NanoCSV: Enabling Efficient Parallel CSV Extraction with Hierarchical Finite-State Transducer

Peiyuan Dai[1,2,3,4], Rui Liu[5], and Heng Zhang[1,3(✉)]

[1] Institute of Software Chinese Academy of Sciences, Beijing, China
[2] University of Chinese Academy of Sciences, Beijing, China
[3] University of Chinese Academy of Sciences, Nanjing 211135, China
[4] Nanjing Institute of Software Technology, Nanjing, China
[5] The University of Chicago, Chicago, United States
daipeiyuan22@mails.ucas.ac.cn, ruiliu@cs.uchicago.edu,
zhangheng17@iscas.ac.cn

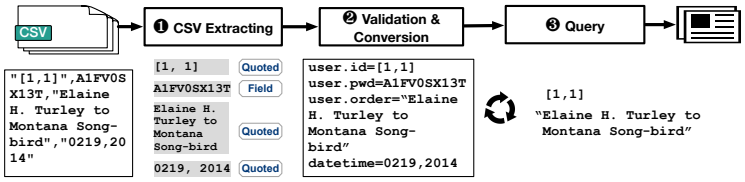
Abstract. The growing use of comma-separated values (CSV) formatted files for various applications has highlighted the need for efficient parsing and analysis. However, traditional methods struggle with parallel transformation due to complex parsing rules, resulting in poor scalability and high overhead. Recent CSV processing techniques face mainly three key challenges: (1) inevitable performance degradation resulting from ineffective parallelization caused by character-by-character scanning, (2) limited parallelism on modern processors, and (3) redundant indexing costs for specific queries. This paper proposes a novel high-performance CSV processing framework, `NanoCSV`[6]. First, `NanoCSV` designs a new **hierarchy finite-state transducer strategy** to facilitate accurate parallel parsing with minimal states and transitions, increasing scalability. Second, `NanoCSV` accelerates parallel CSV data extraction and conversion through a **SIMD string vectorization and extraction pipeline** that incorporates fast SIMD-aware CSV string parsing. Furthermore, to handle queries with varying workloads, `NanoCSV` uses a **dynamic decision-making indexing mechanism** that combines hybrid bitmap and comprehensive tabular indexing. Extensive evaluations demonstrate the substantial performance improvement achieved by `NanoCSV` compared to state-of-the-art frameworks such as SIMDCSV, Spark, MongoDB, PostgresRaw, and GIO.

Keywords: Comma-Separated Values (CSV) · High Performance Parser · Hierarchical Finite-State Transducer.

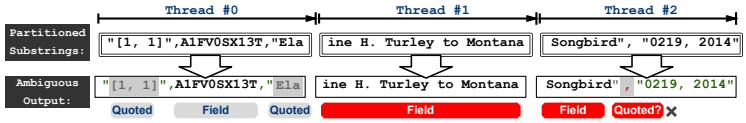
1 Introduction

The Comma-Separated Value Format (CSV), known for its simplicity, flexibility, and ubiquity, has become increasingly prevalent in modern data mining and

[6] The source code can be found at: https://github.com/nanocsv/nanocsv



(a) Workflow of CSV-formatted file analytic.



(b) An example of misinterpretation in parallel CSV parsing.

Fig. 1: Workflow of CSV analytic and a misinterpretation example.

machine learning applications for data exchange and conversion [24]. With the expansion of application scales, the volume of CSV data in analytics pipelines has surged. For example, industrial cloud log management platforms now daily process over 100 petabytes of data and 500 trillion records in CSV format from approximately a million different sources. This increasing dependency on the CSV format has led to its native integration within modern data processing and analytical systems, such as Apache Spark [2], TensorFlow [27], MongoDB [20], and DuckDB [23]. In Figure 1(a), CSV parsing and extraction [29], the conversion of raw CSV data into tabular or key-value forms, is a major efficiency bottleneck due to its computational intensity [10]. Recent emerging research efforts are targeting high-performance methodologies for analyzing CSV-formatted data sets [4, 10]. However, the effectiveness of parallel parsing of CSV-formatted data is hindered by a multitude of unpredictable symbol transformations resulting from stringent parsing rules. This results in a considerable impact on parsing accuracy and suboptimal parallelism. Specific examples of parser misinterpretation issues are as follows.

Example I (Parsing Misinterpretation): Taking Fig. 1 as an example, parallel CSV parsing methods produce partitioned substrings of equal length from a raw CSV Yelp file. Given the standard CSV formatted semantics (see Section 3), there exist two possible interpretations for this substring. The primary field causing ambiguity is "Elaine H. Turley to Montana Songbird". In the ambiguous parsed situation, the thread 2 *is considered part of an ongoing context, ending with "Ela in the same field quoted as the thread* 0*. Situation* 2 *views thread* 2 *as unrelated to the previous thread's context, isolating ", "as a separate field. Without determining the correct interpretation of* `Elaine H. Turley to Montana SongbirdFig`*, these two parsed interpretations would lead to drastically different results and impact further data analysis.*

Challenge I: Ensuring accurate CSV parsing while optimizing on multicore processors is still challenging due to the dependency of nested delimiters. Symbols in CSV files require context-dependent interpretation, necessitating character-by-character sequential scanning in recent frame-

works [18, 2, 20] to accurately identify rows and columns, particularly when fields are enclosed in double quotes. Parallel parsers have to precisely divide CSV records using row delimiters to prevent misinterpretation of contextual semantics caused by arbitrary splits.

Challenge II: High-performance CSV parsing still lies in *the limitation of instruction-level parallel (ILP) implementations in modern architecture*. Achieving parallel parsing of variable-sized CSV partitions [17] from arbitrary split points is essential for scalability. Despite years of optimization, mature methods [7, 3, 11, 10, 5, 6] still depend on sequential scanning and predefined indexing strategies, posing challenges to optimal performance in modern multi-core architectures.

