One Pass to Parse Them All: Fused Parallel CSV Processing

Simon Ellmann
TUM
Munich, Germany
simon.ellmann@tum.de

Thomas Neumann
TUM
Munich, Germany
neumann@in.tum.de

ABSTRACT

CSV remains one of the most widely used formats for exchanging tabular data, making efficient CSV processing an important problem. Yet most CSV parsers are sequential, failing to exploit the parallelism of modern hardware. While parallel CSV parsing approaches have been proposed in the literature, none of these seem to be used in practice. Conversely, a simple idea for synchronization-free speculative parsing that is used for parallel parsing, e.g., in DuckDB, has never been described in the literature, nor has it been exploited efficiently. In this paper, we close this gap. We contribute a) a description of how real-world CSV files can be parsed in parallel on commodity multicore CPUs, b) a new programming model for general-purpose CSV parsers that unifies parallel parsing and parallel data processing into one pass over the data, and c) a new vectorization strategy with efficient index and zero-copy record construction to accelerate parsing. Our evaluation shows that csveee, our parser, outperforms widely-used CSV parsers in single- and multi-threaded performance, and scales near-linearly to achieve throughput of up to 180 GB/s — 22× faster than DuckDB — all while remaining practical for integration into real-world data processing systems.

PVLDB Reference Format:
Simon Ellmann and Thomas Neumann. One Pass to Parse Them All: Fused Parallel CSV Processing. PVLDB, 19(11): 3579 - 3591, 2026.
doi:10.14778/3836663.3836710

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/ackxolotl/csveee-evaluation.

1 INTRODUCTION

CSV (Comma-Separated Values) is one of the oldest text-based data formats, even predating personal computers[1]. The format represents tabular data as plain text: fields are separated by a delimiter (typically a comma), and records are separated by newlines. Despite its age, it remains remarkably prevalent — probably due to its simplicity, as CSV files are easy for humans to read and write. The scale is substantial: Open data platforms from governments (e.g., data.gov, open.canada.ca) and companies (e.g., Google's Kaggle) host hundreds of thousands of CSV files for download; GitHub alone contains over 90 million CSV files[2]; and Amazon reports that CSV is still the dominant format for external data in Redshift [19].

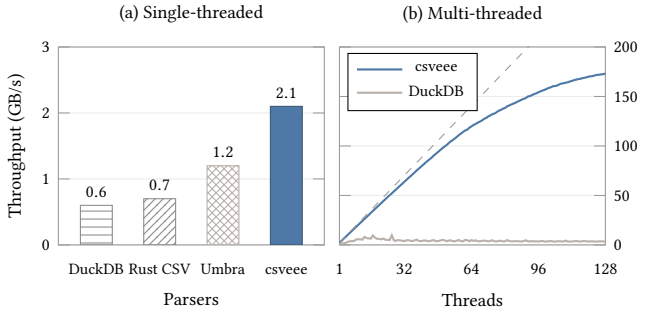


Figure 1: Counting line items with different CSV parsers, including our parser *csveee*, on a 128-core AMD EPYC server.

Although CSV was standardized in RFC 4180 in 2005 [15], the RFC merely documents "the format that seems to be followed by most implementations" and relies in large parts on recommendations rather than strict requirements. In practice, CSV files in the wild vary widely [9, 18]: different delimiters, inconsistent quoting, varying escape mechanisms, and records spanning multiple lines. These features make parallel parsing challenging, as the interpretation of bytes depends on the content that precedes them.

Perhaps for this reason, parsing performance has seen little improvement despite CSV's long history and continued relevance — most parsers remain sequential. The practical implications are significant: when reading a large CSV file, a modern database system like Umbra [12] using a traditional parsing approach may achieve a throughput of only 1.2 GB/s. At 1.2 GB/s, a modern multi-core system with hundreds of GB/s memory bandwidth sits largely idle — a needless waste of time and energy. Figure 1 illustrates the substantial throughput gaps with single- and multi-threaded parsing when counting rows of the TPC-H [17] `lineitem` table (79 GB).

In this paper, we present *csveee*, a parallel CSV parser that reaches up to 180 GB/s on real-world inputs — close to main memory bandwidth. Given a fixed number of fields per record, csveee handles arbitrary CSV files, including those with quoted fields, escape characters, and other complexities.

We make the following contributions:

- We survey parallel parsers in Section 2 and show why none is used in practice: GPU approaches require specialized hardware, the only CPU approach assumes strict RFC compliance, and all are incomplete research prototypes.
- We generalize speculative parsing in Section 3: unlike Ge et al. [4], who infer parse state from RFC-compliant quote patterns in the file, we speculate over an assumption set

Proceedings of the VLDB Endowment, Vol. 19, No. 11 ISSN 2150-8097.
doi:10.14778/3836663.3836710

[1] List-directed I/O with comma-separated values was supported by IBM Fortran as early as 1972.

[2] Based on a GitHub code search for `path:*.csv` in February 2026.

validated by a user-supplied oracle on the file structure, with chunk-local reparsing instead of a two-pass fallback; the oracle is exposed through a UDA-style programming model that fuses parsing and processing into a single pass.
- We extend vectorization end-to-end in Section 4: where prior SIMD parsers vectorize only character matching, we vectorize index + record construction, and UTF-8 validation.
- We show in Section 5 that across a 1,000-file real-world corpus, speculation always resolves within the assumption set — the reparse path is never triggered.
- We integrate our parser into a state-of-the-art database system, demonstrating that the design is practical to adopt.

2 PROBLEM SPACE & RELATED WORK

CSV parsing seems trivial: RFC 4180 defines records as CRLF-separated lines of comma-delimited fields, with a simple quoting and escape mechanism. In practice, however, CSV files vary widely: different characters for quotes, field delimiters (e.g., `\t`), and record delimiters (e.g., LF or mixed line endings); alternative escape mechanisms (e.g., backslashes); quotes inside unquoted fields; records with differing numbers of fields; or even comment lines. Parsing CSV files correctly therefore requires knowledge of the file's specific dialect properties. Determining these properties automatically is a non-trivial problem in itself [3, 18].

Once the dialect properties are known, there are two fundamentally different strategies to parse a CSV file: sequential or parallel parsing. Sequential parsing is straightforward: the parser processes the file from the first byte, transitioning between states and assembling records accordingly. Parallel parsing, in contrast, splits the file into chunks that are processed concurrently by multiple threads.

The central difficulty of parallel parsing lies in determining the parser state at the beginning of each chunk so that record boundaries are identified correctly — without scanning the file from the beginning, which would negate the benefit of parallelism. Record delimiters such as CRLF or LF are a natural starting point for locating record boundaries. However, when quoted fields are present, splitting a file at arbitrary offsets and searching for CRLF or LF is insufficient, as these byte sequences may appear inside a quoted field rather than marking a record boundary. Figure 2 illustrates the issue: When parsing from the beginning, the first field of the record following the header is `Tom`, while parsing from `Chicago` may identify the latter as the first field of a record. In Subsection 2.2, we show how parallel parsers approach this issue.

We classify two common sequential parsing strategies and the three parallel approaches proposed in the literature — all that exist to our knowledge — along four dimensions (Table 1):

(1) *Scalability* — whether performance increases with the number of processing cores and available memory bandwidth.
(2) *Efficiency* — whether the approach makes effective use of processors and memory resources.
(3) *Flexibility* — whether arbitrary CSV dialects can be parsed or whether certain properties, such as RFC compliance, are required.
(4) *Usability* — how easily the approach can be integrated into existing software and whether it depends on specific hardware or software.

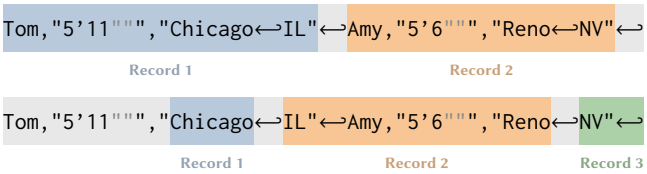


Figure 2: Quoting ambiguity in CSV files. Starting mid-file at `Chicago` (bottom), the parser mistakes the embedded ↩ for a record terminator and reads the closing " as opening a quoted part, misaligning every following record boundary.

2.1 Sequential Parsing

Sequential parsers process CSV files front to back on a single thread, maintaining parse state such as whether the parser is inside a quoted field. There are two different approaches: explicit state via a finite-state machine, or implicit state via vectorized parsing.

Finite-State Machine. In the classical approach, the parser is built around a finite-state machine (FSM), typically a deterministic finite automaton (DFA) with around 15 to 20 states. The state machine transitions on every input byte, assembling fields and records based on the current state. This is the predominant parsing strategy used in many widely adopted CSV parsers, such as the Rust crate csv[3], which we will call Rust CSV in what follows.

