-specific performance characteristics of a parser. It is important to find such characteristics, as it exposes the performance of parsing the string to a type in converting CSV to Arrow. All synthetic datasets used in benchmarks are generated by a custom developed CSV generator [35].

Parameters

The CSV format has a limited number of parameters in which documents can vary. This work recognizes the parameters found in Table 5.3. Together, they can be used to form different types of datasets. However, the number of variations would be very high if no limits were put on parameters 1 or 2. For this reason, the number of columns is fixed for synthetic datasets. This should not affect the characteristics of the performance results, since both parameters linearly increase the size. However, the choice for fixing the columns is more practical when representing the schema as a struct.

#	Parameter	Description
1	Number of columns	Defines the number of fields in a record
2	Number of rows	Defines the number of records
3	Data types	Defines the values of the fields in a record
4	Escaped field	Defines if a field may contain delimiters

Table 5.3: The four possible parameters in which a CSV document can vary. Parameter 1 & 2 define the dimensions of the dataset where as paramter 3 & 4 define properties of the columns.

Generating values

The CSV generator supports random generation of several types, and variants, which are found in Table 5.4. Remember that the CSV format allows any field to be escaped since they are treated as text. This means that in theory, any type, such as a number, could have an escaped variant. However, for simplicity such variants are ignored for types other than strings. The reason for this is that by the textual definition of numbers [13, 12] , they will not contain delimiter characters such as double quote, comma and newline characters. As mentioned in Section 1.1, this work only considers the RFC-4180 [32] specification. Therefore no other delimiters are considered that could break this assumption.

Category	Type	Variants	Parameters
Number	Unsigned integer	8-, 16-, 32-, 64-bit	N.A.
	Signed integer	8-, 16-, 32-, 64-bit	N.A.
	Floating-point	32-, 64-bit	N.A.
	String	escaped, unescaped	minimum and maximum size

Table 5.4: The types, with possible variants and parameters, available to be used in generating random CSV documents.

The values for the categories found in Table 5.4 are generated using the rand crate [30] and an rng seed from the operating system. Since all experiments run on Linux, these values are generated using random data sources from the host system [9]. The first category of types is the number type, which has

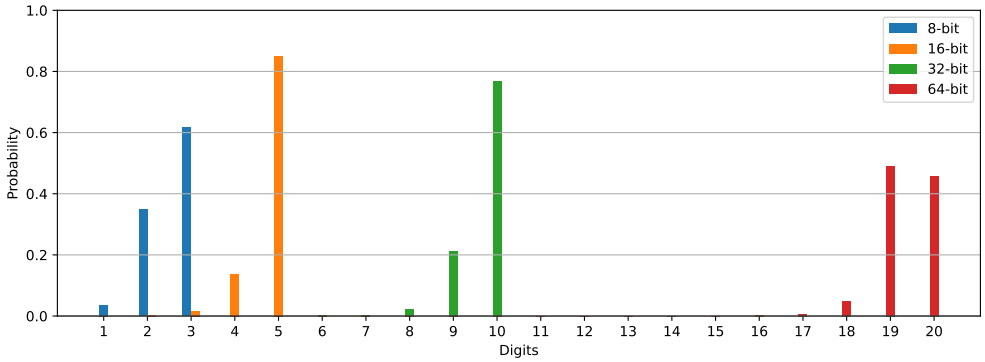


Figure 5.1: The discrete distribution of the number of digits for the 8-, 16-, 32-, and 64-bit unsigned integer value range.

no parameters available to tweak the corresponding generated values. The number types have several variants, that are distinguished by their physical representation. Consequently, due to the difference in their physical representation, their value range, and thus their textual length, is also different. Figure 5.1 shows how the physical size of a number, or value range, impacts the probability of digits, which is directly related to its textual representation.

The last category is strings, which is different from the other categories because of its variable length and escaped variants. Since strings are variable in length, both a minimum and a maximum length parameter can be provided. For each string value, the generator will uniformly pick a number between the minimum and maximum length, determining the length of the value. It will then construct the string value by repeating the character 'O' to match the length. The result is a homogeneous string, which in the context of the CSV format has the same characteristics as any other unescaped string. I.e. it does not matter what character is present in the string as long as it is not an illegal character. A string of a single repeated character should not have a different performance outcome when parsing than any other arbitrary string of the same length. When a value is an escaped variant, a double quote prefix and suffix is added resulting in two more characters. Additionally, when the length of the value is larger than 1, an escaped double quote is added at a random position which is picked uniformly. Examples of the generated unescaped and escaped strings can be found in Figure 5.2.

```
0000000000000000,00000000000
000000000,000000000000
0000000000000000,00000000000
000000000000,00000000
```

(a) Example of generated unescaped strings.

```
"0000000""0000","000000""000"
"000000000""00","000000""0"
"00000000000""0","0000""00"
"0000""000","000000""000"
```

(b) Example of generated escaped strings.

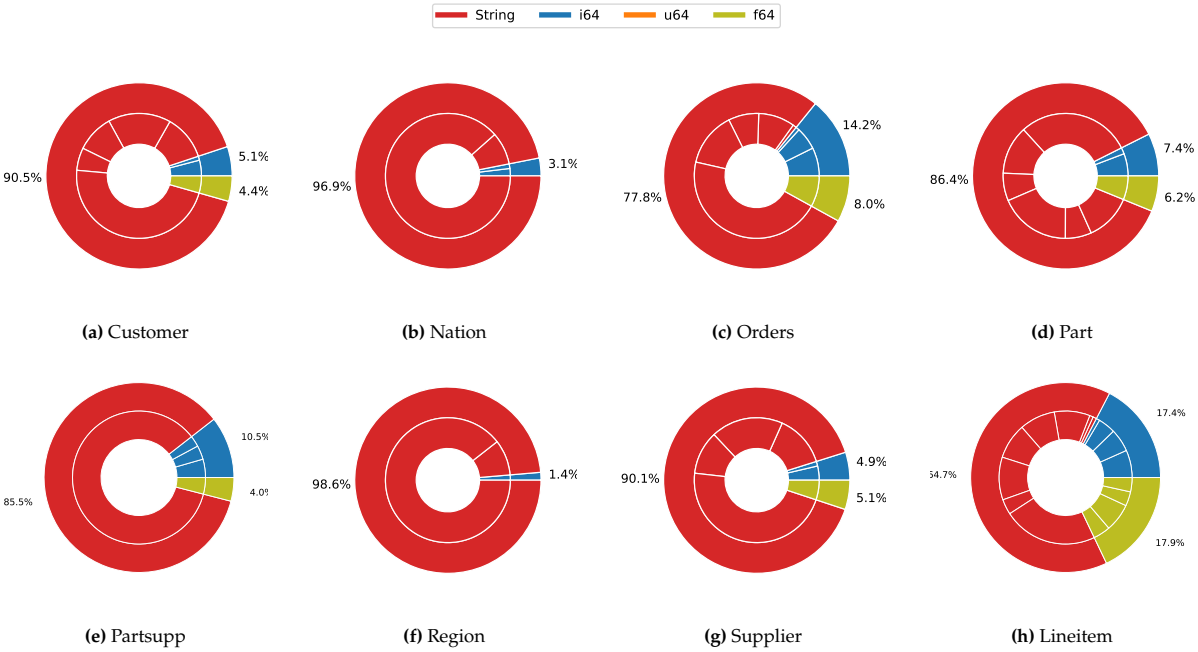
Figure 5.2: Two examples of generated unescaped and escaped generated strings.