Challenge III: *Achieving a trade-off between minimizing index build time and query time is difficult due to various types of workloads.* To balance the overhead of building indexing and query execution, it is necessary to establish adaptive indexing over various types of CSV tabular structure to accommodate varied workloads.

This paper introduces a novel CSV processing framework called `NanoCSV`, exploring architecture-aware parallel design to improve efficiency and support conditional query applications. First, we propose a novel hierarchical finite-state parallel transducer (FST) with minimizing states to guide the parallelism of scanning and merging process. The FST model establishes a rule for complete character states and the transition relationship among symbols, delimiters, and strings according to predefined CSV format standards. To simplify the dependency, FST builds a two-level transition rule to guide the parallel record and field extraction. Second, leveraging multi-core parallelism and SIMD instructions, `NanoCSV` utilizes an instruction-level parallel design that significantly increases the speed of parallel CSV file extraction. `NanoCSV` utilizes AVX (Advanced Vector Extensions) instruction sets to achieve instruction-level parallelism and binds blocks to idle CPU cores to achieve thread-level parallelism. From the experimental evaluation, `NanoCSV` achieves an average 4× improvement in CSV extraction and up to 152× throughput compared to MongoDB.

2 Related Work

Recent years have seen incredible advances in CSV data processing, such as parsers [7, 28, 13, 25, 24, 3], distributed systems [6, 10], and approximate detections [15, 5, 4]. Most databases and distributed data processing frameworks [2, 22, 20] integrate abundant CSV APIs and standard ingestion procedures to support CSV-formatted file analysis and treat them as tabular structures.

CSV Processing Frameworks. PostgresRAW [1] introduces an adaptive indexing mechanism that preserves the positional information of CSV files, facilitating efficient access to raw data in conjunction with a versatile caching framework. This approach successfully bypasses the entire loading overhead while delivering query performance with standard PostgreSQL and continuously surpassing it. Pandas [22] initially ingests large CSV files as a single block and

subsequently processes the input sequentially, foregoing the potential for parallelization. Apache Spark [2] exploits multicore parallelism for CSV data loading and retrieval. However, when exploring distributed CSV data processing, Spark needs to ensure that each resilient data partition includes a complete set of new lines or enclosed delimiters inside double quotation marks; otherwise, it is prone to errors during the splitting process, resulting in incorrect block reduction and misjudgment in the reduction phase.

Parallel Solutions for CSV Processing. Several works introduce the transformation for CSV records and the table detection method to facilitate the CSV query procedure [15, 28, 3, 5]. Pytheas [5] as a representative utilizes approximate pattern mining methodology to discover table structures and potential errors within tolerable limits. Prior projects, regardless of record-centric queries [25, 19, 13, 1], table-centric operations [5], and on-demand fetch [8], were centered on simplifying computation-related programming efforts. GIO [8] initially defines mapping rules for converting raw data to a structured format and automates the generation of an efficient multithreaded reader. This reader processes all raw datasets in the specified format, facilitating parallel processing of CSV files. Moreover, FishStore [30] relies on a multichain hash indexing approach for dynamically registered predicated subsets of data. However, its utilization requires the combination of different parsers, which poses a significant challenge to relevant personnel.

FSM-Based CSV Parsing. The state-of-the-art CSV parallel parsing work [21, 10, 4] introduces speculative methods over FSMs to enhance parallelism, in which each CSV file is partitioned into equal-sized CSV partitions (chunks) for parallel scanning. However, following a prebuilt static FSM model and full-state conversion, parsing from original states must cache all numerous intermediate strings, which introduces a significant, redundant workload and negatively impacts the extraction performance of these work. In this paper, we seek a new hierarchical FST solution to address these limitations by eliminating the overhead of symbol identification. Unlike recent methods, we maintain a two-level transducer to determine the CSV delimiter first and then perform parallel extraction and indexing over substrings.

The primary drawbacks of these state-of-the-art approaches are (a) the neglect of parallel parsing [14] and (b) the challenge of accurately parsing CSV files with diverse symbols. Thus, there exists a complicated trade-off between scalability and transition efficiency.

3 Our Observations

The formation of CSVs is traditionally defined following the standard RFC-4180 [24]. The Web Working Group delineates methodologies for interpreting W3C compliant CSV formatted data sets as relational data. Each CSV `File` contains [`header`] records that describe the structure of the table columns, followed by a sequence of table records. New line characters ("`\n`") commonly separate the record, which consists of a sequence of `field`. Further, commas characters

(“`,`”) separate the `field`. Each `field` in a record is *quoted* or *unquoted*, in which the *quoted* field is surrounded by two quotes (“`"`”), while *unquoted* is purely a sequence of characters. Note that the quotes (“`"`”) embedded in a quoted field need to be escaped by preceding it with another quote.

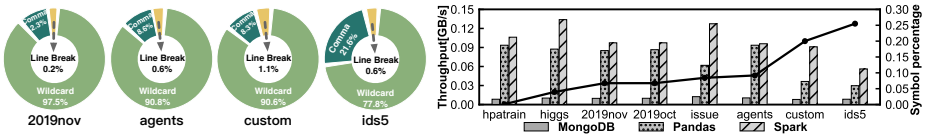
Current hardware architectures have highlighted the bottlenecks in CSV extraction and conversion in the overall analysis pipeline, particularly in parallel processing [9, 16, 12, 17]. As following, three critical observations are identified to guide our system design.