Single-threaded state-machine parsers do not *scale* `−−`, as they cannot exploit the memory bandwidth and processing power of modern hardware across multiple cores. Moreover, state machines are a poor fit for modern CPU microarchitectures: their control flow depends on the input data and cannot be predicted by the branch predictor, leading to frequent branch mispredictions. DuckDB's [14] CSV processing logic attempts to mitigate this by repeatedly skipping blocks of bytes and suppressing state transitions when a block contains no special characters (such as the escape character, field delimiter, or record delimiter). However, with many small fields — a high ratio of structural characters — few blocks can be skipped, and the check itself may waste CPU cycles. Overall, state machines are not very *efficient* `−` on modern hardware.

On the other hand, an explicit state machine is maximally *flexible* `++`, allowing arbitrary CSV dialects to be parsed — Rust CSV, for example, never produces an error and always finds a valid parse through its DFA. FSMs are also very *usable* `++`, imposing no

[3] https://github.com/BurntSushi/rust-csv

Table 1: Qualitative comparison of CSV parsing approaches; csveee combines the scalability of parallel parsers with the efficiency and usability of sequential ones.

	Scalability	Efficiency	Flexibility	Usability
Sequential				
FSM	−−	−	++	++
Vectorized	−−	++	+	++
Parallel				
Multiple FSMs	++	−	++	−
Prefix Sums	++	++	−	−
Speculative	++	+	−	++
csveee	++	++	+	++

hardware or software requirements, and can significantly simplify development and maintenance of the parser code.

Vectorized Parsing. Vectorized parsers leverage SIMD instructions to process multiple bytes in a single CPU instruction. Rather than maintaining an explicit state machine, they build bitmasks and use bitwise operations to identify quoted regions, mask quoted fields, and construct field and record boundaries. State is tracked implicitly, for instance by flipping bitmasks for quoted fields.

This approach was popularized by Langdale and Lemire [6] for JSON parsing. For CSV, Mühlbauer et al. [10] demonstrate how SIMD instructions can be leveraged for fast delimiter matching. Without exploiting the parallelism of modern many-core CPUs, vectorized parsers do not *scale* −−, operating orders of magnitude below the achievable main memory bandwidth. However, vectorized parsing is substantially more *efficient* ++ than finite-state machines: it aligns well with modern CPU architectures by exploiting data-level parallelism, enabling branch-free execution, and producing predictable memory access patterns. Expressing complex parse state through bitmask operations is non-trivial, making it much harder to develop and maintain vectorized parsers, leading to parsers that are less *flexible* + in the input they accept than their state-machine counterparts. Portable SIMD intrinsics let compilers emit architecture-specific instructions for any platform, and even compile on rare hardware without SIMD support. Thus, vectorized parsers are very *usable* ++, just like FSMs.

2.2 Parallel Parsing

Parallel parsers split the file into chunks processed concurrently. As discussed above, the central challenge is resolving the parse state at chunk boundaries. Notably, two of the three approaches proposed in the literature exploit the massive parallelism of GPUs, and only one was explicitly designed for CPUs. While all proposed parallel parsers appear to be incomplete research prototypes, each one follows a different idea to resolve parse state at chunk boundaries. To our knowledge, no approach is used in real-world software.

Multiple Finite-State Machines. Stehle and Jacobsen [16] propose a GPU-based parsing approach called ParPaRaw. As in classical sequential CSV parsing, the core recognizer is a deterministic finite automaton (DFA). The CSV file is split into many small chunks (e.g., 32 bytes) that are processed in parallel by GPU threads. Since the correct starting state is unknown, each thread runs one DFA copy per possible state in parallel, producing a state-transition vector that maps each starting state to its final state. Once all chunks have been processed, a parallel prefix scan [8] over the state-transition vectors determines the correct starting state for each chunk. Threads then reprocess their chunks using the DFA with the now-known correct starting state to identify records and fields.

Unlike the sequential approach, running many FSM copies in parallel exploits the massive parallelism of modern GPUs, i.e., it *scales* ++ well. However, GPUs can be exploited more *efficiently* − as the following *Prefix Sums* approach demonstrates. As with sequential FSM-based parsing, the use of a state machine allows arbitrary CSV dialects to be parsed, and is thus very *flexible* ++. Using GPUs comes at the cost of *usability* −, as the approach requires dedicated hardware and software. Unless CSV data is already on the GPU, data transfer via the PCIe bus impairs efficiency and limits scalability to the bus bandwidth, which is typically lower than main memory bandwidth.

Prefix Sums. Kumaigorodski et al. [5] propose another GPU-based approach called CUDAFastCSV. The CSV file is split into larger chunks (e.g., 1,024 B), and for each chunk, GPU threads count field delimiters, record delimiters, and quotes in parallel. A parallel prefix scan then propagates these counts so that every thread knows the total number of delimiters and quotes preceding its chunk. In a second pass, threads construct an index of field offsets in parallel without synchronization, since each thread knows the number of preceding fields. Delimiters inside quoted fields — detected via odd/even quote parity — produce zero sentinel entries in the index, which are removed in a final compaction pass.

Just like the *Multiple FSMs* approach, this approach *scales* ++ well to the parallelism of GPUs. Notably, it is far less compute-intensive and thus more *efficient* ++ than the multiple FSMs approach, achieving throughput of more than 100 GB/s for CSV files residing in GPU memory. However, the parser cannot detect and unescape escaped quotes within quoted fields, passing them through verbatim. It is much less *flexible* − than the parallel FSM-based approach. Like the other GPU-based method, it is bounded by PCIe bandwidth for non-resident data and limited in *usability* − by specialized hard- and software.

Speculative Parsing. Ge et al. [4] propose a CPU-based parallel parsing approach. The CSV file is split into larger chunks (e.g., 64 KB), and each chunk is scanned by a thread searching for quote characters. If a quote preceded by a regular character (i.e., neither a quote nor a delimiter) is found, the thread concludes that this quote marks the end of a quoted field, meaning that the data before it is inside a quoted region. Conversely, if a quote succeeded by a regular character is found, the thread concludes that it marks the beginning of a quoted field. If neither pattern is found, the chunk is ambiguous, and the thread speculates whether the beginning of the chunk falls inside a quoted field by estimating which interpretation is more consistent with the CSV specification.

Once each thread has determined whether its chunk begins inside or outside a quoted field, it identifies boundaries: if the chunk is not in quotes, the first record delimiter following an even number of quotes is taken as the start of the first record; if the chunk is in quotes, the first record delimiter following an odd number of quotes is used instead. Each thread notes this position and the last record delimiter in its chunk. If the positions noted by all threads are consistent — that is, the first delimiter of one chunk coincides with the last delimiter of the preceding chunk — the chunks are adjusted to their respective record boundaries and parsed in parallel.

If speculation fails, the parser falls back to a deterministic two-pass approach that propagates quote counts across chunks to resolve boundaries.

While this approach *scales* ++ well to modern many-core CPUs, requiring a fallback path, though rarely triggered, and using a finite state machine is not very *efficient* −. It is also not very *flexible* −, as the parser requires files to follow RFC 4180 and cannot handle non-compliant dialects. However, the approach is very *usable* ++ as it does not require specialized hardware or software.

3 DESIGN

Existing approaches to CSV parsing are insufficient: sequential parsers do not scale, while the parallel parsers proposed in the literature either rely on specialized GPU hardware or impose overly strict assumptions on the input — and all are incomplete research prototypes lacking essential functionality such as UTF-8 validation.

In this paper, we propose a new parser that combines the scalability of speculative parsing with the efficiency, flexibility, and usability of vectorized parsing. Unlike the approaches presented in the previous section, we aim to parse and process files in a single pass to allow for file processing close to the memory bandwidth — without a fallback mechanism for edge cases, which are prone to failure as they are harder to test and rarely exercised in practice.

3.1 Parallelism

Commonly used CSV parsers are very permissive. They follow the TCP robustness principle [13] as recommended by RFC 4180 [15]: "be conservative in what you do, be liberal in what you accept from others". Taken to its extreme, some parsers such as Rust CSV will always find a valid parse, i.e., at least one record if the input is not empty. Whether this is desirable is debatable. Regardless, liberal parsing of CSV files makes parallelization challenging: if valid records can be constructed from any or many offsets in a file, the parsing state at a chunk's start can only be determined once the previous chunk's final state is known. Avoiding this sequential dependency would require parsing each chunk in all possible parser states, which is computationally expensive.

In practice, however, applications commonly expect CSV files to follow a specific structure — fields of particular data types, a fixed number of fields per record, etc. This user-provided information on the file structure can be used by a parser as an oracle to determine the parser state at the beginning of a chunk.