Generating datasets

Each synthetic dataset consists of a homogeneous schema, i.e. all fields are the same type. Additionally, their schema is set to have exactly 8 fields (or columns). The datasets used in the benchmarks are generated by specifying a number of rows, starting with 1024. Each subsequent dataset has the number of rows multiplied by 2. The synthetic data are generated by a custom CSV generator available in the repository for all source code from this research [37].

5.1.4. TPC datasets

The TPC datasets represent real world datasets where the type and size of the fields is more variable. These datasets are necessary to analyze the performance without the single type bias that is present in the synthetic datasets. For this purpose the data from two of the TPC benchmarks are used to showcase the performance of CSV to Arrow parsers on real-world data. The first benchmark is the TPC-H benchmark [46] and the second is the TPC-DS benchmark [45]. The data from these benchmarks are not



(a) Customer **(b)** Nation **(c)** Orders **(d)** Part

(e) Partsupp **(f)** Region **(g)** Supplier **(h)** Lineitem

Figure 5.3: The distribution of types in bytes for different tables in the TPC-H benchmark represented in CSV.

immediately available in CSV format. Consequently, these data were generated using the DuckDB core extensions [7] for TPC-H and TPC-DS. The data were generated using the scale factor (sf) 10 and then exported to CSV files.

To further analyze the results of running benchmarks on these TPC datasets, it is useful to know certain dataset characteristics. As discussed in the previous section, several parameters exist in which a CSV file can be different. One of these parameters is the types present in a file. Each CSV field has a type, combining this a distribution of types can be estimated for a file. For instance, a CSV file containing 4 string fields and 4 integer fields could be defined as 50% of the file being each type. Nevertheless, in practice data types can differ in size due to their values. Figure 5.1 shows how likely it is for unsigned integers to have a certain textual representation length. As seen in the figure, the length is not fixed. Moreover, for a string the value length can be even more unpredictable due to its dynamic sized nature. For this reason, counting the characters, or bytes, that are used to represent the values in a CSV file gives a more accurate distribution. Figure 5.3 and Figure 5.4 display the distribution of types in the datasets by having their values counted by the number of bytes needed for each value. Having this insight helps reason about the performance of certain parsers over other parsers.

5.1.5. Benchmarks

To measure the performance of parsing CSV to Arrow for different frameworks, it is important to define benchmarks. Each framework is benchmarked using the Criterion benchmark library in Rust, which allows accurate statistics to be computed. For synthetic datasets, each framework is benchmarked per type with datasets of increasing size. For real world datasets, each framework is benchmarked per TPC benchmark with the respective datasets as input. The benchmarks measure performance in terms of throughput in megabytes per second. Each benchmark starts with setting up the reader for the respective framework. This includes providing a schema and starting the build process. This ensures that frameworks, such as DuckDB, have as little setup overhead measured as possible. Additionally, the frameworks are limited to single-threaded parsing. This is needed to ensure that Criterion can run many samples without saturating the CPU. Moreover, Apache Arrow has no support for multi-threading, which would make comparison with multi-threaded frameworks unfair. The benchmarks can be found in a dedicated module named `benchmarks` [37]. To run the benchmarks, the respective datasets have to be generated and stored.

(a) Call center **(b)** Catalog page **(c)** Catalog returns **(d)** Catalog sales

(e) Customer address **(f)** Customer demographics **(g)** Customer **(h)** Date dim

(i) Household demographics **(j)** Income band **(k)** Inventory **(l)** Item

(m) Promotion **(n)** Reason **(o)** Ship mode **(p)** Store returns

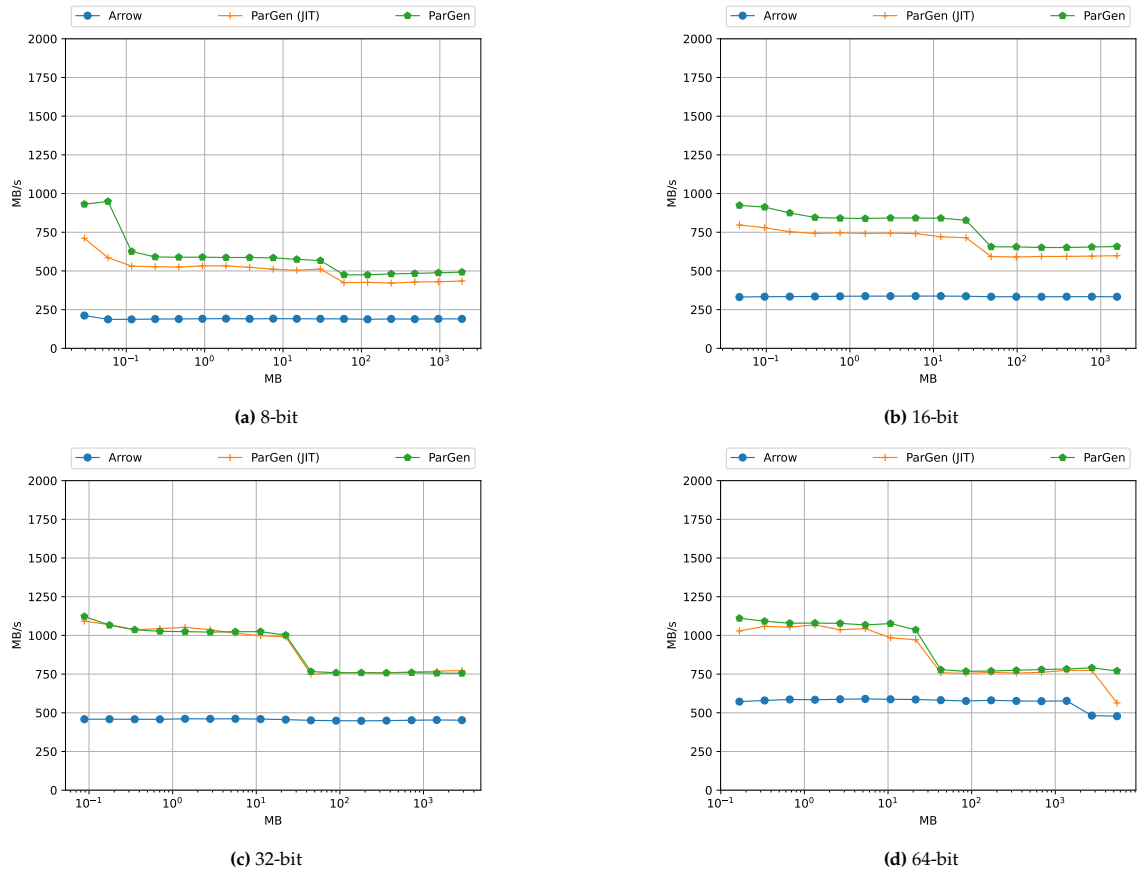
(q) Store sales **(r)** Store **(s)** Time dim **(t)** Warehouse

(u) Web page **(v)** Web returns **(w)** Web sales **(x)** Web site

Figure 5.4: The distribution of types in bytes for different tables in the TPC-DS benchmark represented in CSV.

5.2. Just-in-time vs. ahead-of-time performance

Figure 5.2 shows the result of parsing CSV to Arrow for 64-bit unsigned integer datasets using Apache Arrow, the just-in-time parser generator and the ahead-of-time parser generator. The figure shows that the performance of both parser generators are similar, however the compile-time generated parser always outperforms the run-time generator. This is most likely due to the run-time generated parser having to construct the parser dynamically without any optimizations.