Observation I: Data conflicts and dependencies limit the parallelism of CSV extraction. Fig. 2(b) presents an analysis of runtime across recent frameworks [7, 28, 25], including MongoDB, Pandas, and Spark. It is evident that performance fluctuates depending on the size of the file and the mix of different symbols contained within the CSV files. The root of this complication lies in the dependency of each parser on the output generated by its sequentially preceding parser. Before starting its execution, a parser requires the outcome of the previous parser to ensure a conflict-free parsing of shared characters. Hence, the dependency on sequentially parsed results before subsequent execution underscores a critical bottleneck, necessitating innovative approaches to mitigate conflicts and dependencies, thereby fostering improved parallelism within processing pipelines.

Observation II: Parsing operations that involve multiple strings can be substantially expedited by employing SIMD vectorization instructions. Recent manufacturers have progressively scaled the number of cores, expanding SIMD capabilities. However, most existing frameworks still process single-order characters and delimiters individually [20, 22]. They inherently restrict the potential for parallel processing during the data extraction and conversion phases, as noted in seminal works within the field. An exploration of enhanced parallelism for CSV data handling reveals that the extraction of CSV messages could greatly benefit from increased parallelism across a wider range of characters. Hence, CSV analysis frameworks could be constructed with scalability, leveraging the current level of hardware parallelism, and taking advantage of the improvement of parallelism.

Example II (SIMD Vectorization Opportunity): For example, standard characters such as commas and newlines, represented in the ASCII *code, occupy* 8 *bits each. The capabilities of modern multicore processors, equipped with advanced Single Instruction, Multiple Data (SIMD) vectorization instructions at* 256*-bit or* 512*-bit widths, suggest significant untapped potential. These instructions facilitate the simultaneous scanning of* 32 *or* 64 *characters, thus aligning these characters in the processor's cache and enhancing processing efficiency.*

Observation III: The varied workloads of queries need to tailor different indexing strategies. Previous CSV query techniques have mainly focused on ad-hoc optimizations for specific stages of the CSV pipeline. Recent work [10, 30] focuses on full file parsing and indexing, but requires that CSV data sets be fully loaded to execute queries. Query engines, on the other hand, process CSV-formatted inputs with only a tiny percentage of the messages typ-



(a) Ratio of Different Symbols. (b) Throughput over Different Symbol Ratios.

Fig. 2: Throughput with the different capacity ratios of symbols.

ically analyzed. Therefore, a lightweight tabular structural or comprehensive indexing strategy must be deliberated over various query workloads to facilitate application-specific content filtering and on-demand data analytics.

Inspired by these observations, we seek to mitigate the performance challenges associated with scaling comprehensive CSV processing capabilities that are centered on a novel finite-state transducer model, while effectively harnessing parallel processing on contemporary hardware and SIMD intrinsics.

4 `NanoCSV`: Parallel CSV Analytics with Hierarchical Finite State Transducer

4.1 System Overview

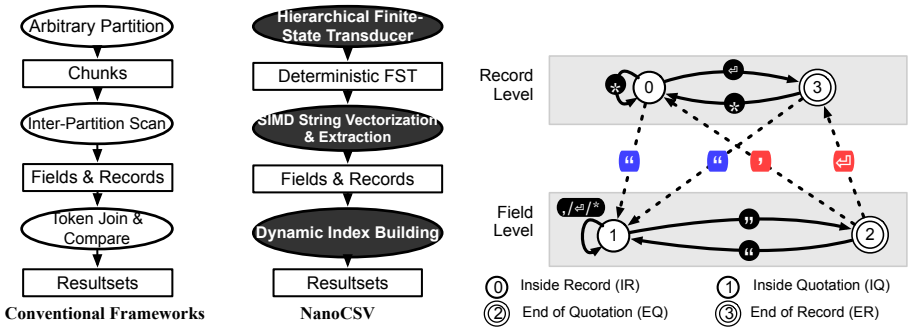
Based on our novel observations, we first propose a new hierarchical finite-state transducer (FST) with record and field-level layers to drive the parallel extraction automaton. The linkage between the two hierarchical layers is dependent on the occurrence of nested quotations. In addition, `NanoCSV` newly advocate the following three principles of system design to maximize the advantages of hierarchical FST.

Delimiter Identification via Hierarchical FST: In contrast to the workflow in the existing work (Fig. 3(a)), `NanoCSV` *design a hierarchical finite state transducer (FST) model with state minimization.* Our hierarchical FST faciliates the parallel character transitions at the record and field level to separate the processing of records and fields. The FST model simplifies the transition rules as the syntax stack becomes a call stack, automatically managed by the `NanoCSV` runtime system.

Rapid Extraction through SIMD String Reorganization: To achieve high parallelism, we further leverage a new SIMD string vectorization and extraction strategy to determine the symbol states and logical positions of delimiters in each record. Each thread performs extraction operations via vector-formatted instructions for parallel scanning and delimiter identification.

Dynamic Indexing based Workload: In order to balance the overhead associated with indexing and querying, `NanoCSV` implements a dynamic index construction methodology by thoroughly evaluating querying mechanisms, computational resources, and workloads. Ultimately, the selection of the optimal indexing approach is guided by a decision based on a cost model.

These three novel designs of `NanoCSV` are described as a hierarchical FST model for parallel CSV analysis, determining unpredictable delimiters, architecture-



(a) CSV extraction workflow of existing frameworks and `NanoCSV`.

(b) Hierarchical FST for record- and field-level automaton.

Fig. 3: Workflow Comparison and Hierarchical FST Model Design in `NanoCSV`.

aware CSV data parsing pipeline strategy, and dynamic indexing. Together, they provide an efficient parallel implementation of the CSV analytics framework.