To our knowledge, this strategy has not been formally described in the literature. The idea is simple: the parser speculatively parses each chunk in a default initial state, using the oracle to validate records. If validation fails — for example, a field cannot be parsed as the expected type or a record has the wrong number of fields — the parser retries the chunk in another state. If no state succeeds, an error is reported.

The key advantage is that this allows for parallel parsing with almost no inter-thread communication. The downside is that a chunk may — although unlikely — be parsed incorrectly. Note that this constitutes a performance not a correctness issue, as the parser can validate in the end that the speculation was successful, for example by verifying alignment of record boundaries across chunks. The higher the oracle quality, i.e., the more information the oracle provides, the lower the likelihood of parsing chunks in the wrong state. DuckDB follows this idea in its CSV reader[4], leveraging schema information such as data types and column counts; we extend it to general-purpose CSV parsers.

Unlike the speculative approach of Ge et al. [4], files need not be RFC-compliant, and chunks can be reparsed in the unlikely case of misparsing, avoiding a slow fallback. Furthermore, the oracle allows combining parsing and processing into a single pass, as formalized in the next subsection.

[4]Observed in DuckDB's source code; to our knowledge, the strategy is not described in any publication.

3.2 Programming Model

CSV parsers typically follow the iterator pattern [7]: they return an iterator over records, parsing the file lazily by advancing through the input only when `next` is called. While iterators can also yield pre-parsed records — possibly generated in parallel — and even be turned into parallel iterators, they cannot combine parallel parsing and processing into one pass over the data due to the inherent uncertainty of parsing at arbitrary offsets. To address this limitation, we propose a new programming model inspired by user-defined aggregates (UDAs) [20] in database systems such as PostgreSQL's `CREATE AGGREGATE`. It can also be viewed as a variant of MapReduce [2] with the added guarantee of ordered combination.

Interface. The parser exposes a single entry point,

$$\textsc{Parse}(F, f_{\text{init}}, f_{\text{acc}}, f_{\text{merge}}) : T$$

parameterized by:

- a file F containing m records,
- a triple of user-supplied functions:

$$f_{\text{init}} : 1 \to S$$
$$f_{\text{acc}} : S \times R \rightharpoonup S$$
$$f_{\text{merge}} : S^* \to T$$

  where 1 is the unit type, S is the type of a chunk-local accumulator, R is a record, and T is the result type. The accumulation function f_{acc} is a partial function ($\rightharpoonup$): it may reject a record by signaling an error $\varepsilon \in \mathcal{E}$.

Semantics. The system parses in two phases: a parallel *speculative parsing* phase that parses each chunk independently under competing assumptions, and a sequential *merge phase* that reconciles the chunk-local results. Let $F = (r_1, r_2, \ldots, r_m)$ be the ordered sequence of records in the file. The system divides F into k contiguous, non-overlapping byte-aligned chunks $B_1, B_2, \ldots, B_k$. A record r is *owned* by chunk B_i if the first byte of r falls within B_i, regardless of where r ends; each chunk's parser reads past the chunk boundary as needed to complete the last owned record. A chunk may own no records (e.g., if B_i falls entirely within a single long record).

Let $\mathcal{A} = (a_1, \ldots, a_l)$ be the finite, priority-ordered sequence of *assumptions* about the parse state at the start of a chunk; a lower index denotes higher priority. Since we locate a chunk's first record by scanning forward, each assumption is an equivalence class of parse states — those implying the same first-record position, and hence the same boundaries — so l is typically smaller than the number of states of the underlying parser. For each chunk B_i and each assumption $a_t \in \mathcal{A}$, let $\text{parse}(B_i, a_t)$ denote the record sequence produced by the parser under a_t. The system computes:

$$s_i^{(a_t)} = \text{foldl}(f_{\text{acc}},\ f_{\text{init}}(),\ \text{parse}(B_i, a_t))$$

where foldl propagates errors: if $f_{\text{acc}}(s, r_j) = \varepsilon$ for some record r_j produced by $\text{parse}(B_i, a_t)$, then $s_i^{(a_t)} = \varepsilon$ and no subsequent records are processed. The system evaluates assumptions in priority order $a_1, a_2, \ldots, a_l$, stopping at the first assumption a_t that yields records with $s_i^{(a_t)} \neq \varepsilon$. The system retains each attempted assumption's

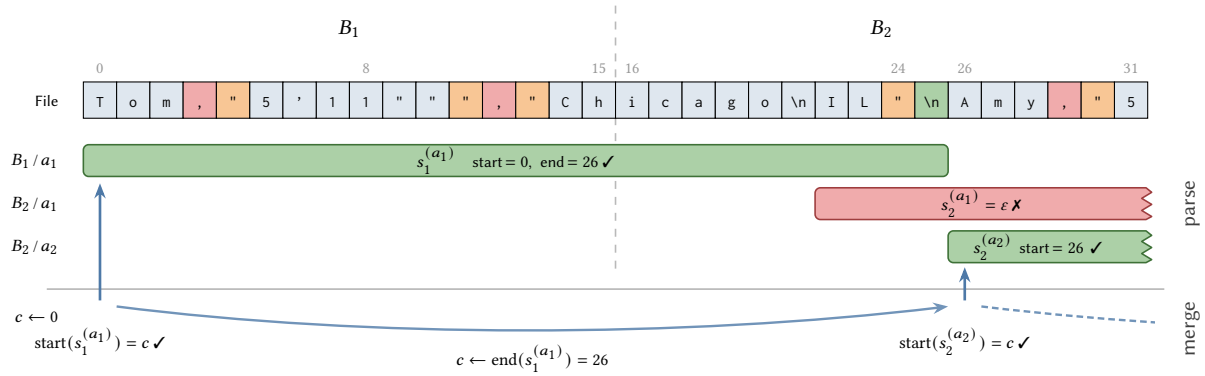


Figure 3: Speculative parsing and the merge phase. Green and red bars denote speculative results accepted and rejected (ε).

result, state or error; the merge phase then selects the authoritative result by record boundaries.

The Dual Role of f_{acc}. The partiality of f_{acc} plays a dual role:

(1) *User-level validation.* The caller may reject records that do not conform to application-level expectations — for instance, a field that cannot be parsed as a numeric type.
(2) *Speculative disambiguation.* The parser may initially misidentify records — for example, by interpreting bytes inside a quoted field as field delimiters. When such a misparse produces a record that f_{acc} rejects, the system interprets this rejection as evidence that the current parsing assumption a_t is incorrect. The system then retries the chunk from the beginning under the next assumption a_{t+1}.

The user-supplied f_{acc} thus acts as a *validation oracle*: it confirms or refutes a speculative parse. Crucially, this disambiguation is chunk-local — it requires no cross-chunk communication during the parallel phase. Figure 3 shows this on the first two chunks B_1, B_2 of a file. The boundary between them falls inside a quoted field, so for B_2 the higher-priority assumption a_1 (not in quotes) misreads the field's embedded newline as a record terminator; the resulting record does not meet the application's expectations (number of fields, types, etc.), so f_{acc} rejects it (ε) and the system falls back to a_2 (in quotes).

Speculative Parsing. The system dynamically assigns the k chunks to n worker threads with $1 \le n \le k$. For each chunk B_i, the assigned thread tries the assumptions $a_t \in \mathcal{A}$ in priority order until one succeeds ($s_i^{(a_t)} \neq \varepsilon$) or all are exhausted. For each, it:

(1) invokes $f_{\text{init}}()$ to create a fresh accumulator,
(2) folds the chunk's records under a_t into it via f_{acc}, and
(3) stores $s_i^{(a_t)}$ — together with the first record's absolute file position and the position immediately following the last record — in a chunk-indexed result array.

Merge Phase. After all chunks have been processed, a sequential merge phase iterates through the result array in chunk order, maintaining a cursor c that tracks the byte position immediately following the last record selected so far (initialized to the position of the first record in the first chunk, if any). For each chunk B_i, the system selects the correct speculative result by finding the assumption $a_t \in \mathcal{A}$ whose first record begins at c (Figure 3). Since records may span multiple chunks, the cursor may skip over chunks entirely: If c is beyond the byte range of B_i, the chunk does not own any records and is skipped without advancing the cursor. When a match is found, the cursor advances to the end of the last record produced under the selected assumption.