(a) 8-bit

(b) 16-bit

(c) 32-bit

(d) 64-bit

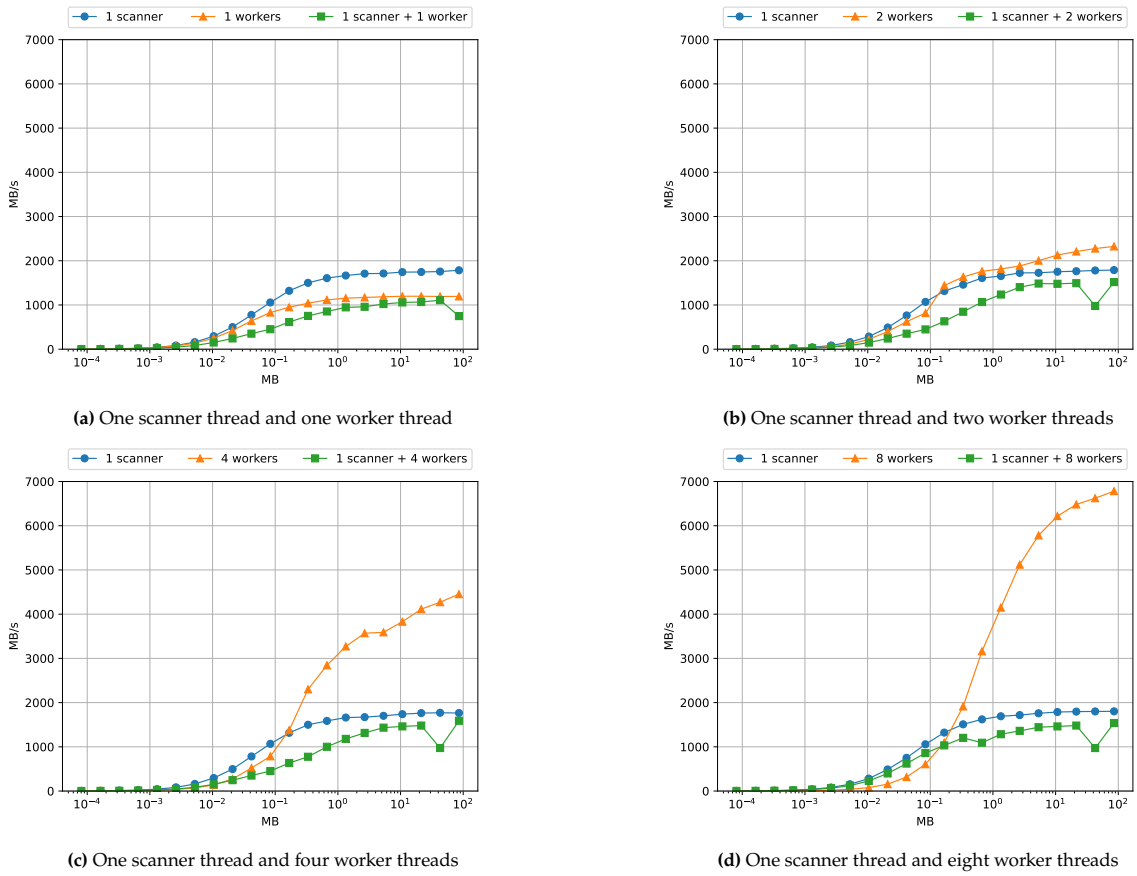
Figure 5.5: The throughput of Apache Arrow, the just-in-time parser generator and the ahead-of-time for parsing variable sized CSV files, containing only unsigned integers.

5.3. Multi-threading performance

In section 4.3, a custom multi-threaded CSV to Arrow parser is defined. It parses CSV to Arrow using one scanner thread and one or more worker threads. To identify the throughput characteristics of this multi-threaded parser, four benchmarks with a different number of worker threads are defined. Theoretically, if the number of threads is doubled, the throughput is expected to also be doubled since double the data can be parsed in parallel. However, in practice this should be lower due to the overhead of thread synchronization, scheduling and other real-world factors. Figure 5.6 shows the throughput of parsing CSV to Arrow with the multi-threaded parser for different size datasets containing only 64-bit unsigned integers. The throughput is measured for four different number of worker threads, 1, 2, 4 and 8 respectively. Furthermore, it shows the throughput measured for its scanner and workers when operating independently. The results show the following characteristics. First, when a single worker thread is used, the throughput of the parser is bottlenecked by the worker thread. Second, when two worker threads are used, they achieve a higher performance than the scanner, and the performance of the parser levels with the scanner. Finally, for four and eight worker threads, they achieve a much higher performance than the scanner, and the performance of the parser remains lower than that of the scanner.

The throughput characteristics indicate that increasing the number of worker threads beyond one,

the scanner thread can not keep up with the throughput of the workers. Consequently, the scanner bottlenecks the parser and the threads will idle. Nevertheless, the worker threads show that scaling their number can increase the throughput if it is not bottlenecked by a scanner.



(a) One scanner thread and one worker thread

(b) One scanner thread and two worker threads

(c) One scanner thread and four worker threads

(d) One scanner thread and eight worker threads

Figure 5.6: The throughput of parsing CSV to Arrow using a custom multi-threaded reader. A scanner thread is used to divide and provision one or more worker threads with CSV data. The figure shows the throughput of the scanner, worker(s) and their combination, ultimately showing that the scanner is the bottleneck when scaling the number of threads. The datasets used are 64-bit unsigned integer only datasets.

Both the Polars and DuckDB frameworks include support for reading CSV using multiple threads. As such, their throughput can be measured against the number of threads to show how efficient they are. Figure 5.7 shows how each of the frameworks performs in parsing CSV to Arrow for different size 64-bit unsigned integer datasets. It displays that DuckDB has a lower throughput than Polars overall. Furthermore, Polars reaches it maximum throughput of 1750 MB/s using 8 threads and DuckDB reaches its maximum throughput of 1250 MB/s using 16 threads. Comparing this to Figure 5.6, which parses the same type of dataset, it is clear that both do not reach the maximum performance of the threads that divide the parsing work. Consequently, if their scanner could be improved, there is more throughput to be unlocked in parser threads.

SIMD could be used to improve the scanner of the multi-threaded implementation as described in Section 4.3. Scanning consists of calculating the bitmaps for escaped characters, after which the bitmaps for unescaped delimiters are computed to perform quick scans. The throughput for each of these SIMD operations using AVX2 instructions can be found in Table 5.5. It shows that record delimiters can be scanned at a speed of 2.8 GB/s, which is approximately 1 GB/s higher than the maximum throughput in Figure 5.7. Consequently, these SIMD operations could improve multi-threaded parsing for both DuckDB and Polars.

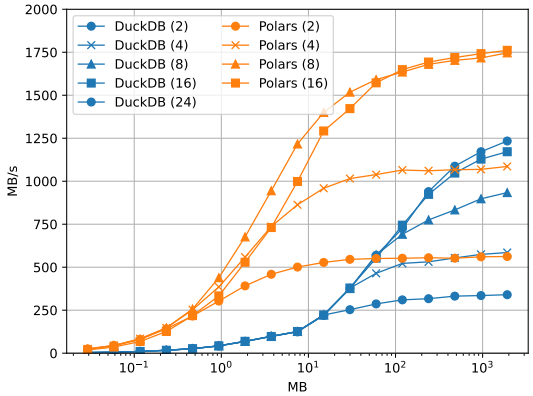


Figure 5.7: The throughput of parsing CSV to Arrow using multi-threaded readers for Polars and DuckDB. Parsing is measured for multiple number of threads to see the maximum throughput and how well threads scale. The datasets used are 64-bit unsigned integer only datasets.

