Speculative Distributed CSV Data Parsing for Big Data Analytics

Chang Ge[§*] Yinan Li[†] Eric Eilebrecht[†] Badrish Chandramouli[†] Donald Kossmann[†]

[†]Microsoft Research [§]University of Waterloo
c4ge@uwaterloo.ca, {yinan.li, eric.eilebrecht, badrishc, donaldk}@microsoft.com

ABSTRACT

There has been a recent flurry of interest in providing query capability on raw data in today's big data systems. These raw data must be parsed before processing or use in analytics. Thus, a fundamental challenge in distributed big data systems is that of efficient parallel parsing of raw data. The difficulties come from the inherent ambiguity while independently parsing chunks of raw data without knowing the context of these chunks. Specifically, it can be difficult to find the beginnings and ends of fields and records in these chunks of raw data. To parallelize parsing, this paper proposes a speculation-based approach for the CSV format, arguably the most commonly used raw data format. Due to the syntactic and statistical properties of the format, speculative parsing rarely fails and therefore parsing is efficiently parallelized in a distributed setting. Our speculative approach is also robust, meaning that it can reliably detect syntax errors in CSV data. We experimentally evaluate the speculative, distributed parsing approach in Apache Spark using more than 11,000 real-world datasets, and show that our parser produces significant performance benefits over existing methods.

ACM Reference Format:
Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative Distributed CSV Data Parsing for Big Data Analytics. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3299869.3319898

*Work performed during internship at Microsoft Research.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319898

1 INTRODUCTION

The world is generating an ever growing amount of data. A great deal of the data is never used. It would be prohibitively expensive to cook and load all these data into database systems. Instead, these data are simply stored in file systems in raw format and remain available on demand. If needed, however, the data must be parsed before processing. This way, parsing is on the critical path of many data processing systems and, in particular, big data processing systems. Prominent examples of cloud services that provide SQL query capability on raw, unparsed data are Amazon Athena [1], Amazon Redshift Spectrum [2], and Google BigQuery [4, 21]. In the research community, NoDB [8, 17] is an example system that enables query capability on raw data.

To be effective, big data systems work in clusters of machines and execute tasks in parallel on these machines. As parsing is on the critical path and an expensive task, it is crucial to parallelize parsing, too. Otherwise, the benefits of parallelizing other tasks are diminished according to Amdahl's law. Unfortunately, parallel and distributed parsing is difficult [12]. The problem is that files can be split arbitrarily into chunks without alignment of record or field boundaries in modern distributed file systems such as HDFS, Amazon S3 or Microsoft Azure Blob Store. In fact, these storage systems might not even be aware of what kind of data they are storing and how to best align chunk boundaries to fields and records. The following example illustrates the challenge of parallel parsing with arbitrary chunk boundaries.

```
Ambiguous CSV Chunk:  lice,",",16\nBob,",",17
   Interpretation 1:  lice,",",16\nBob,",",17
   Interpretation 2:  lice,",",16\nBob,",",17
```

Figure 1: Example ambiguous CSV chunk (quoted fields are marked gray)

1.1 Example

Figure 1 shows a snippet of a real-world CSV dataset (Section 7). This snippet could be the beginning of a chunk stored on a machine of an HDFS cluster. This example shows the ambiguity that a parser faces when it needs to parse this

chunk of data out of context. Given the semantics of the CSV data format (see Section 2), there are two possible interpretations. Interpretation 1 considers "lice" as the end of a field and then "," as the next field, followed by a nummeric field and interprets the *newline* as the end of record. In contrast, Interpretation 2 considers "lice," as the end of a field and then the *newline* is part of the next field and does not signal a new record. Obviously, these two interpretations are totally different and would result in drastically different query results if further processed by a big data system such as Apache Spark [11, 26]. According to the CSV specification, both interpretations are possible. A traditional CSV parser would need to get more context (i.e., the result of parsing the previous chunk) to disambiguate and could not process this chunk of data in parallel.

Interestingly, this example also guides the path to build an efficient speculative parser. First, even though both interpretations are possible according to the CSV specification (see Section 2), the first interpretation is much more likely: the string "`,16\nBob,`" is rare as a field value whereas numeric fields and the string "Bob" look like "normal" field values. Second, even though handling ambiguity is indispensable, it turns out that such ambiguous chunks of data are not very common in practice. As a result, we can skip the speculation process for unambiguous chunks to minimize the overhead associated with speculation, if there exists a way to quickly detect such chunks. In sum, even though the CSV specifications give room for many forms of ambiguity, there is no reason to give up on parallel parsing.

1.2 Contributions

This paper presents a parallel CSV parser and shows how it can be integrated into Spark. It gives the results of comprehensive performance experiments on real-world datasets.

The main contribution is a novel speculative method to determine field and record boundaries in arbitrary chunks of CSV data. Our parser parses CSV data in almost all real-world cases in a single pass and in parallel. As a baseline, this paper also describes an altnerative conservative, non-speculative approach for parallel parsing. Such a conservative approach requires two passes over the CSV data to handle *bad cases* such as those of Figure 1. The first pass determines field and record boundaries and thus provides context for the second pass in which the tokenization happens. While both passes can be carried out in parallel, the additional pass is wasteful in the light that our speculative approach almost always guesses correctly and simply degenerates to the two-pass approach if it mispredicts.

One important feature of our speculative method is that it detects ill-formed CSV data. Obviously, error detection is extremely important in practice and non trivial in concert with speculation. Another important feature is that our speculative method can be integrated with any state-of-the-art parsers. Recently, there have been breakthroughs in the development of parsers for semi-structured data [19, 24] and it is important that our approach composes well with those parsers. Specifically, we used the Mison parser [19] for our performance experiments.

While this paper focuses on CSV data, the principles of speculation for parallel parsing of semi-structured data are general. We are currently working on extending the framework for other plain-text formats such as JSON and XML.

We present the results of extensive performance experiments on more than 11,000 real-world CSV datasets, using Apache Spark [11, 26] as the query engine. We show that a speculative parallel parser is up to 2.4X faster than a conservative, two-pass parallel parser. The speculative approach mispredicts in less than once in ten million chunks. In fact, we show that our parallel parser performs almost as well as an *ideal* parallel parser which has a perfect oracle to determine record boundaries. In other words, our speculative method to predict record boundaries (and recover from mispredictions) has negligible overhead.

The remainder of this paper is structured as follows: Section 2 introduces background on CSV format. Section 3 presents the distributed parsing approaches. Sections 4 gives details on speculative parsing on CSV chunks. Syntax error handling is covered in Section 5. Section 6 presents our implementation of the proposed approaches in Apache Spark. Section 7 describes our experimental results. Section 8 covers related work. Section 9 contains concluding remarks.

2 BACKGROUND ON CSV FORMAT

CSV (Comma-Separated Values) is a lightweight, plain-text, tabular data format, and is the the most commonly used data format in many applications ranging from data analytics to machine learning. According to the standard RFC-4180 [25], the CSV format can be recursively defined as follows (we omit the formal definition of Header in the interest of space):

$$\begin{aligned}
\textsc{File} &= [\textsc{Header}\ \boxed{\texttt{\textbackslash n}}\ ]\ \textsc{Record}\ \boxed{\texttt{\textbackslash n}}\ \ldots\ \boxed{\texttt{\textbackslash n}}\ \textsc{Record} \\
\textsc{Record} &= \textsc{Field}\ \boxed{\texttt{,}}\ \ldots\ \boxed{\texttt{,}}\ \textsc{Field} \\
\textsc{Field} &= \textsc{Quoted} \mid \textsc{Unquoted} \\
\textsc{Quoted} &= \boxed{\texttt{"}}\ \textsc{Escaped}\ldots\textsc{Escaped}\ \boxed{\texttt{"}} \\
\textsc{Escaped} &= \textsc{Char} \mid \boxed{\texttt{"}}\,\boxed{\texttt{"}} \mid \boxed{\texttt{\textbackslash n}} \mid \boxed{\texttt{,}} \\
\textsc{Unquoted} &= \textsc{Char}\ldots\textsc{Char} \\
\textsc{Char} &= \texttt{Any char except of}\ \boxed{\texttt{"}}\texttt{,}\ \boxed{\texttt{\textbackslash n}}\texttt{,\ and}\ \boxed{\texttt{,}}
\end{aligned}$$

A CSV *file* contains an optional *header* record that includes names corresponding to the fields in this file, followed by a sequence of zero or more records, separated by newlines

("\n"). A *record* is a sequence of one or more fields, separated by commas (","). Each *field* is either *quoted* or *unquoted*. A field containing quotes (""), commas, or newlines must be quoted, which is represented as quotes surrounding a sequence of zero or more (escaped) characters. An embedded quote within a quoted field must be escaped by preceding it with another quote. Whitespaces are considered as a part of a field, and should never be ignored.

Despite its popularity, the CSV format has never been officially standardized. RFC-4180 [25] is largely considered to be the main reference for CSV parsers, but it is not an official specification. Some variations on the CSV format use another delimiter (e.g., tab or space) to separate fields, or use an alternative character (e.g., backslash) to escaped a quote inside a quoted field. In this paper, we assume that all CSV files are in the format defined by RFC-4180. However, the proposed approaches can be extended to support other CSV variations such as TSV (Tab Separated Values).

3 DISTRIBUTED PARSING

This section provides an overview of distributed parsing on CSV data. We assume that the CSV input is well-formed, i.e., there is no syntax error. We relax this assumption and extend our approaches to handle ill-formed data in Section 5.