Formally, let start($s_i^{(a_t)}$) and end($s_i^{(a_t)}$) denote the file positions of the first and last record boundaries under assumption a_t. Note that end($s_i^{(a_t)}$) may extend beyond the byte range of B_i, since the last owned record can span into subsequent chunks. The system selects:

$$s_i = s_i^{(a_t)} \quad \text{where} \quad \text{start}(s_i^{(a_t)}) = c$$

and updates $c \leftarrow \text{end}(s_i^{(a_t)})$. In Figure 3, the merge starts with the cursor at the first record of B_1, sets $s_1 \leftarrow s_1^{(a_1)}$, and advances c to its end; that position coincides with the start of B_2's parse under a_2, so the merge sets $s_2 \leftarrow s_2^{(a_2)}$ and discards $s_2^{(a_1)}$. If the selected assumption for a chunk produced an error, the merge phase can now report it. If no assumption for B_i starts at c although the cursor is in the byte range of the chunk, speculative parsing of the chunk ended too early due to a successful assumption (false positive), and the chunk needs to be reparsed from the current offset. Note that this is highly unlikely as we show in our evaluation in Section 5.

After the merge phase selects the valid assumption for each chunk that owns records, skipping those that own none, the surviving states form an ordered subsequence $(s_1, s_2, \ldots, s_p)$ with $p \le k$. The merge function then receives the validated states, and the parser returns the result of the merge function:

$$t = \begin{cases} f_{\text{merge}}(s_1, s_2, \ldots, s_p) & \text{if all selected states are successful} \\ \varepsilon_j & \text{otherwise, for the least } j \text{ where } s_j = \varepsilon \end{cases}$$

Since f_{merge} receives the states in chunk order, the model supports order-sensitive computations — including both non-commutative aggregations (e.g., running statistics) and operations that preserve the full input.

This two-phase design — parallel speculative parsing followed by sequential boundary validation — achieves full parallelism during the dominant computational phase while confining the inherently sequential disambiguation to a lightweight merge step.

4 IMPLEMENTATION

We implement the parser in Rust for its performance, memory and thread safety, and portable SIMD support.

4.1 Parallelism

We implement work-stealing for parallelization, ensuring load balancing without centralized scheduling. Let $C = \lceil F/B \rceil$ be the number of chunks for a file of size F with chunk size B, and let H be the number of hardware threads. The parser spawns worker threads such that the total thread count, including the main thread, is $T = \min(H,\ C,\ 6\lfloor\sqrt{C}\rfloor)$. The $\sqrt{C}$ term caps thread count for medium-sized files ($36 < C < H^2/36$), where we observe diminishing returns on machines with hundreds of threads; we picked the constant factor (6) empirically to match observed throughput optima. A user-specified limit is still respected as an upper bound.

Each thread is assigned an initial chunk at startup to minimize synchronization overhead. Threads, including the main thread, then process their chunks as described in Subsection 3.2, retrying in alternative parser states if needed. Once a thread finishes processing a chunk, it claims the next unprocessed chunk via a shared atomic counter. Since chunks are equal in size (1 MiB by default), the counter suffices to determine each chunk's byte boundaries.

After all chunks have been processed, the main thread verifies that the record boundaries of chunks align and merges the results as described in Subsection 3.2. In the highly unlikely event of a false positive (Subsection 3.2), the affected chunk would need to be rescheduled and revalidated; so far we have not implemented this logic yet; the case never occurred across any of the files in our evaluation.

4.2 Programming Model

We mirror the programming model from Subsection 3.2 directly in the API. The parser exposes a single function `parse`($file$, f_{init}, f_{acc}, f_{merge}) that accepts three user-defined closures as shown in Figure 4. The field count per record is fixed at compile time via a const generic `N`. This is a deliberate implementation choice, not a limitation of the design — close to 99% of datasets in our evaluation (Section 5) have a uniform field count; it matches the default of Rust CSV and is required by Postgres' `COPY`; fixing `N` enables cheap index and record construction (Subsection 4.3) and adds discriminating power to the speculative oracle to cheaply determine parse state.

```
let tallest = Parser::new().parse(
  "census.csv",
  // init
  || { 0 },
  // accumulate
  |acc, [name, height, city]| {
    Ok(*acc = (*acc).max(height.parse()?))
  },
  // merge
  |accs| { accs.iter().max().cloned() }
)?;
```

Figure 4: Determining a maximum in a CSV file.

For performance, distinct per-thread parse functions are compiled for each CSV dialect and selected at runtime. Errors in f_{acc} may be propagated through the parser's return type. To demonstrate that our programming model is strictly more expressive than the iterator model, we also provide a `records` function that desugars into a call to `parse`, returning an iterator over records. While the iterator defeats the idea of inline data processing, the function often outperforms sequential parsers due to the parallel record aggregation while providing a familiar interface.

4.3 Vectorization

Inspired by simdjson [6], we apply vectorization end-to-end, extending the SIMD delimiter matching of Mühlbauer et al. [10] to full structural character classification and bitmask-based record construction (Subsection 4.4).

Overview. A worker thread parses its chunk under the first assumed parse state; we try out-of-quotes before in-quotes, as it is the more likely state for most files. The chunk is read piece-by-piece from its offset into a thread-local I/O ring buffer (Subsection 4.5), and we iterate over 64 B vectors — one CPU cacheline per step on most systems. Per-chunk parsing consists of three phases:

(1) *Prologue.* We find the first record in the chunk based on the current parse state. To prevent unpromising long runs, we default to strict (RFC) parsing and fall back to lenient parsing for record detection if none is found.
(2) *Core.* A record has been encountered and we are parsing in the interior of the chunk; this is the main phase of parsing, which we explain in the next paragraph in more detail.
(3) *Epilogue.* Once we reach the end of the chunk, we continue parsing records but stop at the next record delimiter.

Bitmask Construction. We describe the main phase for the default configuration (quotes enabled, escaped with quotes, newlines as record terminators), starting with a single vector and extending to subsequent vectors later. Assume we have the following data:

```
Tom,"5'11""",Chicago\nIL"\nAmy,"5
```

The parser must determine two things: field boundaries, and characters to remove (e.g., escape characters). This requires classifying structural characters correctly — distinguishing field and record delimiters and quotes marking quoted fields from the commas, newlines, and quotes that appear as data within them:

```
Tom,"5'11""",Chicago\nIL"\nAmy,"5
```

Since every record has the same number of fields, we do not need to keep track of record boundaries. We therefore derive three bitmasks from the input data: a bitmask of field beginnings (B), a bitmask of field ends (E), and a bitmask of characters to remove (R):

```
  T o m , " 5 ' 1 1 " " " , " C h i c a g o \n I L " \n A
B 1 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1
E 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0
R 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
```

Bits in B mark the first character in a field; bits in E mark the character after the last, i.e., the delimiter or terminator. This asymmetry ensures that the end index is always at or after the beginning.

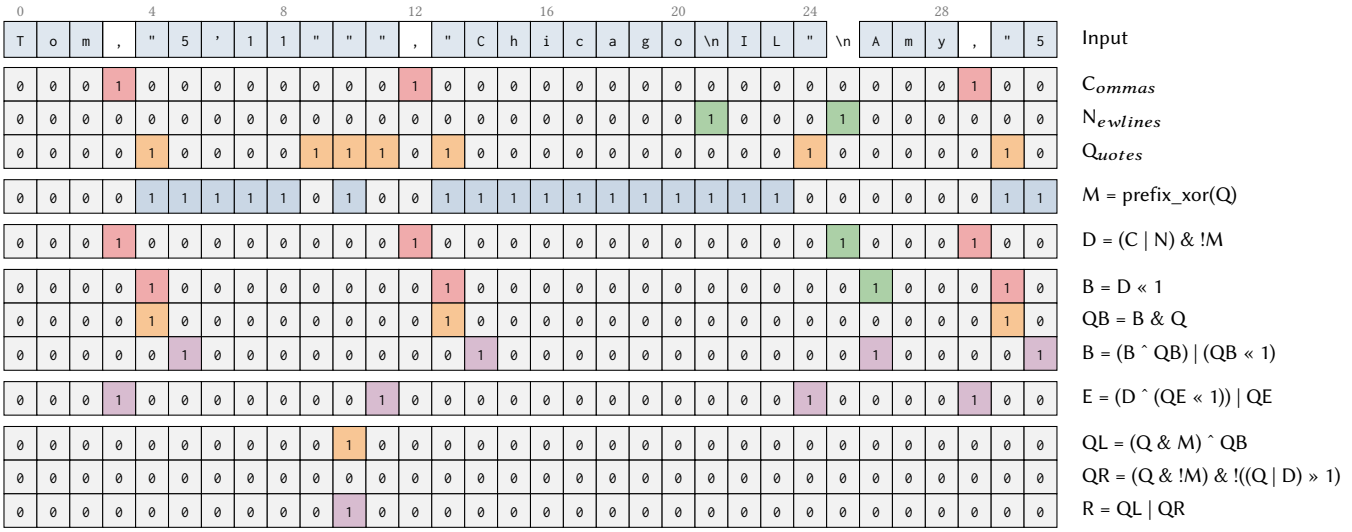


Figure 5: Computing bitmasks for field beginnings (B), field ends (E), and characters to remove (R). Bit 0 is on the left to align with input byte 0; left shifts therefore appear as rightward motion. Input is 64 byte and bitmasks are 64 bit in our implementation.