Operation	Throughput
Computing bitmaps for escaped characters	6.5 GB/s
Scanning for record delimiters	2.8 GB/s

Table 5.5: The throughput for two of the SIMD operations needed to quickly scan CSV for record positions. Both operations make use of AVX2 (i.e. operations with 512 bit SIMD registers).

5.4. Synthetic results

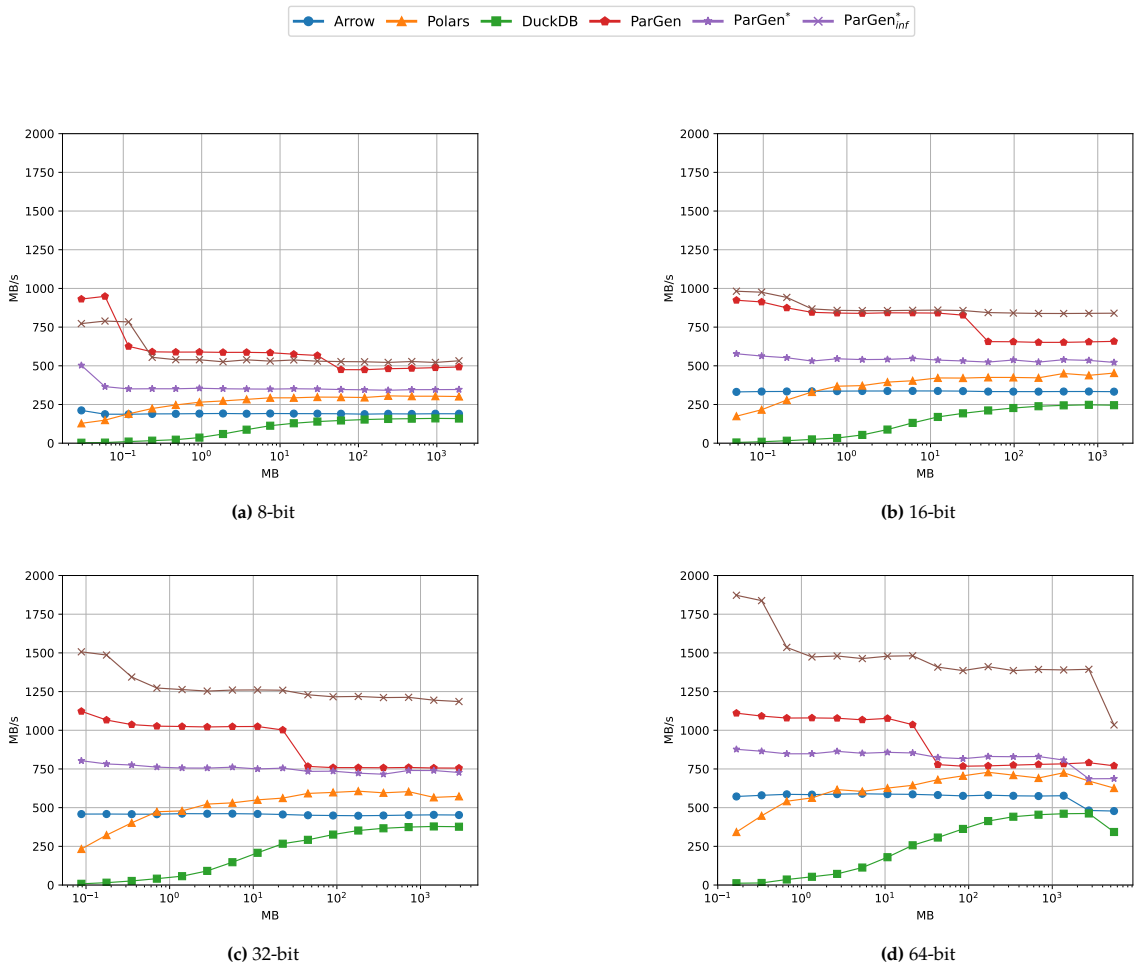
The following sections showcase the performance of the three state-of-the-art frameworks and three parser generator implementations per type. The three parser generator implementations are ParGen, ParGen* and ParGen$^*_{inf}$. ParGen is the first attempt at a parser generator. It expects the input data to be completely in memory. The second and third parser generators are denoted by P^*, which indicates that the parser reads the input data using a buffer. Hence, consuming the bytes from the input in increments. Additionally, ParGen$^*_{inf}$ performs size inference on the schema to possibly add additional optimizations.

5.4.1. Unsigned integers

The unsigned integer type has four variants, 8-bit, 16-bit, 32-bit, and 64-bit. Figure 5.8 shows the results for parsing the respective homogeneous datasets as described in Section 5.1.3. The results show several characteristics. First, the performance of each parser seems to scale with the size of the unsigned integer. Increasing the size of an integer typically means that each integer has more digits or bytes that are used to represent them in text. Consequently, the percentage of control characters, such as the field delimiter, might become lower because the percentage of value characters increases. Second, the shape of the performance curve for each parser seems to remain similar when the size of the unsigned integer increases. This shows that the performance of the parsers depends on the input size of the CSV file. Finally, it shows that a parser generated ahead-of-time performs better over the other frameworks. The unbuffered reader performs better than the buffered reader for smaller input datasets but drops to the same level at about 60 MB. This is more or less the size of the L3 cache, possibly indicating a performance benefit due to cache for smaller datasets. Furthermore, the parser generator achieves even more performance using reduced bounds checking, achieving more than 2 times the performance than that of the Apache Arrow implementation.

5.4.2. Signed integers

The signed integer type has four variants, 8-bit, 16-bit, 32-bit, and 64-bit. Figure 5.9 shows the results for parsing the respective homogeneous datasets as described in Section 5.1.3. The results show similar characteristics as for the unsigned integer (i.e. the same performance characteristics over the size of the dataset and the size of type). For signed integers, the parser generator also outperforms the other



(a) 8-bit

(b) 16-bit

(c) 32-bit

(d) 64-bit

Figure 5.8: The throughput for parsing variable sized CSV files, containing only unsigned integers, using different frameworks.

frameworks. Nevertheless, reducing the size bounds did not achieve the same performance gain as unsigned integers. The reason for this is that signed integers require more control logic due to dealing with the sign. Additionally, the unbuffered reader shows the same characteristics as unsigned integers, where performance drops at an input size of approximately 60 MB.