4.2 Hierarchical Finite-State Transducer Model

The critical step to enhance the scalable performance of CSV parsing is to allow a thread to automatically determine the context of the assigned chunk and interpret its symbols. The newly designed hierarchical finite-state transducer in `NanoCSV`, as shown in Fig. 3(b), consists of character transitions at the record and field level, to separate the processing of records and fields. FSTs derived from delimiter-separated files depend on the character state from different predefined spreadsheet standards. First, we define the transitions of the confined set of FSTs utilized for parallel CSV extraction.

Definition 3.1 (Finite State Transducer). We define $FST_{CSV} = (S, T)$ as a complete state set of characters for file formation, where S is the set of symbols and delimiters to separate strings, T is the transition relationship based on symbols, delimiters, and characters.

The complete model of a finite-state transducer consists of all deterministic states for the transitions of a CSV formatting file. The following property of CSV supports the strategy:

Lemma 3.2. The number of character states S in CSV-formatted strings is finite. The whole symbol set from all delimiter-separated formatted files consists of $\{\hookleftarrow\}$ for record delimiter, $\boxed{,}$ for field delimiter, $\boxed{"}$ for quotations, and $\boxed{\backslash}$ and $\boxed{/}$ for escape sequences. The combination of these symbols determines the state transitions T in FST.

According to the state transition of the characters, we conclude the following four states and their corresponding transitions to conclude all cases while analyzing the CSV files.

⓪ **Inside of Record (`IR`)**: where the record is enclosed by a comma on both sides or a comma on one side and a line break on the other.
① **Inside of Quotation (`IQ`)**: where the field is enclosed by double quotes.
② **End of Quotation (`EQ`)**: where a quotation is encountered after another, indicating the end of a set of quotations. The following character is a comma or a line break; otherwise, an error exists in this invalid quotation.
③ **End of Record (`ER`)**: corresponds to the end of a record.

Example III (Hierarchical FST Sample): Fig. 3 shows an example of hierarchical FST with the four states and their corresponding transitions. After instant loading of the input, the symbols and characters in each partitioned fragment will be inspected along with the transitions in FST to ensure the extraction accuracy without ambiguous results. Each thread assumes the first character's state and then conducts the inspection to determine the final state of the chunk. We note that when the procedure for the EQ state (②) encounters other characters except{`,` `"` ↩}, it will lead to an invalid state (marked as `IV`).

Furthermore, initiating from each state in the whole set, the interpretation procedure needs to identify the end state of the chunks and interpret each character inside the chunks until the last one. Thus, it is necessary to construct an FST with minimized states to reduce redundant interpretation workloads. Interestingly, we observe that the field-level transducers could be exactly eliminated when handling CSVs without double quotations. Based on Lemma 3.2, we can infer the following state-minimizing property of transducers for a CSV with multiple symbols.

Theorem 3.3. Only the pairing of existing double quotations `"` for tagged strings triggers the transitions of the field-level transducer. The symbol of `"` yields the trigger point for state minimization.

4.3 Efficient CSV Extraction through SIMD String Reorganization

Using the FST model, we establish the core design of parallel CSV extraction with two essential techniques in their stages to tailor a high-performance CSV framework (addressing Challenge II). These include: (1) (preprocessing) Parallel transducer schema for chunk scanning; (2) (extraction) SIMD-accelerated bitwise extraction; These stages are specifically designed to handle different states of the symbol set according to the predefined hierarchical FST and architectural features. We introduce the concrete design for these two stages in the following.

Parallel Transducer Schema for Chunk Scanning. We start with the pre-processing stage to partition the file evenly and launch the scanning over the chunks. Following the paradigm of FST parallelism, `NanoCSV` allows threads to operate rapidly pointer chasing to determine the state of each character in chunks, as depicted in Fig. 4. Initially, the whole file stream from a CSV-formatted file is partitioned into n chunks, with each thread processing one. Using the previously determined FST as a starting point, the `NanoCSV` program initiates a fixed round scanning procedure. As each character is processed,

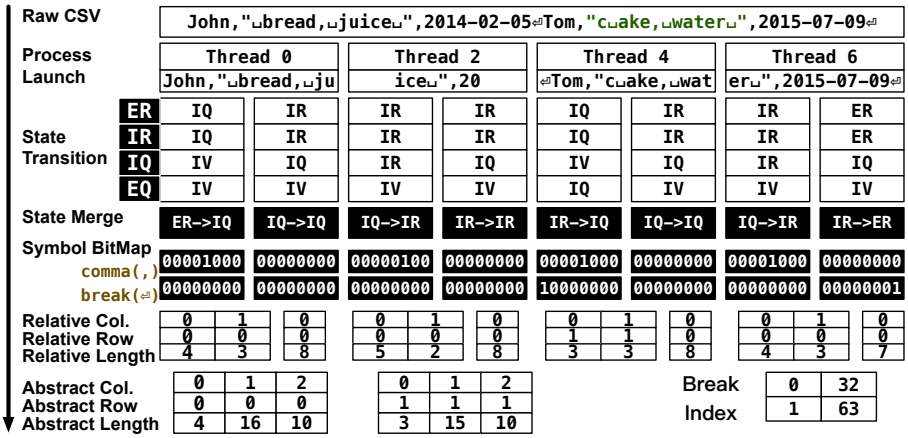


Fig. 4: The process flow of NanoCSV operates pipelined.

the corresponding state transition is determined on the basis of its tag and the state of the preceding character. For example, when scanning the chunk of "John,"□b", Thread-1 starts from the four states {ER, IR, IQ, EQ}, and interprets to final states {IQ, IQ, IV, and IV} along transitions.