3.1 Framework

The distributed parsing framework takes as input a list of *chunks* that are split from a CSV input, and a CSV *parser* that is capable of processing a sequence of complete records. For each chunk, the framework assigns a worker, which is responsible for locating the first record delimiter in the chunk, as well as the first record delimiter after the end of the chunk. The region between the two delimiters is called *adjusted chunk*. If a chunk does not contain any record delimiters, its adjusted chunk is empty. As each adjusted chunk contains a sequence of complete CSV records, the worker can invoke the input parser to process the adjusted chunk. It is straightforward to see that all adjusted chunks are connected end to end, and any character in the CSV input is contained in one and only one adjusted chunk. Thus, parsing is fully parallelized in a distributed fashion. Figure 2 illustrates a sequence of chunks and their adjusted chunks.

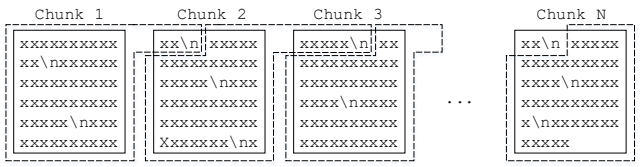


Figure 2: Chunks and adjusted chunks (solid boxes are chunks, dashed boxes are adjusted chunks)

This framework is actually far from new – in fact, it has been widely used in the Hadoop ecosystem. For example, Spark [11] uses this framework to process data in a simplified variation of the CSV format, where newlines are not allowed within quoted fields. For each chunk, Spark scans the chunk to find the first newline within the chunk and the first newline appearing after the end of the chunk, and processes the region between the two newlines. This approach, unfortunately, is not applicable to process standard CSV data because in general, a newline could be an escaped newline inside a quoted field rather than a record delimiter. In the rest of this section, we present a two-pass approach (Section 3.2) and a speculative approach (Section 3.3) to support standard CSV dataset. Both approaches follow this framework.

This framework assumes that a worker can access data in the subsequent chunks. This assumption holds for a majority of usage scenarios (e.g., HDFS), but may not apply to certain usage cases. In Appendix B.2, we relax this assumption and extend our approaches to handle records at chunk boundaries without accessing data in other chunks.

We note that the framework is designed for scenarios where chunks are generally much larger than records, and thus aims to achieve coarse-grained parallelism. For records that are larger or close to the chunk size, the framework can still correctly process the input, but the overhead of handling records at chunk boundaries may increase significantly, and hinders the overall parsing speed. Nevertheless, this rarely occurs in big data systems where record sizes typically range from tens of bytes to thousands of bytes, while chunk sizes are typically at least several megabytes.

This framework decouples parsing from distributed processing, and composes well with any CSV parser. This decoupling enables us to exploit recent innovations in parsing algorithms such as Mison [19] and Sparser [24], to achieve the best per-chunk parsing performance.

3.2 Two-Pass Approach

We first present a non-speculative approach for distributed CSV parsing. This approach parses CSV data in a fully parallel way, but needs to scan the input twice. This approach is used as the fallback solution in our speculative approach to recover from mispredictions.

The basic idea behind this approach has already been implemented in the open-source project ParaText [5], which aims to parse CSV data in parallel on multi-core machines. Nevertheless, to the best of our knowledge, this approach has not been documented in the literature yet. We therefore describe this approach in the interest of completeness.

Following our framework (Section 3.1), this approach runs in two passes: the first pass identifies adjusted chunks; the second pass uses the CSV parser to process complete records

within each adjusted chunk. Unlike a more straightforward approach that sequentially scans over the CSV input to identify adjusted chunks, the two-pass approach determines the cutting points in a fully parallel fashion, by exploiting the simplicity of the CSV format.

More specifically, in the first pass, each worker scans the assigned chunk and collects three statistics: 1) the number of quotes within the chunk; 2) the position of the first newline after an even number of quotes within the chunk; 3) the position of the first newline after an odd number of quotes within the chunk. Once the scan is complete, each worker sends these statistics back to the master. The master then sequentially iterates over the statistics of all chunks and computes the starting positions of all adjusted chunks. For the k-th chunk, the master sums up the number of quotes in the first $k-1$ chunks. If the number is even, then the k-th chunk does not start in the middle of a quoted field, and the first newline after an even number of quotes (the second collected information) is the first record delimiter in this chunk. Otherwise, if the number is odd, the first newline after an odd number of quotes (the third collected information) is the first record delimiter. The end position of the adjusted chunk is obtained based on the starting position of the next adjusted chunk. Clearly, this approach follows our framework, but requires an addition pass of scan to locate adjusted chunks.

3.3 Speculative Approach

The two-pass approach fully parallelizes CSV parsing, but requires an additional pass of scan to identify record delimiters. In this section, we aim to remove this additional pass and process each chunk in one pass. As shown in Section 1.1, it is inherently impossible to fully determine starting parsing states for an arbitrary chunk. As a result, we design a speculation-based approach by exploiting the opportunities we discussed in Section 1.1.

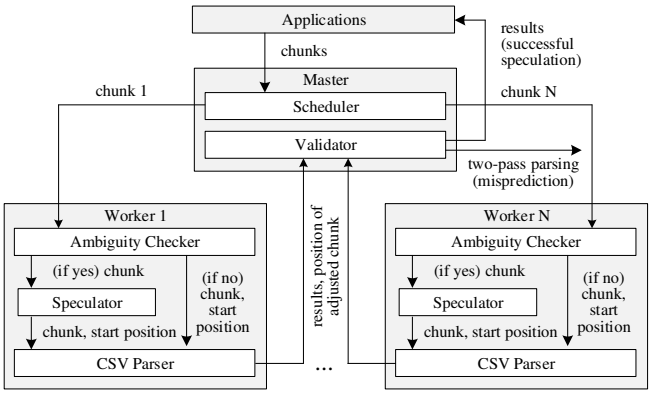


Figure 3: Architecture of speculative parsing

Figure 3 shows the architecture of the speculative approach. We focus on the master in this section and will present our parsing techniques for workers in Section 4.

The master assigns each chunk to a worker that can process the data in parallel. Each worker speculatively parses the chunk and returns the parsing results along with the position of the predicted adjusted chunk to the master. The master is responsible for validating the speculations made independently by all workers. A speculation on a chunk is considered to be successful if its adjusted chunk starts immediately after a record delimiter (or the beginning of the input) and ends at a record delimiter. Formally, we introduce Property 1 for validating speculations of all workers. The formal proof is provided in Appendix A.

Property 1. *A speculation on the k-th chunk is successful if the first k predicted adjusted chunks are connected end to end.*

Following Property 1, the master sequentially iterates over all positions of all predicted adjusted chunks in the order of chunk IDs, and examines if each predicted adjusted chunk is connected to its previous adjusted chunk end to end. If not, the speculation fails, and we fall back to the two-pass approach (Section 3.2) to recover from mispredictions. If all predicted adjusted chunks are connected, the speculative parsing completes successfully.

The speculative approach also follows our framework to handle records at chunk boundaries (Section 3.1). If all speculations are successful, each worker starts from the first record delimiter within the corresponding chunk, and ends with the first record delimiter in the subsequent chunks, which is fully in line with the requirements of the framework. If speculations fail, we discard all parsing results and fall back on two-pass parsing, which follows the framework as well.

The speculative approach runs in one pass if speculations on all chunks are successful, and in three passes (two of which are from the fallback reparsing) in the case of mispredictions. As we will show empirically using a large number of real-world datasets (Section 7.3), our method mispredicts less than once in ten million chunks. Thus, the speculative approach is nearly an one-pass approach from the statistical point of view.

4 SPECULATIVE PARSING ON CHUNKS

In this section, we present techniques for parsing individual chunks speculatively. Given an arbitrary chunk of a CSV file, our algorithms predict the starting parsing state without the context on previous chunks, and optimistically parse the chunk based on the predicted state.

Our parsing approach is in general independent to the character encoding of the CSV input. It relies on the decoder of the input to convert the encoded byte stream into (multi-byte) characters, and applies algorithms presented in this

section on decoded characters. However, for common encodings such as UTF-8 and UTF-16, our algorithms can process an encoded byte stream as a sequence of single-byte characters, without having to decode the multi-byte characters. This optimization is discussed in Appendix B.1.

4.1 Overview

We say that a chunk is *quoted* if the chunk starts in the middle of a quoted field, or is *unquoted* if not. The fundamental challenge for parsing individual chunks is to speculate whether a chunk is quoted or unquoted. Once this starting state is predicted, the adjusted chunk of this chunk will be determined, so is the way to parse. More specifically, if the chunk is speculated to be unquoted, the first record delimiter is the first newline after an even number of quotes within the chunk. Otherwise, if the chunk is predicted to be quoted, the predicted first record delimiter is the first newline after an odd number of quotes. Having the predicted first record delimiter as the starting position of the adjusted chunk, we then invoke the parser to optimistically scan and parse the chunk. When it reaches the end of the chunk, it continues processing by reading data from the subsequent chunks until it encounters a record delimiter, whose position is considered as the predicted end position of its adjusted chunk.

Therefore, in the remaining of this section, we focus on the problem of speculating whether or not a given chunk is quoted. First of all, we observed that in many cases a worker may be able to fully determine the starting state of a chunk, if the alternative starting state results in invalid parsing on the chunk. According to this observation, we define that a chunk is *ambiguous* if we cannot fully determine whether a chunk is quoted or not by only analyzing the data within the chunk. Conversely, a chunk is *unambiguous* if it is confirmed to be either quoted or unquoted.