For quoted fields, B marks the character after the opening quote and E the closing quote, avoiding separate character removal for the common case. Figure 5 shows how we compute the three bitmasks.

Before matching any structural characters, we ensure UTF-8 validation (Subsection 4.6). We match the characters configured for field delimiter, record terminator, and quotes, yielding three bitmasks (C, N, Q) where a set bit indicates the presence of the respective character. We compute a prefix sum of Q under XOR, as described by Langdale and Lemire [6], using the carry-less multiplication instruction (`pclmulqdq`), to obtain a bitmask M where all positions inside quotes are set to 1 (including the opening quote) and all positions outside quotes are set to 0. The carry-less multiplication is the most expensive instruction in the pipeline at 5-cycle latency on Zen 5, but is required only once per 64 B vector. We use M to distinguish whether matched field delimiters and record terminators are part of a quoted field or are actual structural characters. We validate that records have the expected number of fields.

We compute the field beginnings bitmask B in three steps: first, we left-shift the field delimiters and record separators by one to obtain an initial B; then, we compute QB, the quotes at field beginnings; finally, we recompute B, left-shifting bits matching QB.

Similarly, we compute the field ends bitmask E: we take the field delimiters and record separators as an initial E, compute QE, the quotes at field ends, and recompute E, removing bits matching QE left-shifted by one, and adding the QE bits as field ends.

Lastly, we compute the bitmask of characters to remove. To support real-world, non-RFC-compliant CSVs, we must handle cases such as quotes appearing in unquoted fields:

```
John , say " hello " world , 42
```

Here, the second field contains quotes but is not itself a quoted field. We follow Postgres' logic and treat fields as collections of quoted and unquoted parts that are stitched together before being passed to the user. Unfortunately, with the CSV quote escape mechanism, we cannot simply remove all quotes inside of fields. To determine which quotes to remove, we use M to split the quotes into two subsets: quotes at the beginning of in-quotes sequences and quotes at the end of in-quotes sequences.

We remove all quotes at the beginning of in-quotes sequences unless they are quotes at the beginning of a field, since those have already been accounted for by the field beginnings bitmask. We remove quotes at the end of in-quotes sequences *if they are not followed by another quote*, unless they are quotes at the end of a field, since those have already been accounted for by the field ends bitmask. In this manner, we keep one of the two quotes of escaped-quote sequences. QL determines the first set of quotes, QR the latter; we combine both into R, the bitmask of characters to remove.

Once we have collected a certain number of field beginnings, endings, and character removal bitmasks, we assemble the fields (see Subsection 4.4), call the user-provided closure, and consume data from the I/O buffer. If there are no complete records to process once we reach the last vector of the I/O buffer, then the current record exceeds our buffer in size, and we grow the I/O buffer.

Counting Fields. To ensure correct field counts, we use a parallel bit extract on the bitmask of field delimiters and record terminators D to obtain a bitmask P where every 0 represents a field delimiter and every 1 a record terminator (Figure 6).

For up to 63 fields per record, we verify P against a compile-time lookup table of bitwise rotations of the expected pattern, selecting the correct rotation based on a `popcnt` of D. We mask the pattern to cover only the fields in the current vector and XOR it with P; any set bits indicate a missing or misplaced record terminator.

For 64 or more fields, a 64 B vector contains at most one record terminator, so we verify the field count directly via `popcnt` and count-trailing-zeroes on P.

Both `pext` and `popcnt` are fast single-µop instructions; on Zen 5, `pext` has a 3-cycle latency and `popcnt` executes in a single cycle.

Vector Dependencies. Unsurprisingly, we have to carry parse state from one vector to another. Figure 8 shows the data dependencies

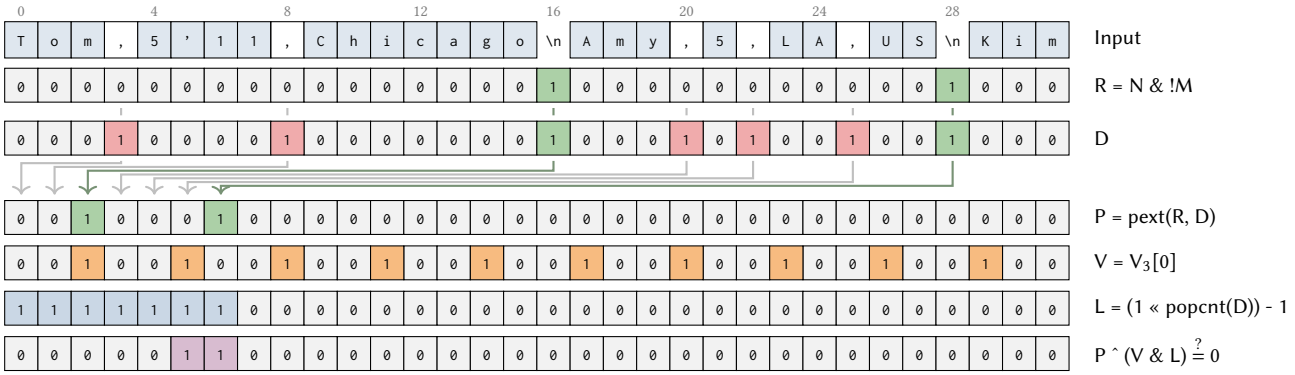


Figure 6: Detecting records with an invalid number of fields.

between bitmasks of consecutive vectors (character removal bitmasks are omitted for space). For example, if M of vector i ends with a set bit, all bits of M in vector $i+1$ must be flipped. Similarly, since B is computed by left-shifting delimiters by one, B of vector i depends on the delimiters of the previous vector, while E of the next vector $i+1$ extends into vector i.

We resolve this mutual dependency by first computing all bitmasks of vector i that depend only on previous vectors, then doing the same for vector $i+1$, and finally completing the bitmasks of vector i using the now-available data from vector $i+1$.

Parse Configurations. As explained in Subsection 4.2, we compile specialized parse functions for each configuration. If quotes are disabled, for example, quotes are not matched, no prefix XOR needs to be computed, field ends are simply the field and record delimiters, field beginnings are computed by a simple left shift, and no bitmask for characters to remove needs to be computed. Different one-byte characters for commas, newlines, and quotes may be configured for any configuration, though they must be distinct.

We also support CRLF line endings. When CRLF is used, we additionally match the carriage return character and verify that every LF is preceded by a CR. We use the CRs to determine field ends and the LFs to determine field beginnings, again handling this across vector boundaries.

Backslashes or other escape symbols require an entirely different mechanism: instead of a prefix sum, we classify each backslash as part of an even- or odd-length run [6]. Backslashes preceding quotes in odd-length runs are removed from Q before computing M; in the characters-to-remove mask, we mark alternating backslashes in each run for removal.

Portability. We use Rust's portable SIMD instructions and operate primarily on 64-bit integers, which are available on all major architectures. For the three x86-specific instructions we use (`pdep`, `pext`, and carry-less multiplication), we provide fallback implementations; our code compiles and runs on both x86 and ARM.

4.4 Index Construction and Record Processing

We convert each bitmask into a sequence of offsets into the buffered reader by extracting set-bit positions. Unlike simdjson's unrolled loop [6], we find that a simple scalar extraction loop performs better for the sparse bitmasks typical of CSV (≤ 7 bits per vector).

We then construct string slices by iterating over pairs of begin and end offsets. For each field, we check whether any offsets in the remove index fall within the field boundaries; if so, we shift each segment between consecutive characters to remove (or between the last character to remove and the field end) forward by the number of preceding removals, copying every byte at most once. This works well for the common case of sparse escape characters.

Once all fields of a record are constructed, we invoke the user-provided accumulation function, then continue with the next record until no complete records remain. Finally, the processed data is consumed from the buffered reader.

4.5 I/O

While `mmap` may be convenient for reading files, it does not scale on modern many-core systems [1]. We benchmark parallel chunked reading of a 10 GB file end-to-end to confirm this claim, analogous to the I/O path in the parser: with `read`, each thread reads its chunk into a thread-local buffer and achieves 183 GB/s on our evaluation server, while a single `mmap` mapping partitioned across threads maxes out at 34 GB/s — likely due to page table setup and teardown overhead (e.g., TLB shootdowns); prefaulting with `MAP_POPULATE` lowers throughput further.