Next, the resulting state is forwarded to the tag of the subsequent character, continuing the state transition process until the chunk's end is reached. This process results in the final-state array for each chunk. During the state array generation process, two corresponding bitmap structures are initialized, representing the *symbol state* of a comma and line break (as shown in Fig. 4), are initialized during the state array generation process. Furthermore, the transitions between IR and ER states are generally identical. Thus, any invalid state renders subsequent state transitions and bitmap generation superfluous. Afterward, each chunk's initial and final state arrays are inscribed into the state matrix based on the chunk number, sequentially arranging all chunks' initial and final states.

Lastly, after chunk scanning is finished, a single procedure is invoked to check each chunk's initial- and final-state arrays. It then determines a correct mapping for interchunk state transitions by verifying whether the state of the chunk's last character matches the next one's initial state. Moreover, we select the bitmap corresponding to the joint state from the limited number of bitmaps within the chunk to form a final large-scale bitmap.

SIMD-Accelerated Bitwise Delimiter Identification. The dependencies between symbol characters are mostly limited to instances where commas and newline characters serve as row and column delimiters. NanoCSV uses SIMD's vector-formatted instructions to develop a parallel instruction solution for delimiter identification. We capitalize AVX2's single-line SIMD instructions to work on 256-bit data, i.e., 32 characters per instruction. Accordingly, we establish the pre-defined chunk size to align with AVX2's processing capabilities, typically setting to a multiple of 32 characters.

The list 1.1 illustrates a pseudocode example of the identification procedure *comma* (same as *newline*). Considering the CSV-formatted delimiters for records

and fields, NanoCSV focuses on detecting *comma* and *newline*, each represented as an 8-bit *ASCII* code. The string of each line in a chunk is first divided into 32-bit groups using _mm256_loadu_si256 intrinsic, and the filters are applied sequentially in *ASCII* code format (Line 3). When a comma or newline character is encountered, the filter sets the corresponding position to 1, producing a binary line via _mm256_cmpeq_epi8 operation where 1 indicates the presence of the delimiters at the corresponding position, and 0 represents an irrelevant wildcard (Line 4). We introduce a mask to convert the binary string obtained from the filter into an integer type via _mm256_movemask_epi8 (Line 5). The mask values represented as binary are stored in binary format. Using a bit scan, the corresponding positions in the bitmap are set to 1 (Lines 7-9). NanoCSV significantly reduces the character size from 256 to 32 bits, enabling more space-efficient subsequent processing. We illustrate an example as follows.

```
1  void identifyDelimiter(char*& line, bool*&columnBegin, bool*&
       recordBegin){
2      const __m256i column = _mm256_set1_epi8(',');
3      __m256i op =_mm256_loadu_si256((__m256i*)line);
4      __m256i eqC = _mm256_cmpeq_epi8(op, column);
5      int maskC = _mm256_movemask_epi8(eqC);
6      while(maskC) {
7          int CP = bitScan(maskC) - 1;
8          columnBegin[CP] = true;
9          maskC &= ~(0x1<<CP);
10     }}
```

Listing 1.1: Pseudo code of SIMD intrinsics to determine commas.

Fig. 4 illustrates the leveled symbol state bitmaps constructed for the record, assuming a 32-bit processor word for ease of representation. Each symbol state bitmap organizes 32 bits in little-endian order. The delimiters for records and fields, such as the comma symbols (,) and break (↩) symbols, are extracted. In the context of the nested object "↩Tom,}"c ", the comma bitmap sets bits corresponding to 00001000, while the break bitmap sets bits to 10000000.

Considering distributing balanced workloads among multicore, NanoCSV maintains a work queue to organize the available cores and instantly assigns the chunks to be processed to the work unit. Due to the impact of the configuration of the chunk size, NanoCSV automatically selects an optimal chunk size based on the input features and system resources to avoid excessive irregular workloads. This optimal strategy for allocating adaptive resources benefits from maximizing core utilization and achieving effective parallelism.

In general, NanoCSV employs SIMD's vector instructions to facilitate efficient parallel extraction of CSV data. We develop an efficient delimiter identification approach to improve the performance of structural division and multi-core scheduling, thereby achieving a high level of parallelism. Using AVX2's instructions, NanoCSV efficiently constructs data conversion via the elaborate optimization of filter, mask, and bitwise operations.

4.4 Dynamic CSV Tabular Indexing and Query Programming

The dynamic decision-making indexing mechanism employs dynamic programming. It achieves the goal of minimizing query time through a combination of querying mechanisms, computational resources, and workloads.

Computational Resources. By optimizing the configuration and use of these computational resources, the efficiency and performance of data processing systems can be significantly improved. Factors that influence computational resources include the number of CPU cores and clock frequency, read and write speeds of solid-state drives (SSD), data scale, and more.

Workloads. Workload is a major factor that affects query time. The construction time for bitmaps and indexes is proportional to the data volume, and large datasets require more time to create indexes.

We provide the state transition equations for a bitmap and an index. Given $i \in (0, N)$ as the number of file blocks and $j \in (0, T)$ as the query constraint time, we conceptualize our solution as a backpack optimization problem.

$$dp[i][j] = \begin{cases} \min(dp[i][j], dp[i-1][j - T_i] + nT_{bitmap}) \\ \min(dp[i][j], dp[i-1][j - (T_i + \log(T_i))] + nT_{index}) \end{cases}$$

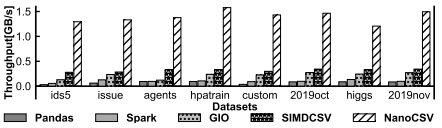
In this context, dp represents the determination of which configuration can handle n queries within a given time j. where T_i represents the time to build the bitmap, and $\log(T_i)$ indicates the time to build an index based on the bitmap. For the current block i, after n queries, T_{bitmap} represents the time required to perform a single query on block i using the bitmap, while T_{index} denotes the time required to perform a single query on block i using the index. `NanoCSV` achieves an optimal solution under different CSV workloads using this formula.