In this section, we present two algorithms to determine the ambiguity of a chunk. The first algorithm, described in Section 4.3, is a more straightforward algorithm but is computationally expensive. Nevertheless, it lays the theoretical foundation upon which we develop the second algorithm. Section 4.4 presents the second algorithm that produces the same results as the first algorithm but in a far more efficient way. If a chunk is unambiguous, its starting state is determined directly and the speculation is unnecessary. For a chunk that is inherently ambiguous, its starting state has to be determined speculatively. In Section 4.5, we present an algorithm to make the speculation decision based on the insights we mentioned in Section 1.1.

Either the algorithm to examine the ambiguity, or the algorithm to speculate the starting state requires access to the data in the chunk. These speculation-related processes introduce an additional pass of scan before parsing, making the speculative approach downgrade to a two-pass approach. However, the speculative property gives us the flexibility to trade-off speculation accuracy for processing speed. This enables us to perform our algorithms on only a (small) portion of the chunk, usually a fixed-size (e.g., 64KB) prefix of the data in the chunk. In this case, the ambiguity inferred from the portion of the chunk may have false positives on the ambiguity of the whole chunk, but definitely no false negatives – in other words, the whole chunk is either definitely unambiguous or possibly ambiguous. These false positives may introduce overhead in initiating the subsequent speculation algorithm and the possible reparsing when mispredict happens. However, our statistical results on real-world datasets show that false positives introduced by this optimization are extremely rare in practice (Section 7.3). In the rest of this section, the term "chunk" refers to an arbitrary fragment of a file, which could be a whole chunk, or a prefix of the chunk.

4.2 Finite-State Machine for CSV

We first introduce the Finite-State Machine (FSM) for the CSV format, a foundation upon which we design and analyze algorithms throughout this paper. The CSV format is defined in a regular language (see Section 2), which has an equivalent FSM and vice versa. As a result, a string follows the CSV specification if and only if the string is accepted by the equivalent FSM.

The FSM for CSV data has five possible states: Record start (`R`), Field start (`F`), Unquoted field (`U`), Quoted field (`Q`), and quoted field End (`E`). Among these states, `R` is the start state, while the final states include `R`, `F`, `U`, and `E`. There are four possible types of input characters: `quote`, `comma`, `newline`, and `other`, i.e., any other character except the three structural characters. Figure 4 shows the state transition function of the FSM. For each possible state, the table shows the output states resulting from each input. Note that not all input characters are allowed at certain states. For instance, if the current state is `U` and the next character is `quote`, the transition is invalid (marked "`-`" in Figure 4), as a field value containing quotes must be quoted. Similarly, `other` is not allowed at the state `E`, because at this state the previous character could be either a closing quote that must be followed by a structure character (i.e., comma or newline), or an escaping quote that must be followed by an escaped quote.

	quote	comma	newline	other
R (Record start)	Q	F	R	U
F (Field start)	Q	F	R	U
U (Unquoted field)	-	F	R	U
Q (Quoted field)	E	Q	Q	Q
E (quoted End)	Q	F	R	-

Figure 4: State transition table of FSM for CSV format

The five states can be categorized into two classes: the R, F, U, and E states are called *unquoted* states as the character at these states are always outside of quoted fields; the Q state is called *quoted* state because characters at this state are embedded in a pair of quotes. Interestingly, as can be observed from Figure 4, states in the same category always produce the same output state (ignoring invalid transitions), whereas states in different categories never transit into the same output state for any input character.

4.3 Determining Ambiguity using a FSM

In this section, we present an algorithm to examine whether a given CSV chunk is ambiguous, based on the FSM described above. The algorithm is adapted from Fisher's algorithm [16].

FSMs are usually used for parsing a sequence of characters from the beginning of the input. Interestingly, the FSM can also be adapted to process a CSV chunk whose beginning is in the middle of the input. The basic idea is to enumerate all possible states as the starting state, and simultaneously execute multiple state machines, one for each possible starting state. Not all FSMs can pass through the chunk. Some FSMs may encounter an invalid transition ("-" in Figure 4), which in turn indicates that the corresponding starting state is invalid for the chunk. After passing through the whole chunk, all invalid starting states are identified. Then, the chunk is ambiguous if and only if the remaining valid starting states are all either unquoted states or quoted state.

	l	i	c	e	,	"		,	"	,	1	6	\n	B	o	b	,	"	,	"	,	1	7
R	U	U	U	U	F	Q	Q	E	F	U	U	R	U	U	U	F	Q	Q	E	F	U	U
F	U	U	U	U	F	Q	Q	E	F	U	U	R	U	U	U	F	Q	Q	E	F	U	U
U	U	U	U	U	F	Q	Q	E	F	U	U	R	U	U	U	F	Q	Q	E	F	U	U
Q	Q	Q	Q	Q	Q	E	F	Q	Q	Q	Q	Q	Q	Q	Q	Q	E	F	Q	Q	Q	Q
E	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

(a) Ambiguous CSV chunk

	l	i	c	e	,	"	\n	"	,	1	6	\n	B	o	b	,	"	M	"	,	1	7
R	U	U	U	U	F	Q	Q	E	F	U	U	R	U	U	U	F	Q	Q	E	F	U	U
F	U	U	U	U	F	Q	Q	E	F	U	U	R	U	U	U	F	Q	Q	E	F	U	U
U	U	U	U	U	F	Q	Q	E	F	U	U	R	U	U	U	F	Q	Q	E	F	U	U
Q	Q	Q	Q	Q	Q	E	R	Q	Q	Q	Q	Q	Q	Q	Q	Q	E	-	-	-	-	-
E	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

(b) Unambiguous CSV chunk

Figure 5: Determining ambiguity using FSM (invalid starting states are marked gray)

Figure 5 demonstrates the process to examine the ambiguity of two example CSV chunks. In Figure 5(a), the starting state E encounters an invalid transition after reading the first character in the chunk, and thus is considered to be an invalid starting state. In contrast, all other starting states successfully pass through the whole chunk. Since the remaining starting states R, F, U, and Q fall into two categories, the chunk is ambiguous. This is consistent with the observation we made in Figure 1. Actually, the sequence of states starting from the R, F, and U states corresponds to Interpretation 1 in Figure 1, whereas the the sequence of states starting from the Q state corresponds to Interpretation 2. In Figure 5(b), the example chunk has an additional invalid starting state Q, because other are not allowed after the state E, which is transited from the starting state Q after reading the string "lice,"\n",16\nBob,"". This eliminates the possibility of having Q as a starting state. Thus, all valid starting states are unquoted and the example chunk is therefore unambiguous.

4.4 Determining Ambiguity using Patterns

Although the FSM-based algorithm (Section 4.3) gives us a fundamental view on this problem, it is unfortunately not a sufficiently practical solution in terms of processing speed. It requires to run up to five FSMs simultaneously, and may dominate the overall execution time. This issue gets more serious when the algorithm is used with state-of-the-art parsers [19, 24], which are up to orders of magnitude faster than FSM-based parsers. To address this issue, we develop a novel approach that can fully determine the ambiguity of a CSV chunk in a much faster way.

We introduce two interesting string patterns. The first one, called *q-o pattern*, represents a class of two-char strings that start with a quote followed by an other (i.e., any character other than a quote, a comma, and a newline). The second one, called *o-q pattern*, represents a class of two-char strings that start with an other followed by a quote.

Both q-o and o-q patterns have a crucial property: *for all possible input states, the FSM transits into the same output state, after reading an input string following the pattern.* Figure 6 shows the transitions starting from all five possible states for both patterns. It is clear to see that all starting states converge to the Q state for the q-o pattern, and converge to the E state for the o-q pattern.

	quote	other
R	Q	Q
F	Q	Q
U	-	-
Q	E	-
E	Q	Q

(a) q-o pattern

	other	quote
R	U	-
F	U	-
U	U	-
Q	Q	E
E	-	-

(b) o-q pattern

Figure 6: State transitions on q-o and o-q patterns

Due to this property, the pattern-based algorithm to determine whether or not a given chunk is ambiguous is remarkably simple: the chunk is ambiguous if and only if it contains neither q-o pattern strings nor o-q pattern strings. The correctness of the pattern-based algorithm is proved

by Property 2 and Property 3 shown as follows. The proofs of these properties rely on the FSM-based algorithm we described in Section 4.3 and can be found in Appendix A.

Property 2. *A chunk that contains q-o pattern string(s) or o-q pattern string(s) must be unambiguous.*

Property 3. *An unambiguous chunk must contain q-o pattern string(s) or o-q pattern string(s).*

Consider the examples in Figure 5 again. The CSV chunk in Figure 5(a) does not contain any strings following either q-o or o-q patterns, and therefore is an ambiguous chunk. In contrast, the chunk in Figure 5(b) contains the q-o pattern string `""M"`, and thus is an unambiguous chunk. These results are consistent with the outcome of the FSM-based method.

4.5 Speculation

As described in the previous section, CSV chunks that contain neither q-o pattern strings nor o-q pattern strings are inherently ambiguous for parallel parsing. In these cases, we have to speculate whether or not the chunk is quoted. In this section, we present a speculation algorithm that exploits conditional probabilities based on the data in each chunk to speculate on the starting state of the chunk.

We first explain the core insights and intuition behind the algorithm by taking the ambiguous CSV chunk in Figure 1 as an example again. Even though both interpretations for the example chunk are completely possible in theory, the first interpretation looks much more likely than the second interpretation. This is because that if, otherwise, the second interpretation is correct, it is too coincidental that the strings inside the quoted fields (e.g., ",513\nBob,") in the first interpretation happen to completely follow the CSV specification in the alternative interpretation. If the string does not start with `comma` or end with `comma`, the string in second interpretation is not a valid CSV string anymore and the chunk becomes unambiguous. This example reveals the intuition behind our speculation algorithm: we randomize field strings in a chunk in one interpretation, and compute the probability that the chunk also follows the CSV specification in the alternative interpretation; the higher the value, the more likely the predicted interpretation is correct.