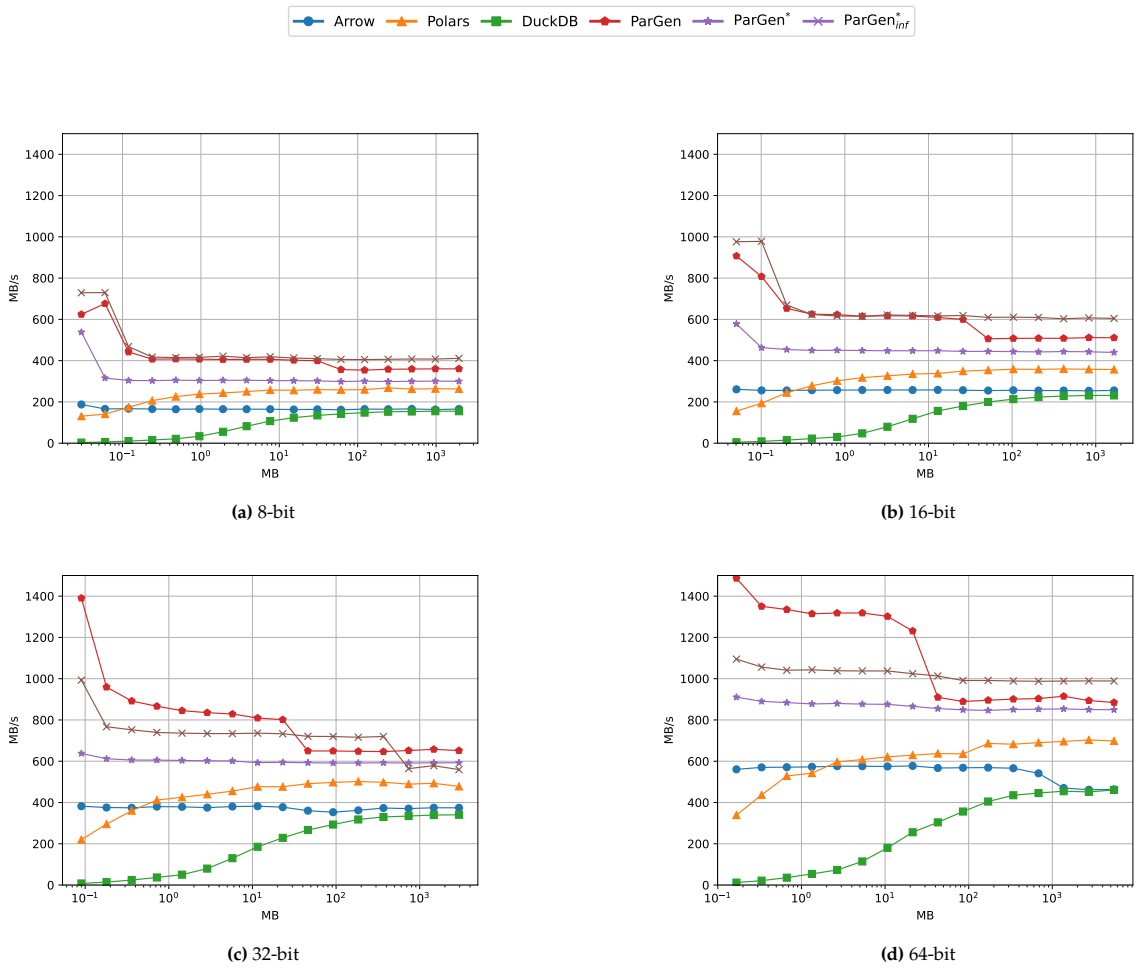
5.4.3. Floating point numbers

The floating point number type has two variants, 32-bit, and 64-bit. Figure 5.10 shows the results for parsing the respective homogeneous datasets as described in Section 5.1.3. The results show similar characteristics as for the signed integer. Furthermore, the generated buffered reader achieves twice the throughput of the Apache Arrow reader, despite using the same `lexical_core` [22] parser for floating-point numbers. Additionally, the unbuffered reader shows the same characteristics as both integer types, where performance drops at an input size of approximately 60 MB.

5.4.4. Strings

The string type has four variants that can be split in two groups, one with a small string size and one with a larger string size. Each group contains two variants, escaped or unescaped strings.

The performance results for the small unescaped and escaped strings can be found in Figure 5.11. It shows that Apache Arrow achieves the highest throughput for both escaped and unescaped strings. This is because the Apache Arrow implementation is specialized in parsing strings. As discussed in Section 2.1.1, Apache Arrow parses CSV directly into string records. This means that when the type of a field is a string, no further processing is needed. Consequently, only efficient `rust-csv` [8] parsing



(a) 8-bit

(b) 16-bit

(c) 32-bit

(d) 64-bit

Figure 5.9: The throughput for parsing variable sized CSV files, containing only signed integers, using different frameworks

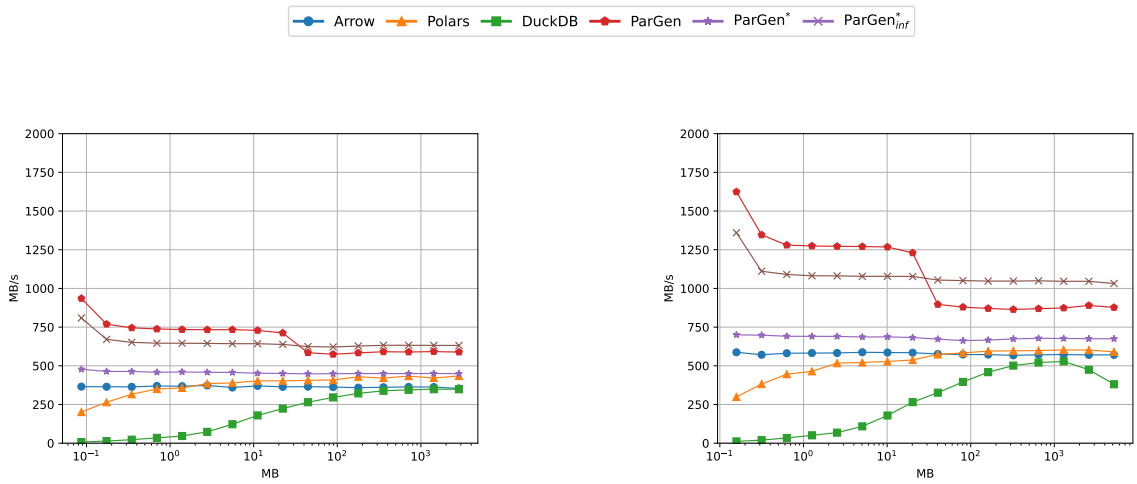
has to be performed. The generated buffered reader performs slightly worse than Apache Arrow. It uses the same technique as `rust-csv` but might differ in small control logic details. Furthermore, the generated unbuffered reader performs almost twice as slow as Apache Arrow.

The performance results for the large unescaped and escaped strings can be found in Figure 5.12. It shows that the generated buffered reader achieves the highest throughput for both escaped and unescaped strings. Apache Arrow performs slightly worse, which might be due to small differences in the implementation of the control logic of the generated reader. Furthermore, the generated unbuffered reader still performs almost twice as slow as Apache Arrow. However, for larger strings there seems to be a higher throughput for datasets of approximately 60 MB.

Finally, it is worth noting that the results show that unescaped strings are easier to parse than escaped strings. For both Apache Arrow and the generated parsers this difference is caused by using a shortcut. In the case of an unescaped field, no escape characters have to be removed from the source string during parsing. Consequently, the data from the source string can be copied in larger chunks, reducing the number of memory stores.