5 Evaluation

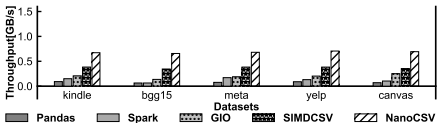
We used a server for experimental evaluation, equipped with a 16-core Intel(R) Xeon(R) Platinum 8269CY CPU @ 2.50GHz with hyper-threading, two 512GB SSDs (actual sequential read rate of 1475MB/s), and 128GB DDR4 memory. Server runs Ubuntu 22.04 with Kernel version 5.15.0. All C++ programs were compiled with the -O3 flag, and timing results are the average of three runs.

Evaluation Datasets. All datasets are from open-source repositories, including Kaggle, Yelp Open Dataset, and UC Irvine Machine Learning Repository. In Fig. 5, we use the datasets ids5(328MB), kindle(701MB), issue(1.39GB), bgg15(1.39GB), agents(1.64GB), meta(1.65GB), hpatrain(2.98GB), custom(4.54GB), yelp(4.72), 2019oct(5.67GB), higgs(8.04GB), 2019nov(9.01GB) and canvas(21.7GB).

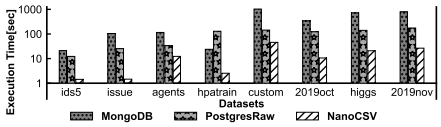
Baselines. We compare `NanoCSV` with six CSV processing frameworks, including Apache Spark (v3.2.0), Pandas (v1.4.2), MongoDB (v6.0.8), PostgresRAW [1], GIO [8] and SIMDCSV [26], regarding parsing throughput, data extraction performance, and end-to-end query time.



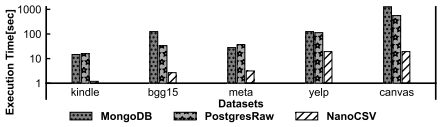
(a) Throughput over CSV File with Simple Delimiters.



(b) Throughput over CSV File with Quotation Marks.



(c) Query Performance over CSV Files with Simple Delimiters.



(d) Query Performance over CSV Files with Quotation Marks.

Fig. 5: Parsing throughput comparison among Pandas, Spark, GIO, SIMDCSV, and our `NanoCSV`. And end-to-end query performance among MongoDB, PostgresRAW, and `NanoCSV`. We report the throughput comparison over the CSV files (a) with simple delimiters and (b) quotation marks and query performance over the CSV files (c) with simple delimiters and (d) quotation marks.

Evaluate Metrics. To showcase the effectiveness of our optimizations, we compare the speedup of our implementation with six state-of-the-art CSV processing frameworks. `NanoCSV`'s performance is further evaluated using numerous micro-benchmarks and diverse datasets under various settings, including the sensitivity of partition chunk size and varying input symbol ratio. Furthermore, we illustrate the runtime breakdown to spotlight the effectiveness of optimizations. Finally, we showcase the scalability of `NanoCSV` under different multi-core configurations.

5.1 Overall Performance

This section evaluates `NanoCSV`'s overall parsing throughput and end-to-end query performance against other state-of-the-art frameworks. We measured the wall-clock time for the overall parsing throughput evaluation, including reading the input from the SSD and writing the parsed data back to system memory. For end-to-end query experiments, we measured the query execution in an end-to-end mode, including parsing and query execution.

Fig. 5 presents an overall comparison among frameworks. Our proposed `NanoCSV` demonstrates impressive performance, with parsing speeds ranging from over 600MB/s to around 1.5GB/s. Specifically, Fig. 5(a) and (b) illustrate the parsing throughput. For the performance of recent frameworks, Pandas and Spark in single-core exhibit similar speeds, both around 100MB/s. GIO first creates a mapping table from the given sample attribute and parses the entire CSV file. Despite using multi-threaded parallel technology, GIO's throughput for CSV files is around 200MB. SIMDCSV utilizes a prediction method, achieving a throughput of approximately 300MB/s. Moreover, as illustrated in Fig. 5(b), it is evident that `NanoCSV` exhibits a marked decrease in throughput when processing CSV files (kindle, bgg15, meta, yelp, and canvas) in comparison to the preceding

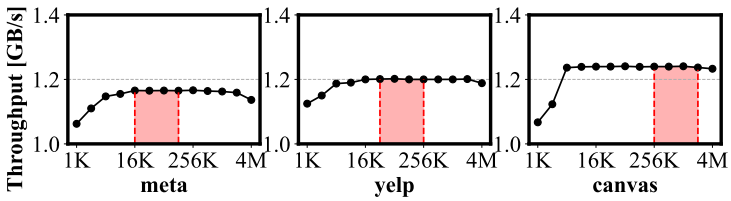


Fig. 6: Evaluation over different chunk size configurations.

dataset featuring straightforward delimiters. This decline is due to the inclusion of double quotations, requiring reduced field granularity for accurate analysis.

We further evaluate the end-to-end query performance by comparing `NanoCSV` with MongoDB and PostgresRaw. Fig. 5(c) and (d) illustrate the results of query execution time. The results show that `NanoCSV` outperforms MongoDB and PostgresRAW by one to two orders of magnitude in the ad-hoc query scenario. MongoDB's parsing throughput fluctuates between 30-50MB/s, primarily due to the reconstruction of its data parsing library, which hinders overall performance. PostgresRAW, employing a similar approach, exhibits a marginal speed improvement over MongoDB.

Insight: Overall, we observe that `NanoCSV` can compete with all other CSV frameworks in terms of parsing and end-to-end query performance, thanks to the complete parallelism achieved during CSV extraction and parsing.