Formally, we model this problem as a conditional probability problem. Given a chunk, let s_Q denote the sequence of states that is produced by reading the quoted chunk, and s_U denote the sequence of states that is produced by reading the unquoted chunk. Let C be the set of chunks that either produce s_Q if the chunk is quoted, or produces s_U if the chunk is unquoted. In addition, let Q be the event that a chunk in C is quoted (i.e., it starts in the middle of a quoted field), and U be the event that the chunk is unquoted (i.e., it does not start in the middle of a quoted field). We also use V_Q and V_U to denote the events that a chunk in C is valid starting from a quoted state and an unquoted state, respectively. Thus, $V_Q \cap V_U$ represents the event of a chunk in C is ambiguous.

Then, the probability of an ambiguous chunk in C being unquoted is given by the following conditional probability:

$$P(U|V_U \cap V_Q) = \frac{P(U) \cdot P(V_U \cap V_Q|U)}{P(V_U \cap V_Q)} \qquad \text{(Bayes' rule)}$$
$$= \frac{P(U) \cdot P(V_Q|U)}{P(V_U \cap V_Q)}. \qquad (U \subset V_U) \quad (1)$$

Similarly, the probability of an ambiguous chunk in C being quoted is given by

$$P(Q|V_U \cap V_Q) = \frac{P(Q) \cdot P(V_U|Q)}{P(V_U \cap V_Q)}. \quad (2)$$

To speculate the starting state of the chunk, we compare $P(Q|V_U \cap V_Q)$ and $P(U|V_U \cap V_Q)$ by computing the ratio between them:

$$\frac{P(U|V_U \cap V_Q)}{P(Q|V_U \cap V_Q)} = \frac{P(U) \cdot P(V_Q|U)}{P(Q) \cdot P(V_U|Q).} \quad (3)$$

Next, we compute $P(V_Q|U)$ based on the characters in the chunk. According to Property 2, an ambiguous chunk must contain neither q-o nor o-q pattern strings. Therefore, given a chunk in C that is valid starting from an unquoted state, the chunk is also valid starting from a quoted state, if the first character after an opening quote is either a `comma` or a `newline` and the last character before a closing quote is either a `comma` or a `newline`. Let q to be the probability of a character is either a `comma` or a `newline`. Then, for the i-th quoted field in the chunk, the probability p_i of the field string being also valid in the alternative interpretation is: 1) 1 if the field string is empty; 2) q if the quoted field is a partial field or its length is 1; and 3) q^2 if the complete quoted field contains at least two characters. Thus, $P(V_Q|U)$ can be computed by Πp_i for all quoted strings in the chunk.

In order to compute $P(U)$ and $P(Q)$ in Equation 3, we assume that the chunk shows the same ratio of quoted characters to the whole CSV input. Thus, $P(U)$ (or $P(Q)$) equals to the percentage of quoted (or unquoted) states in the chunk. Let u_U and u_Q be the ratio of unquoted states in state sequence s_U and s_Q, respectively. Then, we have

$$P(U) = u_U \cdot P(U|V_U \cap V_Q) + u_Q \cdot P(Q|V_U \cap V_Q), \quad (4)$$
$$P(Q) = (1 - u_U) \cdot P(U|V_U \cap V_Q) + (1 - u_Q) \cdot P(Q|V_U \cap V_Q). \quad (5)$$

By substituting Equation 4, 5 into Equation 3, we obtain a quadratic equation in $\frac{P(U|V_U \cap V_Q)}{P(Q|V_U \cap V_Q)}$. By solving this equation, we get the value of $\frac{P(U|V_U \cap V_Q)}{P(Q|V_U \cap V_Q)}$. If this value is greater than 1, it is more likely that the chunk is unquoted. Otherwise, if the value is less than 1, the chunk is more likley quoted.

Taking the ambiguous chunk in Figure 1 as an example again, in the first interpretation (starting with an unquoted state), there are two quoted fields of length 1. Thus, we have $P(V_Q|U) = q^2$. Similarly, in the second interpretation, there are two partial quoted fields and a complete quoted field of length eight. Hence, $P(V_U|Q) = q \cdot q^2 \cdot q = q^4$. Then, we count the number of unquoted states in both interpretations, and get $u_U = 18/22$ and $u_Q = 4/22$. In this example, assuming that all characters in the ASCII character set are uniformly distributed, thus $q = 2/128$. By substituting all parameters into the quadratic equation, we obtain $\frac{P(U|V_U \cap V_Q)}{P(Q|V_U \cap V_Q)} = 18427 \gg 1$, meaning that it is very likely that the example chunk is unquoted.

Finally, let us consider a special case where $\frac{P(U|V_U \cap V_Q)}{P(Q|V_U \cap V_Q)} = 1$. In this case, we cannot determine the starting state of the chunk according to the probability model. Instead, we fall back to a speculation method based on a heuristic: chunks are usually much larger than records in big data applications. To apply this heuristic, we compute the maximum record size in the chunk for both starting states, and choose the starting state that results in smaller maximum record size.

As an example, consider chunks that do not contain any quotes. For these chunks, we always have $\frac{P(U|V_U \cap V_Q)}{P(Q|V_U \cap V_Q)} = 1$ by solving the equation. In this case, we cannot determine between the two cases: 1) the chunk is completely outside of any quoted fields, and 2) the whole chunk is a part of a (large) quoted field. In the former case, the maximum record size equals to the chunk size. In the latter case, the maximum record size is smaller than the chunk size. According to the heuristic, we always choose the case with smaller records and thus speculate that the chunk is unquoted.

5 SYNTAX ERROR HANDLING

Syntax errors are norms rather than exceptions in real-world datasets. These syntax errors are usually caused by a wide variety of reasons, ranging from program bugs, character set and encoding errors, to incorrect usage of toolsets. This issue gets more serious for CSV data, because, as we mentioned before, the CSV format is not fully standardized and there are many other CSV variations. Users may incautiously feed the CSV data in one format variation into a parser that is designed for another format variation. All that said, a practical CSV parsing solution should be capable of dealing with such errors. In this section, we analyze and extend the proposed approaches to handle malformed CSV data.

5.1 Problem Definition

In this paper, we focus on detecting syntax errors in CSV data. Formally, a CSV input has *syntax errors* if and only if the FSM either encounters an invalid transition or ends in non-final states, by reading the CSV input. Semantic errors, such as inconsistent field types or invalid field values, are undetectable from the syntax point of view, and are therefore out of scope of this paper. Automatic correction of detected syntax errors is an interesting direction for future work but is also out of scope of this paper.

We say that a parser is *error-detectable* if it meets two requirements: 1) the parser detects error(s) if and only if the CSV input contains syntax errors; 2) the parser reports the first error if there are multiple syntax errors in the input. The first requirement is for correctness: if a parser fails to find an error in a CSV input, the parser may produce incorrect parsing results without being aware of it. The second requirement is for usefulness: since subsequent errors are usually caused by errors appearing before, only the first error is guaranteed to be an actual error from a user perspective.

In order to formally describe syntax errors encountered during parsing, we augment the FSM for CSV data (see Figure 4) with a new error state `X`. Figure 7 shows the transition table of the augmented FSM. As can be seen from the figure, two transitions shift the state from non-error states to the error state: $\texttt{U} \xrightarrow{\texttt{quote}} \texttt{X}$ and $\texttt{E} \xrightarrow{\texttt{other}} \texttt{X}$. Once a FSM gets into the `X` state, it stays in this state until the end of the input. As an example, Figure 8 shows a CSV input with syntax errors and the corresponding state on each character of the input (we can ignore the chunking on the input for now).

	quote	comma	newline	other
R (Record start)	Q	F	R	U
F (Field start)	Q	F	R	U
U (Unquoted field)	X	F	R	U
Q (Quoted field)	E	Q	Q	Q
E (quoted End)	Q	F	R	X
X (Error)	X	X	X	X

Figure 7: Augmented FSM for CSV format

5.2 Extending Framework

In this section, we extend our framework (described in Section 3.1) to deal with syntax errors. The framework assumes that the states of all characters in the CSV input have been known. This is obviously an unrealistic assumption (except for the first chunk) for a distributed parsing. In this sense, the framework is only served as a theoretical tool to verify whether a distributed parsing approach is error-detectable.

The extended framework defines the starting position of each adjusted chunk as follows. The adjusted chunk of the first chunk in the CSV input always starts with the beginning of the chunk. For other chunks, if the chunk contains characters at the `R` state, its adjusted chunk starts immediately after the first character at the `R` state. Otherwise, its adjusted chunk is either undefined if the chunk contains characters

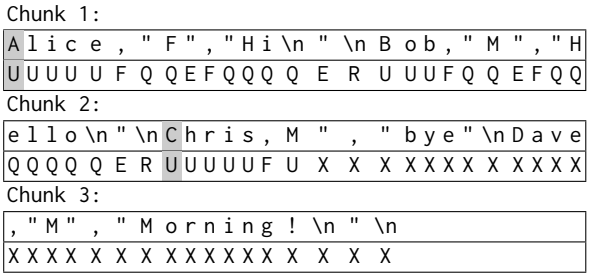


Figure 8: Example CSV input with errors (expected starting position of adjusted chunks are marked gray)

at the X state, or empty if the chunk does not contain any characters at the X state.