5.5. TPC-H results

The results for parsing the TPC-H datasets in the CSV format can be found in Figure 5.13. It shows that the generated buffered reader performs best in all cases, except the nations dataset. Given the byte distributions for types in Figure 5.3, the generated buffered reader seems to perform slightly better for datasets with a higher number presence. As seen in the synthetic data results, it performs the best



(a) 32-bit

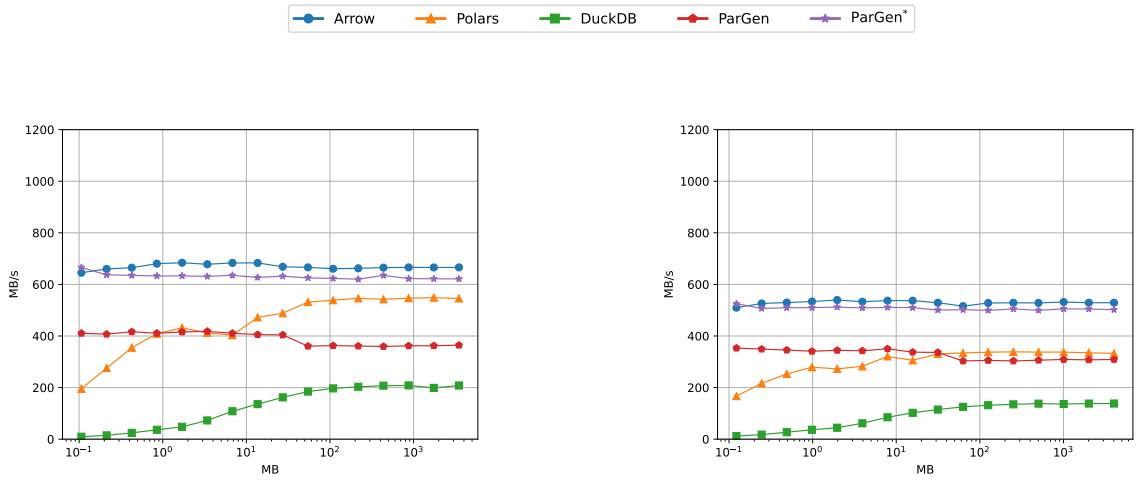
(b) 64-bit

Figure 5.10: The throughput for parsing variable sized CSV files, containing only floating-point numbers, using different frameworks.

for the number types, which explains the slightly better results in this case. Nevertheless, most of the data are strings hence the small differences in throughput with Apache Arrow. This is similar to the synthetic data results for strings.

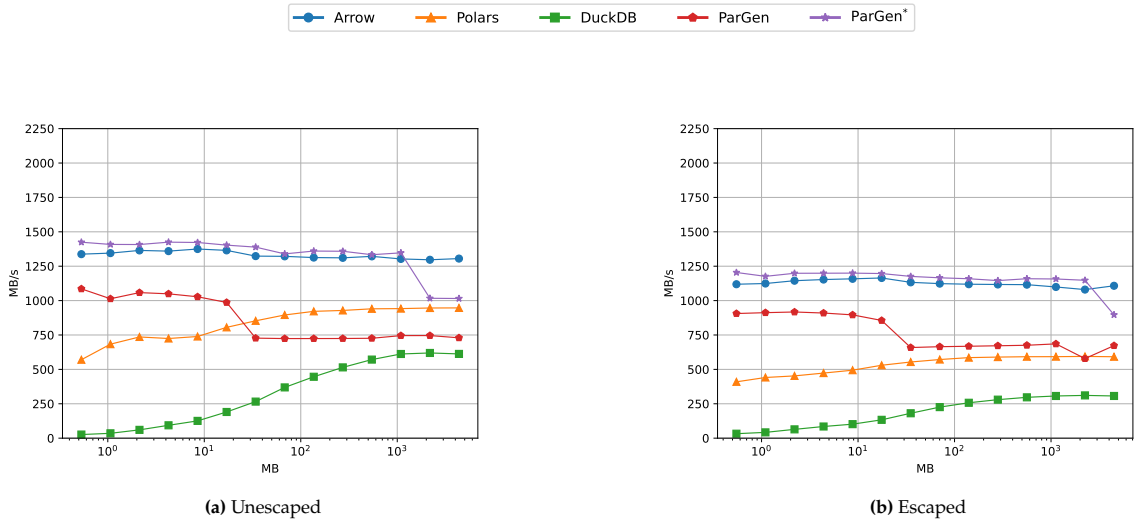
5.6. TPC-DS results

The results for parsing the TPC-DS datasets in the CSV format can be found in Figure 5.14. It shows that the generated buffered reader performs best compared to external frameworks in every case. However, the generated unbuffered reader performs better for files larger than a 100 MB. Similarly, Polars performs better to Arrow for these datasets. Given the byte distributions of types in Figure 5.4, the generated buffered reader seems to perform slightly better for datasets with a higher number presence. This is similar to the TPC-H datasets.



(a) Unescaped **(b)** Escaped

Figure 5.11: The throughput for parsing CSV files containing only small, escaped or unescaped, strings



(a) Unescaped **(b)** Escaped

Figure 5.12: The throughput for parsing CSV files containing only large, escaped or unescaped, strings.

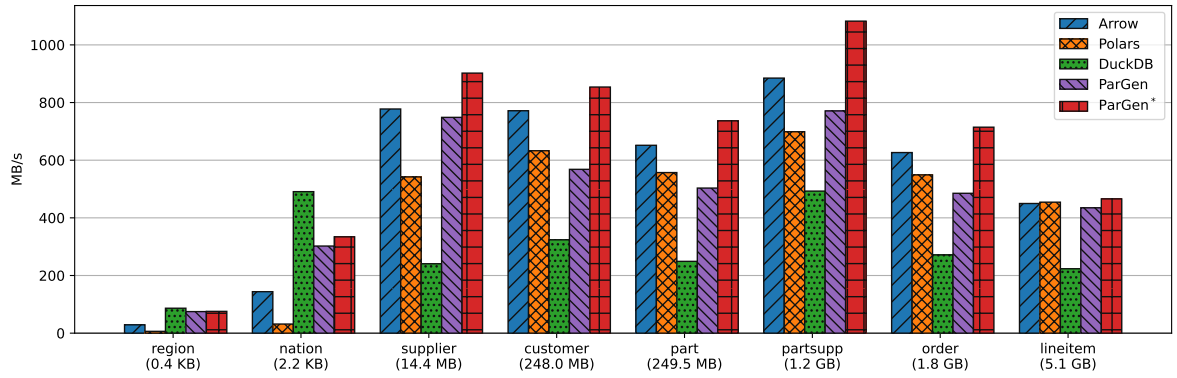


Figure 5.13: The throughput for parsing different TPC-H datasets in the CSV format using different parsers.