5.2 Chunk Sizes Sensitivity and Different Symbol Ratios

In `NanoCSV`'s design, we aim for complete parallelism by dividing the file into chunks and assigning them to idle processes. We evaluate the performance dynamics with various chunk sizes from 1KB to 4MB. Fig. 6 demonstrates that `NanoCSV`'s throughput decreases significantly for chunk sizes less than 4KB due to excessive creation and cancellation of process stacks. However, when the chunk size increases to 4KB, this issue is alleviated. We also identified an optimal chunk interval during experiments, with chunk sizes between 64KB and 512KB yielding better throughput for a 1GB file and between 256KB and 2MB better for a 6GB file. Then, we delve deeper to elucidate the effects of varying symbol ratios. The three datasets contain different symbol ratios (22%, 9%, and 1% symbols, respectively). The results clearly show noticeable differences in their throughputs (4.50, 5.99, and 7.37GB/s), with smaller symbol percentages correlating with higher throughputs. This behavior can be attributed to `NanoCSV`'s need to identify delimiters within symbols before determining the Finite State Transducer (FST), which becomes more efficient as the symbol percentage decreases. By understanding this relationship, we can better optimize the performance of `NanoCSV` under varying symbol ratios.

Insight: Overall, an interesting result is observed, revealing that `NanoCSV`'s performance varies over a range of chunk size configurations. Notably, to ensure a steady throughput improvement, `NanoCSV` incorporates the ability to configure an optimal chunk size dynamically. During runtime, the chunk size value is estimated based on the varying sizes of different CSV files and storage layers.

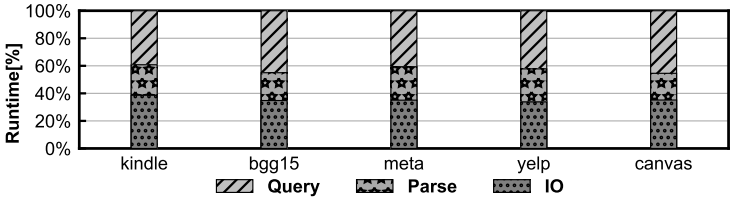


Fig. 7: Runtime breakdown over large CSVs

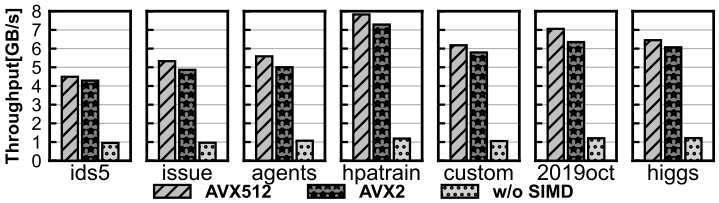


Fig. 8: Evaluation of SIMD-aware optimizations.

5.3 Breakdown and Effects of Optimization

Fig. 7 illustrates the runtime breakdown of `NanoCSV`, showing that data loading (IO), extraction (parse), and query stages of `NanoCSV` account for approximately 34%, 25%, and 41% of the time, respectively, across the five datasets. To regulate the duration of the query and more distinctly highlight the proportion of different phases, we intentionally queried for fields that are absent in the file. From the results, `NanoCSV` has sharply enhanced the performance of extraction and parsing (25%) and focused on querying optimization. Further, we evaluate the efficiency of exploring SIMD (Single-Instruction, Multiple-Data) instructions compared to sequential extraction, as shown in Fig. 8. The SIMD-based optimization yields an average throughput of $5-6\times$ higher than the state-of-the-art framework that adopted sequential extraction. To determine the further impact of AVX instructions, we implemented NanoCSV by selecting AVX512 and AVX2. Fig. 8 illustrates the performance comparison between AV512 and AVX2 implementation, presenting that `NanoCSV` with AVX512 speeds up $1.05-1.1\times$ than AVX2. This phenomenon is because `NanoCSV` mainly exploits SIMD instructions to vectorized read and value compare other than modifications.

Insight: The results of memory utilization and overall runtime show that constructing SIMD-aware bitmaps to retrieve row and column-aligned values enables an efficient delimiter identification strategy and efficient extraction performance.

5.4 Scalability

Finally, we evaluate the scalability of `NanoCSV` 's extraction. On 8-core, 16-core, 24-core, and 32-core systems, the speedup achieved was 7.2x, 14x, 19.4x, and 23x, respectively. The results show that, despite some performance degradation, we optimize most parallelism in the extraction phase, with a single thread managing the block state sequence. This approach maximizes efficiency and boosts overall extraction performance.

Insight: Our FST and SIMD optimizations significantly boost `NanoCSV` 's scalability, achieving near-linear speedup on 32-core machines.

6 Conclusion and Future Work

In this work, we propose `NanoCSV`, a novel CSV pipeline framework designed to address key challenges in state-of-the-art solutions: low sequential computation efficiency, low degree of parallelism with architectural awareness over modern commodity processors, and complexity in writing efficient CSV analytics. `NanoCSV` achieves a highly scalable runtime and a finite state transducer technique to identify the end-to-end status of CSV chunks. Results on real-world applications show `NanoCSV` achieving up to an order of magnitude speedup over existing methods, with strong scalability across different architectures.

Acknowledgments. This work was partially supported by National Natural Science Foundation of China (62002350) and Youth Innovation Promotion Association CAS (2023120).