Once the starting position of an adjusted chunk is determined, the worker invokes the CSV parser to parse records starting from this position. The parser scans the CSV input until it encounters the first character at the R state after the end of the chunk, or until it encounters a syntax error at the X state. For the latter case, the worker reports the syntax error to the master, which is responsible for reporting the first syntax error in the CSV input to the application.

Figure 8 shows an example CSV input that is split into three chunks. We also show the states in the FSM after reading each character in the input. Note that the second chunk contains a syntax error – the quote after "M" is not allowed and results in the X state. As per the specification of the extended framework, the starting positions of adjusted chunks are marked gray in the figure. The first adjusted chunk always starts from the beginning of the chunk. The second adjusted chunk starts after the marked newline character, which is the first character at the R state within the chunk. The third chunk has no defined adjusted chunk as it contains characters at the X state but no character at the R state.

According to Property 4, the extended framework is error-detectable. The proof can be found in Appendix A.

Property 4. *The extended framework is error-detectable.*

5.3 Extending Two-Pass Approach

In this section, we adapt the two-pass approach to handle syntax errors. In addition to the parsing results and the position of its adjusted chunk, each worker also returns the error information to the master. If there are multiple detected errors, the master is responsible for finding the first error in the CSV input, and reporting the error to the application.

We now show that the two-pass approach follows the extended framework and therefore is error-detectable. The two-pass approach only differs from the extended framework by how it determines the starting positions of adjusted chunks. For this reason, we only need to demonstrate that the starting positions of adjusted chunks determined by the two-pass approach follows the specification of the extended framework, which can be shown in three cases. Case 1: if the chunk contains the X state but no R state, its adjusted chunk is undefined according to the framework. Thus, the algorithm does not violate the requirements of the framework, regardless of where the adjusted chunk is determined. Case 2: if the chunk contains neither X nor R states, the adjusted chunk is empty, following the specification of the extended framework. Case 3: if the chunk contains the R state, then there is no syntax error in the CSV input before the R state. Thus, the algorithm can correctly determine the starting state of the chunk with respect to whether the beginning of the chunk is inside a quoted field, according to the number of quotes in all previous chunks. Then, based on this information, the algorithm can correctly select the first character at the R state as the boundary, even though the chunk may contain characters at the X state in the remaining of the chunk.

Consider the CSV input in Figure 8 as an example. In the first pass of the two-pass approach, workers counts the number of quotes in each chunk. According to these statistics, the master infers that the second chunks is quoted. As a result, the master determines that the second adjusted chunk starts after the marked newline in Chunk 2, which is indeed the first character at the R state in Chunk 2, as requested by the extended framework. Starting from this position, the CSV parser scans the second adjusted chunk and detects the syntax error at "M"". The master also (incorrectly) infers that the third chunk is quoted, and treats the newline inside the quoted field "Morning!\n" as the beginning of the adjusted chunk. This results in an undesired syntax error at the first "M" in Chunk 3. Nevertheless, this error is ignored by the master due to the first syntax error found in Chunk 2.

5.4 Extending Speculative Approach

Next, we extend the speculative approach to handle syntax errors. In the speculative approach, each worker parses the assigned chunk based on the predicted starting position of its adjusted chunk, and reports encountered errors during parsing to the master. Such errors may be misreported due to a misprediction, but can be confirmed if all previous adjusted chunks are connected end to end and there are no errors detected in all previous chunks. Algorithm 1 shows the pseudocode for validating speculations by the master.

We then demonstrate that the speculative approach also follows the extended framework. Let the k-th chunk in a CSV input be the last chunk whose adjusted chunk contains non-X states (the k-th chunk is the last chunk if there is no syntax error). Then, if the first k adjusted chunks are not connected end to end, we fall back to the two-pass approach, which follows the extended framework. Otherwise, if the first k adjusted chunks are indeed connected, according to

Algorithm 1 ValidateSpeculativeParsing(C)

Input: C: the sequence of chunks in the CSV input

```
1: R := ∅                                          ▷ initialize parsing results
2: for i := 0 to |C| do
3:     if i = 0 or C_i.start = C_{i−1}.end + 1 then
4:         if C_i.has_error = false then
5:             R := R ∪ C_i.parsing_results        ▷ speculation successes
6:         else
7:             throw exception on the error.       ▷ confirmed error
8:     else
9:         Fall back to two-pass approach.         ▷ speculation fails
10: return R
```

Property 1, all speculations on the first k chunks are successful, meaning that the first k adjusted chunks all start after the first character at the R state in each chunk. Thus, the speculative approach follows the extended framework, and is therefore error-detectable.

Nevertheless, syntax errors may increase the likelihood of mispredictions in rare cases. Figure 8 shows such an example. In the second chunk, we find an o-q pattern string "M"", which indicates that the chunk is unquoted. Consequently, we (incorrectly) infer that the first record delimiter is the newline after "ello", which results in unconnected adjusted chunks between the first and second chunks and introduces extra cost in executing the fallback parsing solution. The fundamental problem behind this is that Property 2 and Property 3 do not hold for an ill-formed CSV input. Fortunately, this issue does not occur when there are o-q or q-o pattern strings appearing before the syntax error within the chunk. In this case, the first record delimiter can be correctly determined by the first pattern string. In general, it is uncommon to see a syntax error before all pattern strings in a chunk, because real-world datasets with syntax errors usually contains a great number of pattern strings in each chunk.

6 APACHE SPARK INTEGRATION

In this section, we present our implementation of both the two-pass and speculative approaches in Apache Spark. These approaches are implemented based on the primitives provided by Spark without changing its architecture, and thus inherit the fault-tolerance characteristics of Spark.

Apache Spark is built based on the RDD abstraction. A RDD is an immutable, partitioned collection of data records. A query in Spark is compiled into a series of RDD transformations. Fault-tolerance is achieved by keeping track of the lineage of RDDs so that they can be reproduced in case of failure. For a query on raw CSV data, a scan operation reads and parses CSV data and produces a RDD that can be consumed by subsequent query processing operations.

For the two-pass approach, we implemented it as two RDD transformations: each transformation corresponds to one pass in the two-pass approach. The master first splits the file into equal-sized chunks according to the parallelism settings. The first transformation reads these chunks and produces the first RDD, according to the method we described in Section 3.2. Each partition of the RDD contains the number of quotes and the positions of record delimiter candidates in the corresponding chunk. Based on this RDD, the master computes a new RDD that include positions of all adjusted chunks. Taking the new RDD as input, the second transformation parses adjusted chunks in parallel and creates the result RDD that is then consumed by subsequent operations.

The speculative approach is implemented as a single RDD transformation. Similar to the two-pass approach, the master splits the file into equal-sized chunks and creates a RDD on boundary information of these chunks. In the speculative parsing transformation, each partition of the RDD is first speculated using the speculation algorithms described in Section 4; and its predicted adjusted chunk is then parsed optimistically. The parsing results are stored in a RDD. In addition to the parsing results, the transformation also needs to send the positions of predicted adjusted chunks to the master for validation purposes. We decide not to include this information in the result RDD, as the result RDD is supposed to be directly consumed by any subsequent operations. Instead, the operation sends this information through the `Accumulators` mechanism provided by Spark. Accumulators are essentially write-only variables, and can be recovered from failure automatically.

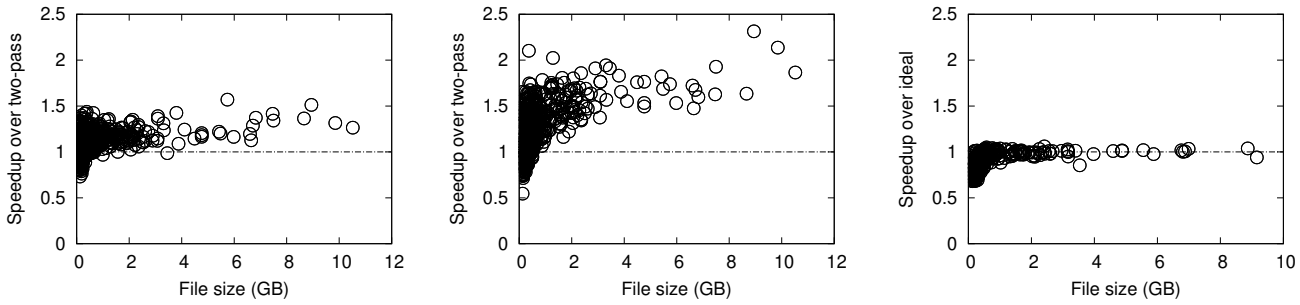
The master is responsible for validating speculations before feeding the result RDD to subsequent operations in the query plan. We implemented the algorithm described in Section 5.4 to validate the positions of predicted adjusted chunks. If any speculation fails, the master discards the result RDD, and creates a new query plan that includes the two-pass approach as the parsing operation.

7 PERFORMANCE EVALUATION

We conducted all our experiments on Microsoft Azure cloud computing platform. We deployed our modified version of Apache Spark (version 2.2.2) on a cluster consisting of 1 master node and 8 worker nodes. Workers are deployed on virtual machines with 4 vCPUs, 128GB RAM, and 2TB SSDs, running Ubuntu 18.04. In our experiments, each worker is configured to use 1 vCPU.

Datasets. We downloaded all CSV datasets from Kaggle data science repository[1]. As of the time of our experiments, there are more than 11,000 well-formed CSV files, and 41 files with syntax errors. We study the parsing performance on the ill-formed data in Section 7.6, and use the well-formed

[1] https://www.kaggle.com/



(a) Speedup of speculative over two-pass with default parser

(b) Speedup of speculative over two-pass with Mison parser

(c) Speedup of speculative over ideal with default parser

Figure 9: Performance comparison among speculative, two-pass, and ideal approaches on individual files

data in the rest of evaluations. The size of these CSV files ranges from 3 bytes to 11GB. All files are in UTF-8 encoding.