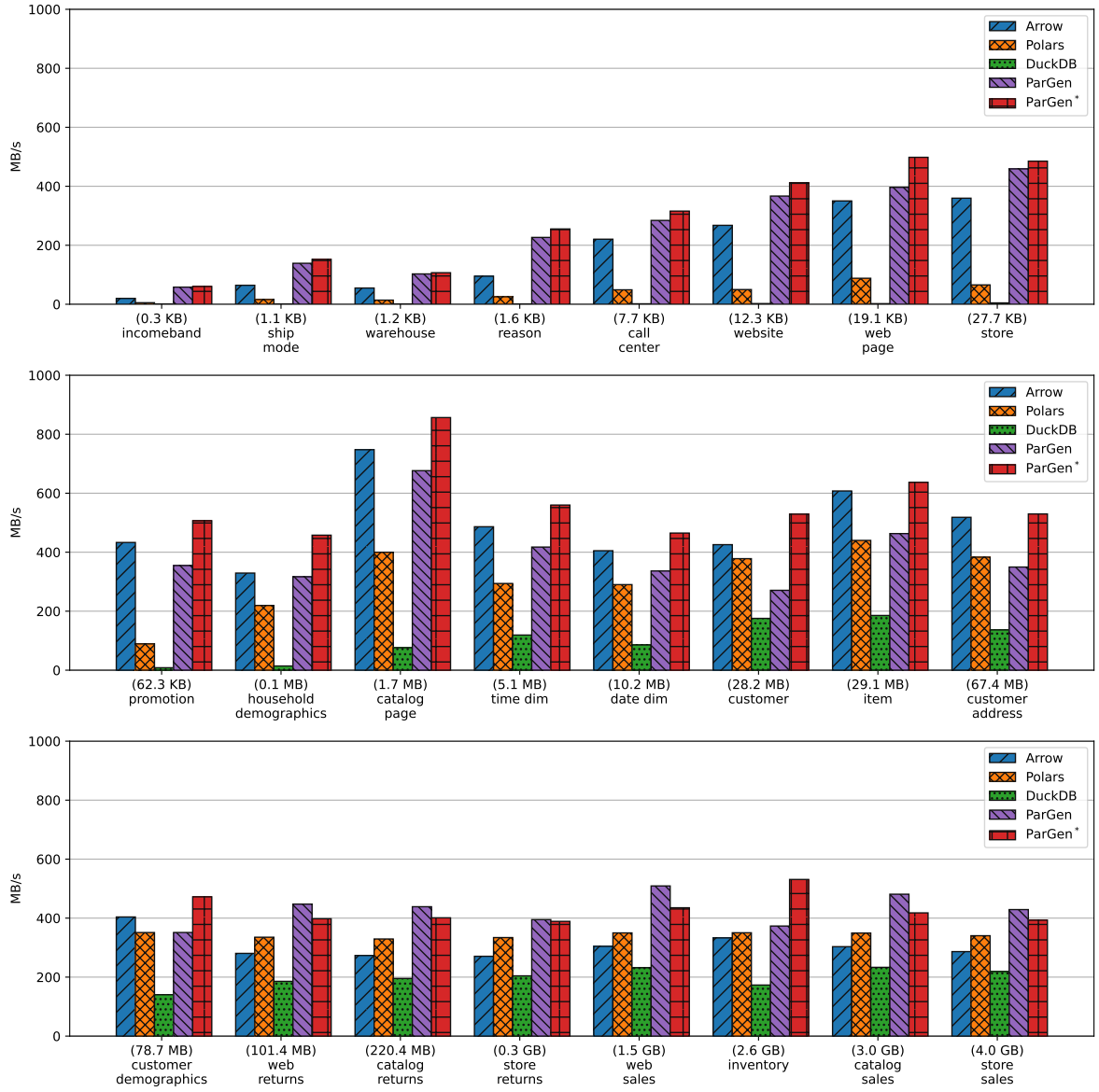


Figure 5.14: The throughput for parsing different TPC-DS datasets in the CSV format using different parsers.

6

Conclusion and future work

This research explored the use of parser generation to improve the throughput of parsing CSV to Arrow. The code developed for this work is available on GitHub [37].

6.1. Conclusion

In order to answer the research questions, this work produced several products. First, an ahead-of-time parser generator was implemented by creating a Rust procedural derive macro library. It supports both unbuffered and buffered reading of CSV data sources. Moreover, a size bound feature was developed to reduce control overhead in buffered reading by calculating the worst-case byte representation of a record at compile-time. Second, a just-in-time parser generator was implemented by creating a library that constructs a parser in-memory. However, when comparing it with the ahead-of-time parser generator, it showed a slightly worse performance due to run-time overhead (see Section 5.2).

The ahead-of-time parser generator was compared to three open-source state-of-the-art frameworks using different benchmarks. These frameworks are Apache Arrow [40], Polars [25] and DuckDB [5]. For all frameworks and the parser generator, two types of benchmarks were run.

First, a synthetic benchmark where the throughput of parsing variable sized datasets with only the integer, floating point or string type were measured. The ahead-of-time parser generator performed significantly better for the numeric types compared to the other frameworks. For the signed and unsigned integer types it achieved almost a 1.5x increased throughput compared to Apache Arrow. Furthermore, using size bounds this increased to almost a 2x increase. For floating point numbers, a generated buffered reader performed slightly better than Apache Arrow. Alternatively, a generated unbuffered reader or a buffered reader using size bounds were each able to achieve a throughput increase of approximately 2x compared to Apache Arrow. Finally, for both the escaped and unescaped string types an ahead-of-time generated parser achieved the same performance as Apache Arrow. This is because they share the same performant string parser implementation. Nevertheless, the generated unbuffered reader did show a 1.5x increase in throughput for datasets smaller than the L3 cache size (60 MB).

Second, datasets from the TPC-H and TPC-DS benchmarks were used to measure the performance of parsers for real-world datasets. For both TPC-H and TPC-DS, a generated buffered parser performed better in almost every dataset compared to state-of-the-art frameworks. However, the unbuffered reader performed better for TPC-DS datasets with a size larger than a 100 MB. For datasets having a higher presence of integers in terms of bytes, the parser seemed to perform slightly better. Similar to the results from the synthetic datasets. Nevertheless, most datasets contained a high presence of string types for which the generated parser has a similar performance to that of Apache Arrow.

Additionally, several experiments were performed to identify enhancements for CSV parsing. Multi-threading was analyzed using scanner and worker threads. Section 5.3 revealed that the scanning of CSV limits the throughput of a worker thread. Furthermore, state-of-the-art frameworks such as Polars

and DuckDB show similar throughput characteristics, with a maximum throughput that is not close to the throughput that worker threads could reach. A solution to this is SIMD, for which a record scanner could reach a throughput of up to 2.8 GB/s as opposed to the 1.25 GB/s maximum multi-threaded throughput reached by Polars.

In conclusion, a parser generator is shown to perform better than state-of-the-art frameworks for both synthetic and real-world datasets. Additionally, a parser generator is able to leverage compile-time context, such as the schema, to reduce parsing overhead. In this work, this was done by calculating the maximum byte representation of a schema and reducing bound checks on input data. Furthermore, by implementing the parser generator using a derive macro it is easy to integrate and requires few changes to update a schema.

6.2. Future work

In this work, the core principle of parser generation was explored, leaving some topics not integrated or left open. These features could be explored to extend the functionality of a parser generator and possibly improve its performance.

6.2.1. Extending size bounds

The results showed that analyzing size bounds can improve the performance of parsing CSV. This analysis only considered the maximum size of a record to determine the lower bound number of records present in CSV data. Nevertheless, it is also possible to infer a minimum record size as types often have a lower bound. In future work, this minimum size could have several applications. First, it can be used to upper bound the number of records in CSV data. Second, when the minimum and maximum size of a datatype is known and they turn out to be the same, data could be handled more efficiently. For instance, fixed sized GUID strings could be directly copied into a string buffer rather than determining where the string ends. This results in efficient memory copies.

6.2.2. Applying SIMD and multi-threading

Whilst SIMD has been explored as a method to accelerate CSV parsing, it required a different approach to generating the parser. SIMD could make multi-threaded CSV parsing faster as seen in Section 5.3. It can lift the scanning bottleneck in CSV reading, to allow a higher throughput in parsing parallelization. Furthermore, knowing the schema of a dataset ahead-of-time, SIMD might be able to be used for type specific parsing (i.e. SIMD instructions are inlined for types that support SIMD in parsing).

References

[1] U.S. General Services Administration. *Data Catalog*. URL: https://catalog.data.gov/dataset/ (visited on 10/29/2025).

[2] Tim Anema et al. "High Throughput GPU-Accelerated FSST String Compression". In: *Proceedings of the VLDB Endowment. ISSN* 2150 (2025), pp. 80–97.