References

1. Alagiannis, I., Borovica, R., Branco, M., Idreos, S., Ailamaki, A.: Nodb: efficient query execution on raw data files. In: ACM International Conference on Management of Data (SIGMOD). pp. 241–252 (2012)
2. Apache Spark. `https://spark.apache.org` (2024), accessed: 2024-10-18
3. van den Burg, G.J.J., Nazábal, A., Sutton, C.: Wrangling messy CSV files by detecting row and type patterns. Data Mining and Knowledge Discovery **33**(6), 1799–1820 (2019)
4. Cheng, Y., Rusu, F.: Parallel in-situ data processing with speculative loading. In: ACM International Conference on Management of Data (SIGMOD). pp. 1287–1298 (2014)
5. Christodoulakis, C., Munson, E.B., Gabel, M., Brown, A.D., Miller, R.J.: Pytheas: Pattern-based table discovery in CSV files. VLDB Endowment **13**(11), 2075–2089 (2020)
6. Ding, C., Tang, D., Liang, X., Elmore, A.J., Krishnan, S.: CIAO: an optimization framework for client-assisted data loading. In: IEEE International Conference on Data Engineering (ICDE). pp. 1979–1984 (2021)
7. Döhmen, T., Mühleisen, H., Boncz, P.A.: Multi-hypothesis CSV parsing. In: International Conference on Scientific and Statistical Database Management (SSDBM). pp. 16:1–16:12 (2017)
8. Fathollahzadeh, S., Boehm, M.: GIO: generating efficient matrix and frame readers for custom data formats by example. Proceedings of the ACM on Management of Data **1**(2), 120:1–120:26 (2023)
9. Gavriilidis, H., Henze, F., Zacharatou, E.T., Markl, V.: SheetReader: Efficient Specialized Spreadsheet Parsing. Information Systems **115**, 102183 (2023)
10. Ge, C., Li, Y., Eilebrecht, E., Chandramouli, B., Kossmann, D.: Speculative Distributed CSV Data Parsing for Big Data Analytics. In: ACM International Conference on Management of Data (SIGMOD). pp. 883–899 (2019)
11. Haesendonck, G., Maroy, W., Heyvaert, P., Verborgh, R., Dimou, A.: Parallel RDF generation from heterogeneous big data. In: Groppe, S., Gruenwald, L. (eds.) Proceedings of the International Workshop on Semantic Big Data, SBD@SIGMOD 2019, Amsterdam, The Netherlands, July 5, 2019. pp. 1:1–1:6. ACM (2019). https://doi.org/10.1145/3323878.3325802

12. Jiang, L., Zhao, Z.: JSONSki: streaming semi-structured data with bit-parallel fast-forwarding. In: ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). pp. 200–211 (2022)
13. Karpathiotakis, M., Alagiannis, I., Ailamaki, A.: Fast queries over heterogeneous data through engine customization. VLDB Endowment **9**(12), 972–983 (2016)
14. Kirk, D.B., Hwu, W.W.: Programming Massively Parallel Processors - A Hands-on Approach. Morgan Kaufmann (2010)
15. Koutras, C., Psarakis, K., Siachamis, G., Ionescu, A., Fragkoulis, M., Bonifati, A., Katsifodimos, A.: Valentine in action: Matching tabular data at scale. VLDB Endowment **14**(12), 2871–2874 (2021)
16. Kumaigorodski, A., Lutz, C., Markl, V.: Fast CSV loading using gpus and RDMA for in-memory data processing. In: Sattler, K., Herschel, M., Lehner, W. (eds.) Datenbanksysteme für Business, Technologie und Web (BTW). LNI, vol. P-311, pp. 19–38 (2021)
17. Langdale, G., Lemire, D.: Parsing gigabytes of JSON per second. VLDB Journal **28**(6), 941–960 (2019)
18. Li, Y., Katsipoulakis, N.R., Chandramouli, B., et al.: Mison: a fast json parser for data analytics. Proceedings of the VLDB Endowment **10**(10), 1118–1129 (2017)
19. Luong, J., Habich, D., Lehner, W.: A technical perspective of datacalc - ad-hoc analyses on heterogeneous data sources. In: IEEE International Conference on Big Data (IEEE BigData). pp. 3864–3873 (2019)
20. MongoDB. `https://www.mongodb.com` (2024), accessed: 2024-10-18
21. Mühlbauer, T., Rödiger, W., Seilbeck, R., Reiser, A., Kemper, A., Neumann, T.: Instant Loading for Main Memory Databases. VLDB Endowment **6**(14), 1702–1713 (2013)
22. Pandas. `https://pandas.pydata.org` (2024), accessed: 2024-10-18
23. Raasveldt, M., Mühleisen, H.: Duckdb: an embeddable analytical database. In: Boncz, P.A., Manegold, S., Ailamaki, A., Deshpande, A., Kraska, T. (eds.) Proceedings of the 2019 International Conference on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019. pp. 1981–1984. ACM (2019)
24. Shafranovich, Y.: Common Format and MIME Type for Comma-Separated Values (CSV) Files. RFC **4180**, 1–8 (2005)
25. Sichert, M., Neumann, T.: User-defined operators: Efficiently integrating custom algorithms into modern databases. VLDB Endowment **15**(5), 1119–1131 (2022)
26. SIMDCSV. `https://github.com/geofflangdale/simdcsv` (2024), accessed: 2024-10-18
27. TensorFlow. `https://www.tensorflow.org` (2024), accessed: 2024-10-18
28. Vitagliano, G., Hameed, M., Jiang, L., Reisener, L., Wu, E., Naumann, F.: Pollock: A data loading benchmark. VLDB Endowment **16**(8), 1870–1882 (2023)
29. Wang, J., Yang, Y., Wang, T., et al.: Big data service architecture: a survey. Journal of Internet Technology **21**(2), 393–405 (2020)
30. Xie, D., Chandramouli, B., Li, Y., Kossmann, D.: FishStore: Faster Ingestion with Subset Hashing. In: ACM International Conference on Management of Data (SIGMOD). pp. 1711–1728 (2019)