To evaluate the parsing performance, we run a simple query "`SELECT count(*) FROM FILE`" on each CSV file to minimize impacts of query processing on execution time. For each file, we report the average execution time over 3 runs. Previous studies have shown that parsing dominates (>80%) the end-to-end query execution time in Spark, when querying raw data [19, 24].

Implementation. We implemented the two-pass and speculative approaches in Apache Spark. The detailed description of our implementation can be found in Section 6. To serve as a yardstick, we include a comparison with a baseline method that sequentially parses CSV files. This is the current implementation in Spark when parsing a standard CSV file (by enabling the "`multiLine`" option) rather than the simplified format variation (where newlines are not allowed in quoted fields). In the graphs below, the tag *sequential* refers to this approach. The tag *two-pass* refers to the two-pass approach we presented in Section 3.2. The tag *speculative* refers to the speculative approach proposed in this paper. On each worker, we use the pattern-based algorithm to examine the ambiguity (Section 4.4). Unless otherwise stated, speculation is performed on a 64KB prefix of each chunk.

We evaluated each approach with two CSV parsers: 1) the CSV parser [6] used in Spark, and 2) the state-of-the-art parser Mison [19]. Mison is originally designed for the JSON format [19]. We adapted the Mison techniques to CSV format and implemented a Mison CSV parser in C++. Spark invokes this native Mison CSV parser through JNI mechanism. In the graphs below, the tags *default* and *mison* refer to these two parsers, respectively.

7.1 Experiment 1: Parsing Performance

We first compare the overall performance of the three distributed parsing approaches. Figure 10 shows the performance of sequential, two-pass, and speculative approaches with default and Mison parsers. For each approach, we show

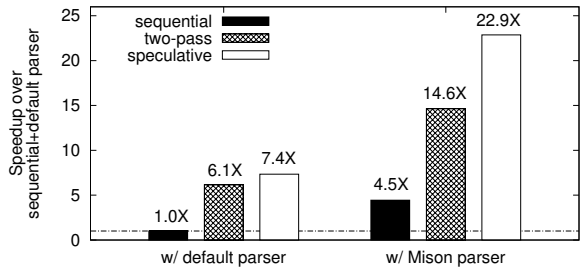


Figure 10: Overall performance comparison among sequential, two-pass, and speculative approaches

the average speedup over the sequential approach with default parser on all CSV files that are larger than 1GB. The execution time on a smaller file is usually dominated or significantly impacted by other components in the total execution time such as scheduling, query optimization, query compilation, and so on. We evaluate the impact of file sizes in the next experiment.

When using the default parser, speculative approach is 20% faster than two-pass approach, and achieves a speedup of 7.4X over sequential approach, which is close to the ideal speedup of 8X. The improvement over two-pass approach comes from eliminating the additional pass of scan in two-pass approach that is used to identify record delimiters. Unlike this conservative approach, speculative approach bets on a speculation-based method by only looking at a small portion (64KB) of data in each chunk and optimistically parsing the data in the risks of mispredications. This strategy wins because mispredictions rarely occur, as we will show in Section 7.3. Eliminating this first pass of scan results in 20% improvement, because the first pass of two-pass approach only counts the number of quotes and thus is much faster than the second pass on full parsing.

The right side of Figure 10 shows the performance of the three approaches with Mison parser. Clearly, it is more beneficial to use speculative approach than two-pass approach with Mison: speculative approach is now 60% faster than

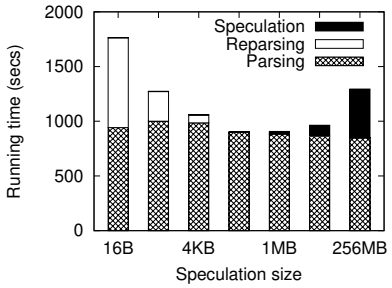


Figure 11: Time breakdown of speculative approach

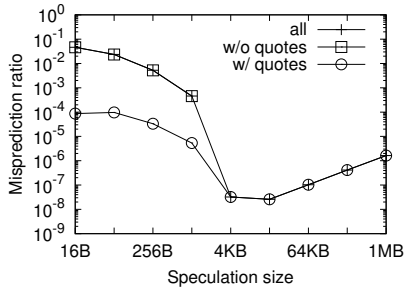


Figure 12: Misprediction ratio, varying speculation size

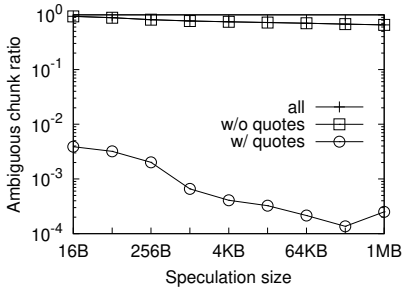


Figure 13: Ambiguous-chunk ratio, varying speculation size

two-pass approach. This is because, with Mison, the parsing time is reduced by 4-5X and thus the time spent on the first pass of two-pass approach accounts for a larger portion of the total execution time, which in turn leads to a more significant improvement of speculative approach.

Next, we analyze the performance improvement of speculative approach on individual files. Figure 9(a) and Figure 9(b) plot the speedup of speculative approach over two-pass approach on all files that are larger than 128MB, using default and Mison parsers respectively. For smaller files, the variance of speedup is very high, because the execution time is significantly impacted by other components in Spark (e.g., query optimization, query compilation, scheduling) other than parsing. As the file size increases, we see that the speedup generally increases and the variance reduces. For some large files, speculative approach achieves a speedup of 1.5X over two-pass approach with default parser. With Mison, speculative approach is up to 2.4X faster than two-pass approach.

Furthermore, it is interesting to ask how close the performance of speculative approach is to an ideal case, where there is an oracle to determine chunk boundaries without incurring additional cost. To simulate such an ideal parallel parser, we configured unmodified Spark to look for newlines (rather than actual record delimiters) to determine chunk boundaries by disabling the option "`multiLine`". This, of course, results in a lot of parsing errors on CSV files that contain newlines within quoted fields. Therefore, in order to avoid interference caused by handling exceptions, we run experiments on a subset of files where there are no quoted fields. It is worth noting that on this set of files, speculative approach is expected to show the worst performance on speculation, because a chunk without quotes is always ambiguous due to the lack of q-o and o-q patterns (according to Property 2). Nevertheless, as can be seen from Figure 9(c), the speculative approach achieves similar performance to the ideal approach, even though this set of files do not favor speculation. This is because, even in these cases, the overhead associated with speculative parsing is quite negligible. We will continue to analyze the overhead in the next section.

7.2 Experiment 2: Speculation Overhead

This section examines the overhead of speculative approach. The overhead comes from two sources: 1) the speculation process which is to scan a portion of a chunk to determine the chunk boundary; and 2) reparsing in case of mispredictions. Figure 11 shows the time breakdown for three components: basic parsing time, speculation time, and reparsing time. Each bar in the figure represents the overall running time on all files larger than 128MB using a specific *speculation size*, i.e. the size of data accessed by the speculation algorithm in each chunk.

As can be seen from the figure, the basic parsing time remains unaffected by varying speculation size. The reparsing time, however, decreases dramatically as the speculation size increases. This is because, when the speculation is unreasonably small (e.g., 16 bytes or 256 bytes), the speculation algorithm may look at a string that is completely inside a quoted field and thus it is inherently impossible to distinguish between a quoted string and an unquoted string. However, when the speculation size is greater than 4KB, the reparsing time is reduced to zero, meaning that speculations on all these files are successful. On the other hand, the speculation time increases as the speculation size increases, because that the speculation algorithm needs to scan a larger region to determine the ambiguity or make a speculation.

Figure 11 illustrates the tradeoff between speculation and reparsing time, and shows that the speculative approach achieves the best performance when the speculation size is 64KB. At this speculation size, speculation time and reparsing time account for 0%, and <1% of the total execution time, respectively. The overhead of speculation is negligible.

7.3 Experiment 3: Speculation Accuracy

This experiment measures the speculation accuracy of our speculation algorithm. As we have shown in the previous section, there are no mispredictions when the speculation size is greater than 4KB, showing a 100% speculation accuracy. However, this number is not statistically meaningful,

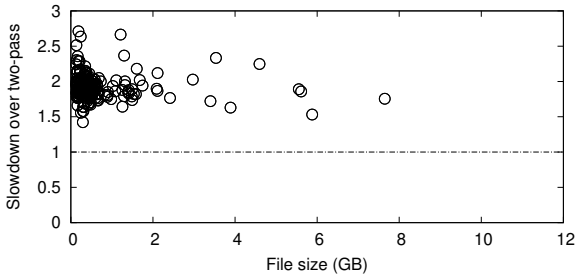


Figure 14: Performance comparison between speculative and two-pass in worst cases

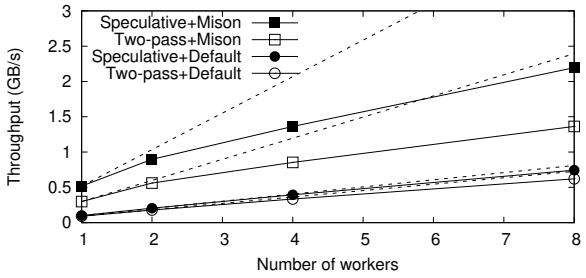


Figure 15: Throughput varying the number of workers

due to the insufficient number of speculations performed. This is because each file is split into only 8 chunks, according to the number of workers. In this experiment, in order to get a high-precision speculation accuracy, we generate many more chunks by splitting each file into chunks whose length equals to the speculation size. In this way, we can generate a large number of chunks. For example, when the speculation size is 64KB, we produce more than 9.6 million chunks.