[3] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. RFC 8259. Dec. 2017. DOI: 10.17487/RFC8259. URL: https://www.rfc-editor.org/info/rfc8259.

[4] Jianyu Chen, Maurice Daverveldt, and Zaid Al-Ars. "FPGA Acceleration of Zstd Compression Algorithm". In: *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2021, pp. 188–191. DOI: 10.1109/IPDPSW52791.2021.00035.

[5] *DuckDB – An in-process SQL OLAP database management system*. DuckDB Foundation. URL: https://duckdb.org/.

[6] Publications Office of the European Union. *European Data*. URL: https://data.europa.eu/data/datasets (visited on 10/29/2025).

[7] DuckDB Foundation. *Core Extensions*. 2025. URL: https://duckdb.org/docs/stable/core_extensions/overview.

[8] Andrew Gallant. *csv*. URL: https://github.com/BurntSushi/rust-csv.

[9] *getrandom: system's random number generator*. rust-random. URL: https://github.com/rust-random/getrandom.

[10] Joel de Guzman and Hartmut Kaise. *Spirit X3 3.10*. Boost C++ libraries. URL: https://www.boost.org/doc/libs/latest/libs/spirit/doc/x3/html/index.html.

[11] Alexander Huszagh. *fast-float2*. 2025. URL: https://github.com/Alexhuszagh/fast-float-rust.

[12] "IEEE Standard for Floating-Point Arithmetic". In: *IEEE Std 754-2019 (Revision of IEEE 754-2008)* (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[13] "IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7". In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (2018), pp. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[14] *Intel® Intrinsics Guide*. Version 3.6.9. Intel Corporation, Dec. 2024. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[15] Simonas Kazlauskas. *libloading*. Version 0.9.0. 2025. URL: https://github.com/nagisa/rust_libloading.

[16] Markus Klein. *atoi-rs*. 2025. URL: https://github.com/pacman82/atoi-rs.

[17] Alexey Kopytov. *sysbench*. 2026. URL: https://github.com/akopytov/sysbench.

[18] Geoff Langdale and Daniel Lemire. "Parsing gigabytes of JSON per second". In: *The VLDB Journal* 28.6 (Oct. 2019), pp. 941–960. DOI: 10.1007/s00778-019-00578-5.

[19] Geoff Langdale and Daniel Lemire. *simdcsv*. URL: https://github.com/geofflangdale/simdcsv.

[20] Daniel Lemire. "Number parsing at a gigabyte per second". In: *Software: Practice and Experience* 51.8 (2021), pp. 1700–1727. DOI: https://doi.org/10.1002/spe.2984. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2984.

[21] *nom, eating data byte by byte*. rust-bakery. URL: https://github.com/rust-bakery/nom.

[22] Alexander Payne and Alexander Huszagh. *lexical*. 2025. URL: https://github.com/Alexhuszagh/rust-lexical.

[23] Johan Peltenburg et al. “Battling the CPU Bottleneck in Apache Parquet to Arrow Conversion Using FPGA”. In: *2020 International Conference on Field-Programmable Technology (ICFPT)*. 2020, pp. 281–286. DOI: 10.1109/ICFPT51103.2020.00048.

[24] Johan Peltenburg et al. “Tens of gigabytes per second JSON-to-Arrow conversion with FPGA accelerators”. In: *2021 International Conference on Field-Programmable Technology (ICFPT)*. 2021, pp. 1–9. DOI: 10.1109/ICFPT52863.2021.9609833.

[25] *Polars — DataFrames for the new era*. Polars. URL: https://pola.rs/.

[26] Rust Project. *Chapter 15 - Linkage*. Accessed: 2025-12-15, Chapter 15 - Linkage. The Rust Reference, 2025. URL: https://doc.rust-lang.org/reference/linkage.html.

[27] Rust Project. *Procedural Macros*. Accessed: 2025-12-15, Chapter 3 - Macros. The Rust Reference, 2025. URL: https://doc.rust-lang.org/reference/procedural-macros.html.

[28] Rust Project. *Traits*. Accessed: 2026-01-16, Chapter 6 - Items. The Rust Reference, 2026. URL: https://doc.rust-lang.org/reference/items/traits.html.

[29] Rust Project. *Using Trait Objects to Abstract over Shared Behavior*. Accessed: 2026-01-16, Chapter 6 - Items. The Rust Programming Language, 2026. URL: https://doc.rust-lang.org/stable/book/ch18-02-trait-objects.html.

[30] *Rand*. rust-random. URL: https://github.com/rust-random/rand.

[31] Dmitry Rodionov et al. *Rust fast &[u8] to integer parser*. 2025. URL: https://github.com/RoDmitry/atoi_simd.

[32] Yakov Shafranovich. *Common Format and MIME Type for Comma-Separated Values (CSV) Files*. RFC 4180. Oct. 2005. DOI: 10.17487/RFC4180. URL: https://www.rfc-editor.org/info/rfc4180.

[33] Michael Sipser. “Regular Languages”. In: *Introduction to the Theory of Computation*. 3rd ed. CENGAGE Learning, 2013, pp. 31–44.

[34] Sam Streef. *Ahead-of-time CSV to Arrow reader*. Accessed: 2026-01-16. 2026. URL: https://github.com/sstreef/csv-to-arrow/tree/main/crates/arrow-csv-reader.

[35] Sam Streef. *CSV Dataset Generator*. Accessed: 2026-01-16. 2026. URL: https://github.com/sstreef/csv-to-arrow/tree/main/crates/csv-generator.

[36] Sam Streef. *CSV to Arrow reader derive macro*. Accessed: 2026-01-16. 2026. URL: https://github.com/sstreef/csv-to-arrow/tree/main/crates/arrow-csv-reader-derive.

[37] Sam Streef. *CSV-to-Arrow*. Accessed: 2026-01-16. 2026. URL: https://github.com/sstreef/csv-to-arrow.

[38] Sam Streef. *Just-in-time CSV to Arrow reader*. Accessed: 2026-01-16. 2026. URL: https://github.com/sstreef/csv-to-arrow/tree/main/crates/arrow-csv-reader-jit.

[39] Sam Streef. *SIMD CSV parsing*. Accessed: 2026-01-16. 2026. URL: https://github.com/sstreef/csv-to-arrow/tree/main/experiments/simd.

[40] The Apache Software Foundation. *Apache Arrow*. 2025. URL: https://arrow.apache.org.

[41] The Apache Software Foundation. *Implementations*. 2025. URL: https://arrow.apache.org/docs/implementations.html.

[42] The Apache Software Foundation. *Native Rust implementation of Apache Arrow and Apache Parquet*. URL: https://github.com/apache/arrow-rs.

[43] The Apache Software Foundation. *The Arrow C data interface*. 2025. URL: https://arrow.apache.org/docs/format/CDataInterface.html.

[44] David Tolnay. *Parser for Rust source code*. URL: https://github.com/dtolnay/syn.

[45] TPC. *TPC-DS*. URL: https://www.tpc.org/tpcds/.

[46] TPC. *TPC-H*. URL: https://www.tpc.org/tpch/.

[47] International Telecommunication Union. *Measuring Digital Development - Facts and figures 2024*. Tech. rep. 2024. URL: https://www.itu.int/hub/publication/D-IND-ICT_MDD-2024-4/.

[48] Robin Vonk, Joost Hoozemans, and Zaid Al-Ars. “GSST: Parallel string decompression at 191 GB/s on GPU”. In: *Proceedings of the 5th Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems*. 2025, pp. 8–14.