Based on this insight, we implement a buffered reader that reuses the same memory pages throughout parsing, achieving the throughput of `read` while retaining desirable properties of `mmap`:

(1) it is cache-line aligned for convenient processing of cache-line-aligned vectors of data,
(2) allows for in-place updates to remove characters such as escape characters, and

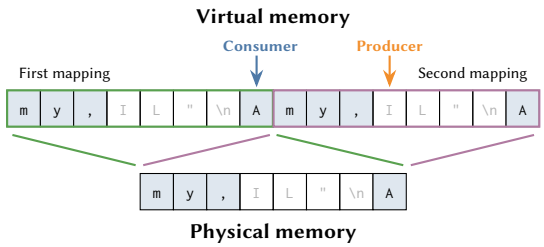


Figure 7: Virtually contiguous ring buffer with CSV data.

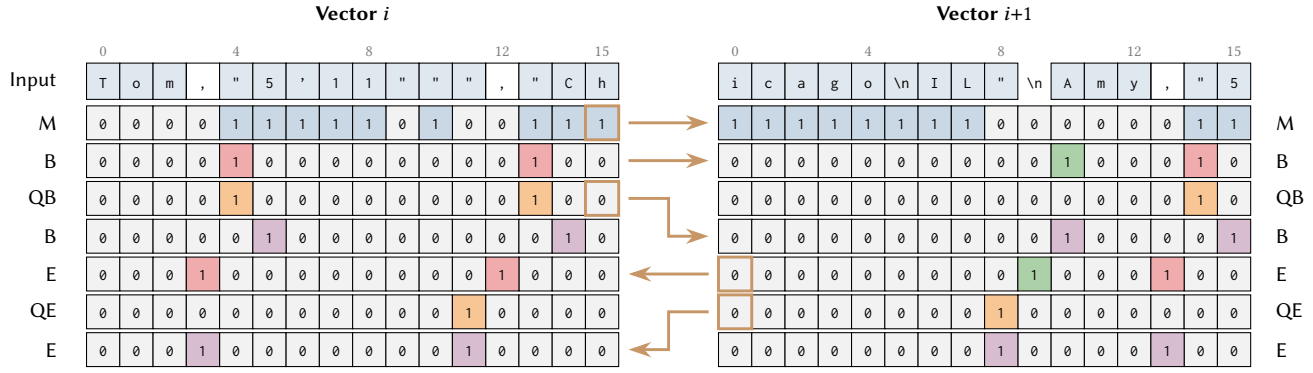


Figure 8: Data dependency (⟶) between consecutive vectors. Field beginnings depend on structural characters of the previous vector. Conversely, field ends depend on structural characters of the next vector. Vector size reduced to 16 B for clarity.

(3) is always contiguous in memory even at buffer boundaries such that records do not have to be copied.

We implement this buffered reader using a ring buffer as its internal buffer. Figure 7 shows the buffer design. The ring buffer is realized efficiently by mapping its internal buffer twice in contiguous memory, such that the end of the buffer is immediately followed by its beginning. We track a producer and a consumer offset, incremented by the number of bytes written or read modulo the buffer size. By using a power-of-two buffer size, the modulo reduces to a bitwise AND.

This makes it impossible to distinguish empty from full buffers, as both have equal offsets. We avoid extra state by leveraging unsigned integer wrap-around: offsets are incremented without reduction and taken modulo the buffer size only when accessing data. Since the maximum buffer size is half the range of the unsigned integer type, producer and consumer offsets are never equal for a full buffer.

Note that our synchronous I/O path is tuned for local storage; network-attached storage instead might benefit from larger reads and asynchronous I/O that overlaps fetching with parsing.

4.6 UTF-8 Validation

All data is UTF-8 validated. We validate as early as possible; however, since a chunk may start in the middle of a multibyte character, we do not validate before its first record boundary. Since record boundaries align across chunks, all data is validated once the merge phase succeeds: the bytes preceding a chunk's first record are validated by the previous chunk's parser, whose final record extends into this region. Skipped header fields are neither parsed nor exposed, and therefore not validated. For efficiency, we adopt simdjson's [6] vectorized approach, validating cache-line-aligned blocks corresponding to our 64-byte vectors, and abort as soon as invalid data is encountered. We evaluate the performance impact in Section 5.

5 EVALUATION

Hardware. We perform our evaluation using two systems, an AMD EPYC server for CPU workloads, and a workstation with a NVIDIA RTX GPU for GPU workloads. Both represent current hardware with the AMD CPU released in Q4 2024 and the NVIDIA GPU in Q1 2025. We summarize the characteristics of the evaluation hardware in Table 2.

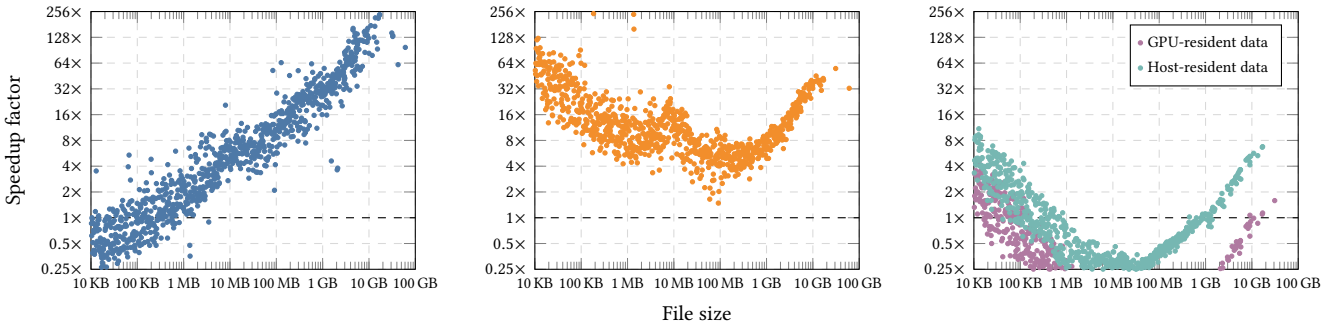
Software. The server runs Ubuntu 24.04, the workstation Ubuntu 25.10. We use CUDAFastCSV (commit 4a43b8b, gcc 15.2 / nvcc 12.4.131), Umbra (v0.2-1645-g497e39f9e, gcc 15.1), DuckDB 1.4.4, and Rust CSV 1.4.0, with our Rust parser built on rustc 1.95.0-nightly. All software was compiled in release with native optimizations for the CPU parsers. We verified our parser's output against unit tests and CSV files from the Rust CSV, Postgres, and DuckDB repositories.

Datasets. Parsing speed depends heavily on the characteristics of the input. A fair evaluation hence requires a wide range of CSV files varying in shape and size. We therefore evaluate our parser on both real-world files and synthetic files commonly used in the database community. For real-world data, we use CSV files sourced from the Kaggle[5] data science platform. We collect 21,929 CSV files from Kaggle across 9,549 datasets, totaling nearly 1 TB of CSV data. Large files are rare; to avoid a bias toward small files we include all 48 files of at least 5 GB — the largest of which is 60.8 GB — and draw a log-uniform sample from the remaining 18,615 files in the range of 10 KB to 5 GB, yielding 1,000 CSV files in total. Despite Kaggle's framing as a data-science platform, these 1,000 files are messy: 27 contain invalid UTF-8, 17 do not have a fixed field count per record. The median number of columns is 12, the maximum 64,000 (in a file with only a handful of records); 97% use commas as field delimiters; 55% contain no quotes; and line endings are split between LF (58%) and CRLF (42%), with one file using bare CRs.

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

Table 2: Hardware configurations used in the evaluation.

System	Processor	Cores / Threads	Base / Max. Freq.	Memory	TFLOPS	Interconnect
Server	AMD EPYC 9745	128 / 256	2.4 / 3.7 GHz	1,024 GB DDR5	–	–
Workstation	NVIDIA RTX 5080	10752 / –	2.3 / 2.6 GHz	16 GB GDDR7	171	PCIe 4.0 x16 (~32 GB/s)



(a) vs. Rust CSV **(b) vs. DuckDB** **(c) vs. CUDAFastCSV**

Figure 9: Speedup of csveee compared to Rust CSV, DuckDB and CUDAFastCSV across 1,000 CSV files of varying sizes; CUDAFastCSV with GPU-resident and host-resident data.

For synthetic data, we use the lineitem table at scale factor 100 from the TPC-H benchmark [17], which has a total size of 79 GB, as well as eight hand-crafted CSV files for micro-benchmarking, shown in Table 3.

Performance. As baseline for parse performance, we measure the upper limit for parse throughput on the server, the memory bandwidth. We write a small program based on a memory throughput measurement approach by Lemire[6]. Every thread allocates 256 MiB of data, synchronizes with the other threads through a barrier, and then reads its data sequentially cacheline for cacheline, performing a sum of values for one 8 B value per 64 B cacheline. We determine a memory bandwidth of 240 GB/s on the server.