Figure 12 plots the misprediction ratio by varying the speculation size from 16 bytes to 1MB. As the speculation size increases, the overall misprediction ratio (labeled "all" in the figure) drops to as low as 2.6×10^{-8}. At the default speculation size of 64KB, the misprediction ratio is 1.0×10^{-7}, meaning that the method mispredicts less than once in ten million chunks.

In order to gain a better understanding of the speculation accuracy, we distinguish two cases: chunks without quotes, and chunks with quotes. As discussed in Section 4.5, the speculation algorithm always predicts that a chunk without any quotes is unquoted. This prediction is obviously wrong for a chunk that fits in a quoted field. As a result, the misprediction ratio for chunks without quotes is fairly high for smaller speculation size. As the speculation size increases, the misprediction ratio decreases significantly because fewer quoted fields are larger than the speculation size. When the speculation size is 4KB, the misprediction ratio is reduced to 0% as no quoted fields are larger than 4KB. For chunks with quotes, the speculation algorithm may also make incorrect speculations. However, this type of misprediction accounts for a tiny portion of all mispredictions when the speculation size is smaller than 4KB. As the speculation size grows beyond 4KB, all mispredictions come from chunks with quotes. When the speculation size is greater than 16KB, the speculative approach makes only one mispredication on all tested chunks. In these cases, the misprediction ratio goes up simply because there are fewer chunks as the speculation size (and chunk size) increases.

Next, we plot the ratio of ambiguous chunks by varying the speculation size in Figure 13. Surprisingly, a majority of chunks are ambiguous. This is mainly because around 70% of chunks contain no quotes, and thus must be ambiguous (according to Property 2). Other than chunks without quotes, less than 1% of chunks are ambiguous.

7.4 Experiment 4: Worst Case Analysis

In this section, we aim to show the worst-case performance of the speculative approach (i.e., the performance when speculation fails). In order to generate a larger set of workloads that make our speculation algorithm mispredict, we intentionally set the speculation size to an unsatisfactory value – 256 bytes. By doing so, we collect 201 files that are used as the datasets for this experiment.

Figure 14 plots the slowdown of the speculative approach over the two-pass approach on these "worst-case" datasets. Similar to the results we showed previously, the variance is high when files are small. For files that are larger than 1GB, the average slowdown factor is 2.1X. This number is consistent with the fact that we have to run both the speculative approach and the two-pass approach in these worst cases.

7.5 Experiment 5: Scalability Evaluation

Figure 15 shows the throughput of two-pass and speculative approaches with default and Mison parsers on all files larger than 1GB, when varying the number of workers from 1 to 8. The dashed lines represent ideal throughputs with linear scalability for each configuration. With the default parser, both two-pass and speculative approaches achieve nearly linear scalability. When using 8 workers, the two-pass approach achieves a speedup of 6.8X, whereas the speculative approach achieves a speedup of 7.4X. However, with the Mison parser, both approaches start to deviate from linear scalability when there are 4 or more workers. This is because Mison is so fast that the parsing time is remarkably reduced and accounts for a smaller portion of the execution time. Even for the largest file (11GB), the execution time with 8 workers is expected to be only around 2 seconds. At this scale of time, many factors may affect the execution time. To confirm this suspicions, we generated a larger file by duplicating a 11GB file 5 times. On this large file, both two-pass and speculative approaches achieve nearly linear scalability.

7.6 Experiment 6: Error Handling

Finally, we evaluate the speculative approach on the 41 ill-formed CSV files we downloaded from Kaggle. These syntax errors can be categorized into three classes: 37 files include unescaped quotes within quoted fields; 2 files have unmatched quotes; and 2 files use incorrect characters to escape quotes within quoted fields. Not surprisingly, the speculative approach successfully detects errors in all ill-formed CSV files and correctly reports the first error in each file.

Furthermore, we observe that all speculations are successful on these ill-formed files, meaning that the speculation algorithm successfully predicts the starting state of each chunk even when the chunk contains errors. As we discussed in Section 5.4, a syntax error causes a misprediction only when this error happens to be the first (o-q or q-o) pattern string in a chunk and therefore misguides the speculation algorithm. This result is in line with our analysis that these cases are fairly rare in practice.

8 RELATED WORK

Our work is closely related to the body of work in loading and processing raw data in database systems [7–9, 14, 15, 17, 18, 23]. In particular, NoDB [8, 9, 17, 18] is a pioneering work that builds structural index on raw CSV files, and adaptively and incrementally uses the index to load the raw data. Many systems have been built to meet this growing demand in industry. In AWS, for example, both Amazon Athena [1] and Amazon Redshift Spectrum [2] enable users directly query raw data in Amazon S3 using standard SQL, without complex and expensive ETL tasks to prepare and load data. Apache Spark [11, 26] is another prominent example of big data systems that support access to raw, unparsed data. Nevertheless, to the best of our knowledge, none of these systems are able to efficiently parallelize CSV parsing. These systems either impose additional constraints on CSV format (e.g., newlines are not allowed within quoted fields), or simply use sequential parsing. Our proposed distributed parsing approach is applicable to all these systems.

There have been extensive literature on parallel parsing [10, 13, 16, 20, 22, 27]. These prior work focus on either general context-free gramma [16, 22], or a specific language such as arithmetic expression [13], XML [20], HTML [27]. Our FSM-based algorithm for determining ambiguity (see Section 4.3) is based on Fisher's algorithm [16], which laid the foundation on parallel parsing. For a chunk of input, the algorithm uses a number of sequential parsers for all possible starting states in the FSM. This general approach, however, suffers from efficiency problems because it has to simultaneously run multiple parsers. Due to this reason, we propose our pattern-based algorithm for determining ambiguity (Section 4.4), which is customized for CSV format. In general, existing parallel parsing techniques developed for general context-free gramma or a specific language are unsuitable for CSV parsing. The simplicity of the CSV specification introduces opportunities for a more efficient parallelization approach. In this work, we develop speculation-based parsing approach customized for CSV parsing.

Recent years have seen some increasing interest in optimizing parsing for data analytics, driven by the demand for providing SQL query capability on raw data in big data systems. Many techniques [19, 23, 24] has been developed to accelerate data parsing. In particular, Mühlbauer et al. proposed to exploit SIMD parallelism to accelerate CSV parsing and data deserialization [23]. Mison [19] is a fast JSON parser particularly designed for data analytics applications. It allows analytical engines to push down query operations, e.g., projections and filters of analytical queries, into the parser, and thus avoids a great deal of wasted work by only parsing fields that are relevant to the query. It also exploits the parallelism available in modern processors to avoid pitfalls in traditional FSM-based parsing approach. Sparser [24] is a parsing technique applicable to both plain-text and binary formats such as CSV, JSON, and Avro [3]. Sparser can filter records without having to parse the input data. This is achieved by searching for records that may satisfy the filter predicates of the query, in the raw, unparsed input data. A matched record may be a false-positive result and must be validated using a standard parser. All these work is actually complementary to our speculative parsing approach, which can leverage these innovations to achieve the best parsing performance in each worker.

9 CONCLUSION AND FUTURE WORK

With the growing demand for querying raw data in big data systems, there is a strong need for efficient distributed parsing techniques. This paper proposes a speculation-based distributed parsing approach for CSV data. At the core of this solution is an algorithm that can speculatively determine record boundaries in individual chunks of CSV data, even when the CSV data is ill-formed. Our experimental results on a large number of real-world datasets demonstrate that the approach mispredicts in less than once in ten million chunks. Due to the near-perfect speculation accuracy, our parallel parser performs almost as well as an ideal parallel parser and achieves up to 2.4X speedup over existing methods.

For future work, we plan to extend our solution for other common data formats such as JSON and XML. Furthermore, in addition to parsing, the speculative approach for finding record delimiters could be used for many other applications, such as skipping syntax errors in CSV data, and sampling CSV data for machine learning applications.

REFERENCES

[1] Amazon Athena. https://aws.amazon.com/athena/.
[2] Amazon Redshift. https://aws.amazon.com/redshift/.
[3] Apache Avro. https://avro.apache.org/.
[4] Google BigQuery. https://cloud.google.com/bigquery/.
[5] ParaText. https://github.com/wiseio/paratext.
[6] Univocity Parsers. https://github.com/uniVocity/univocity-parsers.
[7] A. Abouzied, D. J. Abadi, and A. Silberschatz. Invisible loading: access-driven data transfer from raw files into database systems. In *EDBT*, 2013.
[8] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: efficient query execution on raw data files. In *SIGMOD*, 2012.
[9] I. Alagiannis, R. Borovica-Gajic, M. Branco, S. Idreos, and A. Ailamaki. Nodb: efficient query execution on raw data files. *Commun. ACM*, 58(12):112–121, 2015.
[10] H. Alblas, R. op den Akker, P. O. Luttighuis, and K. Sikkel. A bibliography on parallel parsing. *SIGPLAN Notices*, 29(1):54–65, 1994.
[11] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD*, 2015.
[12] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical report, Technial Report, UC Berkeley, 2006.
[13] F. Baccelli and T. Fleury. On parsing arithmetic expressions in a multiprocessing environment. *Acta Inf.*, 17:287–310, 1982.
[14] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel data analysis directly on scientific file formats. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 385–396, 2014.
[15] Y. Cheng and F. Rusu. Parallel in-situ data processing with speculative loading. In *SIGMOD*, 2014.
[16] C. N. Fischer. On parsing context free languages in parallel environments. Technical report, Ithaca, NY, USA, 1975.
[17] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here are my data files. here are my queries. where are my results? In *CIDR*, 2011.
[18] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive query processing on RAW data. *PVLDB*, 7(12):1119–1130, 2014.
[19] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: A fast JSON parser for data analytics. *PVLDB*, 10(10):1118–1129, 2017.
[20] W. Lu, K. Chiu, and Y. Pan. A parallel approach to XML parsing. In *IEEE/ACM GRID*, pages 223–230, 2006.
[21] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *PVLDB*, 3(1):330–339, 2010.
[22] M. D. Mickunas and R. M. Schell. Parallel compilation in a multiprocessor environment (extended abstract). In *Proceedings of the 1978 Annual Conference*, ACM '78, pages 241–246, New York, NY, USA, 1978. ACM.
[23] T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. Instant loading for main memory databases. *PVLDB*, 6(14):1702–1713, 2013.
[24] S. Palkar, F. Abuzaid, P. Bailis, and M. Zaharia. Filter before you parse: Faster analytics on raw data with sparser. *PVLDB*, 11(11):1576–1589, 2018.
[25] Y. Shafranovich. Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC 4180, Oct. 2005.
[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[27] Z. Zhao, M. Bebenita, D. Herman, J. Sun, and X. Shen. Hpar: A practical parallel parser for html-taming HTML complexities for parallel parsing. *TACO*, 10(4):44:1–44:25, 2013.