5.1 Performance

We compare the throughput of csveee against Rust CSV, DuckDB, and CUDAFastCSV in Figure 9, quantifying the difference as speedup. Since parsing speed depends heavily on how parsed data is processed (see Subsection 5.5), we opt to count records with all parsers rather than performing complex record processing. We use the 1,000 files from the Kaggle dataset for our throughput measurements.

We also considered zsv[7] (v1.3.0), but its parallel mode miscounts records on quoted inputs: on a 3,000-row Project Gutenberg file, `zsv count --parallel` reports 26,834 rows, whereas `zsv count` and csveee both return the correct 3,000. We therefore exclude zsv from our measurements, despite its otherwise high throughput (~125 GB/s on TPC-H `lineitem` at SF100).

To determine CSV file properties — field delimiter, newline convention, quote and escape characters, and whether quoting is enabled — we use the DuckDB CSV sniffer and configure all parsers accordingly. For files that cannot be sniffed, we supply a default configuration with standard symbols and quotes enabled. For the DuckDB CSV reader, we supply the sniffer-generated prompt including the schema. For CUDAFastCSV, we measure kernel setup time, runtime, and data transfer time, and report results both including data transfer (host-resident data) and excluding it (GPU-resident data); we also configure it to skip its default columnar-tape materialization and only report the number of records found. All parsers perform one warmup run followed by ten measured runs, and we report the average parse time over the measured runs. The warmup run is particularly beneficial for CUDAFastCSV due to the high GPU setup costs.

The parsers reject differing subsets of the 1,000 files. Rust CSV rejects 35 files: 18 due to UTF-8 errors and 17 due to records with inconsistent field counts. DuckDB rejects 52 files due to inconsistent field counts (21), sniffer errors (12), parser limitations (8), line size limits (6), and UTF-8 errors (5). CUDAFastCSV rejects 250 files: 194 have more columns than the 32 supported ones, for 50 we get memory / buffer size errors, and for 6 we could not determine the config with the DuckDB sniffer because of invalid JSON output of DuckDB. Notably, CUDAFastCSV does not reject any files due to invalid UTF-8, as it lacks UTF-8 validation entirely. csveee rejects

[6] https://lemire.me/blog/2024/01/13/estimating-your-memory-bandwidth/
[7] https://github.com/liquidaty/zsv

Table 3: Synthetic datasets for micro-benchmarking CSV parser stages. Each file is 100 MB. *Quoted* implies all fields are quoted / not quoted, *Escapes* denotes the fraction of characters within quoted fields that are escaped quotes.

Dataset	Encoding	Cols	Field size	Quoted	Escapes	Primary stress
`plain-small`	ASCII	50	5–10 B	✗	—	Index construction
`plain-large`	ASCII	4	1–8 KB	✗	—	Core SIMD scanning
`quoted-small`	ASCII	50	5–10 B	✓	—	Quote handling + index
`quoted-large`	ASCII	4	1–8 KB	✓	—	Quote handling + scanning
`escaped-sparse`	ASCII	20	50–200 B	✓	1%	Light escape overhead
`escaped-dense`	ASCII	20	50–200 B	✓	5%	Field construction
`utf8-plain`	UTF-8	20	50–200 B	✗	—	Encoding effect
`utf8-escaped`	UTF-8	20	50–200 B	✓	5%	Worst-case fields

45 files in total: 26 due to invalid UTF-8, 13 due to records with inconsistent field counts, and 6 due to malformed line endings.

Figure 9a shows the speedup of csveee compared to the sequential Rust CSV parser. For small files of up to 100 KB, Rust CSV clearly outperforms csveee due to the relatively high memory setup costs for our data structures, such as the I/O buffers. From 1 MB upward, the effects of parallelism become apparent, leading to a maximum speedup of nearly 256× for files exceeding 10 GB. While the slowdown for small files may seem concerning, parsing those files takes less than 0.5 ms with both parsers. In contrast, parsing a 10 GB file finishes in under 0.1 s with csveee, whereas Rust CSV requires approximately 15 s — a substantial difference in practice.

Figure 9b shows the speedup compared to the parallel DuckDB CSV parser. For small files, per-query overhead dominates parsing in DuckDB. Notably, the slowest speedup is in the range from 100 MB to 1 GB. In this range DuckDB reaches its peak throughput while csveee just begins to scale out.

Figure 9c depicts the speedup compared to CUDAFastCSV. For small files, kernel setup time dominates throughput in both the GPU-resident and host-resident configurations. For files between 1 MB and 1 GB, CUDAFastCSV outperforms csveee due to the high on-GPU throughput, even when accounting for host-to-device data transfer. For larger files, however, the PCIe bus limits parse throughput to roughly 25 GB/s, making the host-resident configuration unattractive; for GPU-resident data, csveee is nearly on par owing to the scalability and efficiency of our implementation.

The comparison with CUDAFastCSV should be treated with caution, though, as this parser lacks important features such as unescaping escaped quotes and UTF-8 validation. Nevertheless, the results hint at what throughput might be achievable on a GPU.

5.2 Scalability

Figure 10 shows the throughput of csveee on the lineitem dataset with a varying number of hardware threads. Again, we count the number of records in the dataset. We evaluate two configurations: *Quoted* mode, the default configuration of the parser, in which quoting and escaping are enabled, and *Unquoted* mode, in which quote handling is explicitly disabled.

On a single thread, csveee achieves 1.8 GB/s in quoted mode and 2.1 GB/s in unquoted mode, 3× the throughput of single-threaded Rust CSV at 0.7 GB/s. The dashed line in the plot indicates ideal linear scaling based on the single-thread unquoted throughput.

Both configurations exhibit near-linear scaling up to 64 physical cores, with smooth degradation between 64 and 128 cores. Beyond 128 cores, the shaded SMT region shows that hyperthreading yields diminishing or even slightly negative returns, indicating that the

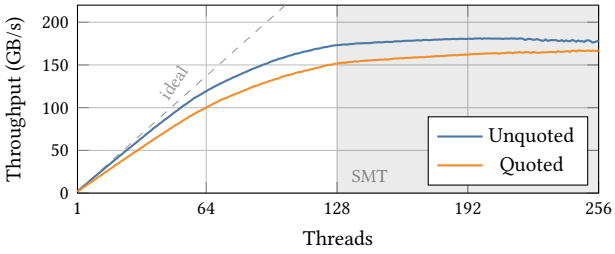


Figure 10: Throughput varying the number of threads.

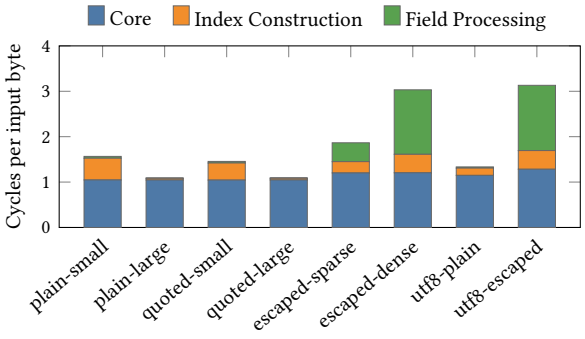


Figure 11: Cycles per input byte and processing phase.

bottleneck shifts from compute to memory bandwidth. Performance peaks at 181 GB/s in unquoted mode (198 threads) and 167 GB/s in quoted mode (256 threads).

5.3 Time Distribution

Figure 11 shows the distribution of CPU cycles across three processing phases of the parser when counting records, normalized to input size. *Core* encompasses the vectorization logic described in Subsection 4.3: character matching, UTF-8 and field count validation, and computing bitmasks for field beginnings, ends, and characters to remove. *Index Construction* turns these bitmasks into indexes into our buffered reader. *Field Processing* covers field construction, escape character removal, and invocation of the user-provided callback. We measure on eight synthetically crafted files with varying characteristics designed to stress different parts of the parser (Table 3).

Across all files, the core vectorization logic costs roughly 1 cycle per input byte and remains nearly constant, confirming that the SIMD pipeline is largely data-independent. Once the index has been constructed, field processing is essentially free as long as no characters need to be removed. Files with few columns, large fields, and no escaping such as `plain-large` represent the best case: index construction is cheap because the index is sparse relative to the input. For files with 5–10 fields per vector (`quoted-small`, `plain-small`), index construction rises significantly to around 0.5 cycles per byte. Perhaps surprisingly, `quoted-small` requires slightly fewer cycles than `plain-small`: with near-constant core cost and no escape characters, the quoted file simply contains around 20% fewer records and therefore fewer fields relative to its size, reducing both index construction and field processing overhead. `escaped-sparse` and `escaped-dense` demonstrate the cost of escape handling: index construction increases and field processing becomes substantially more expensive, especially for `escaped-dense` with 2–10 escaped characters per field. Finally, the negligible difference between `escaped-dense` and `utf8-escaped` confirms that UTF-8 validation is effectively free.