A THEOREMS AND PROOFS

Property 1. *A speculation on the k-th chunk is successful if the first k predicted adjusted chunks are connected end to end.*

Proof. This property can be proved easily by induction. Base case: when $k = 1$, the first adjusted chunk always starts from the beginning of the CSV input and ends at a record delimiter, and thus is considered to be successful. Induction step: let $n \in \mathbb{N}$ be given and suppose the property holds for $k = n$. We now show that the property also holds for $k = n+1$. If the first $n+1$ adjusted chunks are connected end to end, the first n adjusted chunks are also connected end to end. Thus, by induction hypothesis, the speculation on the n-th chunk is successful, meaning that the n-th adjusted chunk ends at a record delimiter. Since the $(n+1)$-th adjusted chunk is connected to the n-th adjusted chunk end to end, the $(n+1)$-th adjusted chunk starts immediately after a record delimiter. As the parser starts the parsing immediately after a record delimiter, it is able to correctly find a record delimiter as the end of the $(n+1)$-th adjusted chunk. As a result, the speculation on the $(n+1)$-th chunk is successful and the property holds for $k = n+1$. By the principle of induction, we have proved that the property folds for all $k \in \mathbb{N}$. □

Property 2. *A chunk that contains q-o pattern string(s) or o-q pattern string(s) must be unambiguous.*

Proof. For the string following either q-o or o-q patterns, the FSM transits all possible starting states to a single output state (`Q` for q-o pattern, and `E` for o-q pattern) by reading the string, as shown in Figure 6. Suppose that the end of the pattern is at the i-th character of the chunk. According to the state of the i-th character, we can find all possible states of the $(i-1)$-th character in the chunk, based on the transition table of the FSM. It is clear to see that all possible states of the $(i-1)$-th character are all either unquoted states or quoted state, because states in the same category (quoted or unquoted) always produce the same output state whereas states in different categories never transit into the same output state for any input character (as shown in Figure 4). Continuing this process, we know that all possible states of any character appearing before the i-th character are all either unquoted states or quoted state. This implies that the valid starting states of the chunk are all either unquoted states or quoted state. Then, by the algorithm described in Section 4.3, we know that the chunk is unambiguous. □

Property 3. *An unambiguous chunk must contain q-o pattern string(s) or o-q pattern string(s).*

Proof. Consider an unambiguous chunk that contains neither q-o nor o-q pattern strings. We now show that the valid starting states of the chunk must contain the states R, F, and Q. First, as these three states never transit to an invalid state by reading the next character (as shown in Figure 4), transitions of the FSM starting from these three states must be valid after reading the first character. Now, suppose that there is an invalid transition at the i-th character after the first character, i.e., $i >= 2$. Then, according to the transition table of the FSM, it much be one of the two cases: case 1) the state before reading the $(i-1)$-th character is Q and the next two characters are a q-o pattern string ($\mathsf{Q} \xrightarrow{\texttt{quote}} \mathsf{E} \xrightarrow{\texttt{other}} -$); or case 2) the state before reading the $(i-1)$-th character is in $\{\mathsf{R}, \mathsf{F}, \mathsf{U}\}$ and the next two characters are a o-q pattern string ($\{\mathsf{R}, \mathsf{F}, \mathsf{U}\} \xrightarrow{\texttt{other}} \mathsf{U} \xrightarrow{\texttt{quote}} -$). This means that the string at the $(i-1)$-th and i-th characters must be either a q-o pattern string or a o-q pattern string. Since we have a contradiction here, it must be that all transitions of the FSM are valid after the first character. Then, we know that the states R, F, and Q must be valid starting states, according to the FSM-based algorithm. Because the R and F states are unquoted states whereas the Q state is the quoted state. By definition, the chunk is an ambiguous chunk. This completes the proof by contraposition. □

Property 4. *The extended framework is error-detectable.*

Proof. Let the k-th chunk in a CSV input be the last chunk whose adjusted chunk contains non-X states (the k-th chunk is the last chunk if there is no syntax error). By definition, the adjusted chunks of the first k chunks are connected end to end, meaning that any character in the input that appearing before the first syntax error is contained in one and only one adjusted chunk. As per the extended framework, the adjusted chunk of each of the first k chunks always starts with the first character after the first R state within the chunk. Since R is the initial state of the FSM, the FSM that starts from this position produces exactly same transitions on the subsequent characters as the FSM that starts from the beginning of the input. As a result, each character before the first X state is processed correctly, meaning that no error can be detected before the first actual syntax error. Due to the same reason, the first syntax error must be detected using the FSM. The master then collects all errors detected by workers and reports the first one, which must be the first actual syntax error, as no errors can be detected before the first actual syntax error. □

B DISCUSSIONS

B.1 Supporting Multi-Byte Encodings

The parsing approaches presented in this paper are in general independent to the character encoding of the CSV input. These approaches rely on the decoder of the input to convert the encoded byte stream into (multi-byte) characters, and apply algorithms on the decoded characters. However, for common encodings such as UTF-8 and UTF-16, our algorithms can process an encoded byte stream as a sequence of single-byte characters, without having to decode the multi-byte characters.

In this section, we focus our discussion on the UTF-8 encoding, the dominant encoding of big data applications. UTF-8 is a variable-length character encoding that represents Unicode characters using one to four bytes. Figure 16 shows the structure of the UTF-8 encoding. For an ASCII character, i.e., an Unicode character whose code is in the range of $0 \sim 127$, UTF-8 maps it to a single byte with the same value as the ASCII character. Non-ASCII characters are encoded with multiple bytes, each of which is guaranteed to be different from any ASCII byte. This means that an ASCII byte in an UTF-8 byte stream must represent an ASCII character rather than a byte in a multi-byte character.

Character Length	1st Byte	2nd Byte	3rd Byte	4th Byte
1 byte	0xxxxxxx			
2 bytes	110xxxxx	10xxxxxx		
3 bytes	1110xxxx	10xxxxxx	10xxxxxx	
4 bytes	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Figure 16: The structure of UTF-8 Encoding (the x characters represent actual code of an Unicode character)

With this property, our parsing algorithms can process an UTF-8 byte stream as a sequence of single-byte characters, without having to decode the multi-byte characters. This is because our speculative parsing algorithms only examine if each input character is one of the three structural characters: quote, comma, or newline (see Figure 4). Quote, comma, and newline are all ASCII characters, meaning that a byte in the input byte stream represents a quote/comma/newline character if and only if its byte value is identical to the quote/comma/newline character. All other bytes, including other ASCII bytes or bytes in multi-byte characters, are then treated as "other" characters. Thus, a multi-byte character is treated as a sequence of up to 4 "other" bytes. Interestingly, this simplification does not impact the outcome of the algorithm to determine the ambiguity (Section 4.4): if the input contains q-o (or o-q) pattern characters, it must also contain q-o (or o-q) pattern bytes, and vice versa. However, this approach may impact the speculation accuracy of the speculation algorithm (Section 4.5), as it may change the number of "characters". Nevertheless, according to our experiments on a large number of real-world datasets, we found that this impact is negligible.

B.2 Boundary Handling

In the framework described in Section 3.1, we assume that workers can access any fragments of a file. In this section, we relax this assumption and extend the proposed approaches to handle records at chunk boundaries, without having workers access subsequent chunks.

We first extend the framework to handle records at chunk boundaries. In the extended framework, each worker finds the first and last record delimiters in the chunk that is assigned to the worker. The region between the two record delimiters is considered to be the adjusted chunk. Following the terminology of previous work [23], the incomplete record before the first record delimiter in the chunk is called the *widow record* of the chunk, whereas the incomplete record after the last record delimiter in the chunk is called the *orphan* record of the chunk. A worker is responsible for parsing records in its adjusted chunk, and sending the parsing results along with the widow and orphan records to the master. For each chunk, the master needs to concatenate the orphan record of the chunk and the widow record of the next chunk, and parse the concatenated record.

In the speculative approach, the master is responsible for validating whether or not the concatenated record at each chunk boundary is a complete and valid CSV record. If this is true, all adjusted chunks and concatenated records are connected end to end, meaning that speculations are successful. If any concatenated record is incomplete, at least one speculation made in chunks fails. In this case, we fall back on the two-pass approach to recover from the mispredication.