5.4 Mispredictions

We evaluate the speculation overhead across the 1,000 real-world CSV files from the Kaggle dataset. Of 885K chunks, 74.6% succeed on the first speculative pass (assuming out-of-quotes with strict validation), and an additional 21.0% succeed on the second pass (in-quotes), for a combined 95.5% success rate on strict passes. For

725 files, all chunks could be parsed in the first pass. No chunks required re-parsing at validation time due to false positive parsing. Speculation in all our parsing experiments is structural only (number of fields, UTF-8 validation); this matches the weakest oracle the system supports, equivalent to a schema with all-string fields.

Mispredictions abort typically within a small number of bytes: even on a Project Gutenberg file (700 KiB avg. record size, two columns) where only 0.1% of chunks succeed on the first pass, throughput reaches 4.9 GB/s — 1.4× our corpus median. A configurable per-record size limit safeguards against pathological non-RFC-compliant inputs.

5.5 Parsing vs. Processing

Comparing parse performance of different systems including type parsing (or other data processing operations) is an apples-to-oranges comparison, since measurements then conflate throughput of the parser and throughput of the particular type conversion routine, whose performance can vary by orders of magnitude across systems and languages. Nevertheless, we want to highlight the impact of type parsing on throughput. We measure three scenarios on the lineitem dataset. The *baseline* counts records, representing a data-independent, near-zero-cost accumulation. *Integer parsing* extracts the quantity field into an unsigned integer, adding a single type conversion per record. *Complex parsing* converts all 16 columns into appropriate types (integers, floats, dates, and characters; three string columns need no conversion).

Baseline	188 GB/s	
Integer parsing	186 GB/s	(1.0×)
Complex parsing	40 GB/s	(4.7× slower)

A single lightweight conversion per record has negligible impact, whereas complex parsing is 4.7× slower — depending on the workload, type conversion can dominate parsing. Performing it inline avoids a redundant materialization pass, underscoring our fused parse-and-process model.

5.6 Integration

We integrate our CSV parser into Umbra, a JIT-compiling state-of-the-art database system [11, 12]. On the side of csveee, we expose a C FFI interface for both the parser builder and the parser itself. On the side of Umbra, we integrate the parser into the CSV scan operator. The operator configures the parser, JIT-compiles an accumulation function tailored to the columns that the operator needs to produce, and passes this function to the parser. Columns are collected into string arenas if needed. Once the parser has finished processing the CSV file, the operator produces its output.

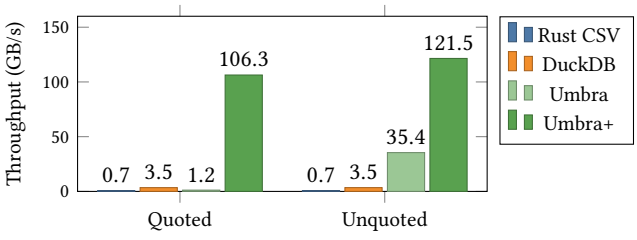


Figure 12: Counting line items again with various parsers in quoted and unquoted mode. Umbra+ uses csveee.

We verify the throughput of the integrated version. As in Figure 1, we count the number of records in the lineitem table at scale factor 100. Figure 12 shows the throughput of Rust CSV, DuckDB, and Umbra before and after integration of csveee. Compared to Umbra's built-in CSV parser, the integrated version achieves a speedup of approximately 86× in *Quoted* mode and 3.5× in *Unquoted*. The relatively modest speedup in the latter case is explained by the fact that Umbra is already able to process quote-free files in parallel.

6 OUTLOOK

Native Integration. Umbra and csveee maintain separate schedulers and worker threads, forcing data across thread boundaries and adding considerable overhead over the standalone results. A native re-implementation would remove this duplication and could drive the entire pipeline from within the accumulation function.

Translation to GPUs. Our vectorized, largely branch-free design is a natural fit for the SIMT execution model of GPUs, where branch divergence is a primary performance concern. A detailed GPU port is beyond the scope of this paper, and we leave it to future work.

Beyond CSV. Our programming model is not tied to CSV: the fused single-pass interface fits any line-oriented format such as JSON Lines, while speculation with user-defined oracles extends to formats with byte-ambiguous boundaries, like multi-line logs.

Richer Error Recovery. csveee rejects inputs that fail under every assumption whereas DuckDB, Spark, and pandas can be configured to skip malformed records and continue instead. Skip-and-continue could be added either at merge time (reparsing failed chunks with skipping enabled) or in the parallel phase via ranked assumptions and a shadow per-chunk state committed only on oracle acceptance. We view loud failure as the more robust default for aggregates.

Beyond Fixed Field Counts. Variable field counts per record would require a record interface exposing a variable rather than fixed number of fields, and a more sophisticated vectorization scheme or a DFA-based implementation; both trade throughput for flexibility. We leave this extension to future work.

7 CONCLUSION

In this paper, we addressed the CSV parsing scalability problem. Unlike parsers proposed in the literature, our parser, csveee, handles real-world CSV files, runs on general-purpose CPUs, and combines high single-thread performance with near-linear scaling. Starting from a simple synchronization-free strategy for speculative parallel parsing, we presented a new programming model for single-pass parsing, an end-to-end approach to vectorized CSV parsing, and an I/O buffer design for zero-copy record construction. Our evaluation shows that these contributions, combined with an efficient scheduling strategy, yield speedups of more than 256× over existing parsers. With our approach, modern database systems can achieve CSV parse throughputs exceeding 100 GB/s, finally making CSV ingestion match the bandwidth of modern many-core systems.

ACKNOWLEDGMENTS

We thank Alexander Kumaigorodski for providing the implementation of CUDAFastCSV.

REFERENCES

[1] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are you sure you want to use mmap in your database management system?. In *CIDR*.

[2] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[3] Till Döhmen, Hannes Mühleisen, and Peter Boncz. 2017. Multi-hypothesis CSV parsing. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*. 1–12.

[4] Chang Ge, Yinan Li, Eric Eilebrecht, Badrish Chandramouli, and Donald Kossmann. 2019. Speculative distributed CSV data parsing for big data analytics. In *Proceedings of the 2019 International Conference on Management of Data*. 883–899.

[5] Alexander Kumaigorodski, Clemens Lutz, and Volker Markl. 2021. Fast CSV loading using GPUs and RDMA for in-memory data processing. In *BTW 2021*. Gesellschaft für Informatik, Bonn, 19–38.

[6] Geoff Langdale and Daniel Lemire. 2019. Parsing gigabytes of JSON per second. *The VLDB Journal* 28, 6 (2019), 941–960.

[7] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. 1977. Abstraction mechanisms in CLU. *Commun. ACM* 20, 8 (1977), 564–576.

[8] Duane Merrill and Michael Garland. 2016. Single-pass parallel prefix scan with decoupled look-back. *NVIDIA, Tech. Rep. NVR-2016-002* (2016).

[9] Johann Mitlöhner, Sebastian Neumaier, Jürgen Umbrich, and Axel Polleres. 2016. Characteristics of open data CSV files. In *2016 2nd International Conference on Open and Big Data (OBD)*. IEEE, 72–79.

[10] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. Instant loading for main memory databases. *Proceedings of the VLDB Endowment* 6, 14 (2013), 1702–1713.

[11] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment* 4, 9 (2011), 539–550.

[12] Thomas Neumann and Michael J Freitag. 2020. Umbra: A disk-based system with in-memory performance.. In *CIDR*, Vol. 20. 29.

[13] Jon Postel. 1981. *Transmission control protocol*. RFC 793. Internet Engineering Task Force.

[14] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*. 1981–1984.

[15] Yakov Shafranovich. 2005. *Common format and MIME type for comma-separated values (CSV) files*. RFC 4180. Internet Engineering Task Force.

[16] Elias Stehle and Hans-Arno Jacobsen. 2020. ParPaRaw: Massively parallel parsing of delimiter-separated raw data. *Proceedings of the VLDB Endowment* 13, 5 (2020), 616–628.

[17] Transaction Processing Performance Council. 2022. TPC-H Benchmark Specification, Revision 3.0.1. https://www.tpc.org/tpch/. Accessed: 2026-07-22.

[18] Gerrit JJ van den Burg, Alfredo Nazábal, and Charles Sutton. 2019. Wrangling messy CSV files by detecting row and type patterns. *Data Mining and Knowledge Discovery* 33, 6 (2019), 1799–1820.

[19] Alexander Van Renen, Dominik Horn, Pascal Pfeil, Kapil Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3694–3706.

[20] Haixun Wang and Carlo Zaniolo. 1999. User-defined aggregates in database languages. In *International Symposium on Database Programming Languages*. Springer, 43–60.