Parallel In-Situ Data Processing with Speculative Loading

Yu Cheng
UC Merced
5200 N Lake Road
Merced, CA 95343
ycheng4@ucmerced.edu

Florin Rusu
UC Merced
5200 N Lake Road
Merced, CA 95343
frusu@ucmerced.edu

ABSTRACT

Traditional databases incur a significant *data-to-query* delay due to the requirement to *load data* inside the system before querying. Since this is not acceptable in many domains generating massive amounts of raw data, e.g., genomics, databases are entirely discarded. *External tables*, on the other hand, provide instant SQL querying over raw files. Their performance across a query workload is limited though by the speed of repeated full scans, tokenizing, and parsing of the entire file.

In this paper, we propose SCANRAW, a novel database physical operator for in-situ processing over raw files that integrates data loading and external tables seamlessly while preserving their advantages: optimal performance across a query workload and zero time-to-query. Our major contribution is a *parallel super-scalar pipeline implementation* that allows SCANRAW to take advantage of the current many- and multi-core processors by overlapping the execution of independent stages. Moreover, SCANRAW overlaps query processing with loading by *speculatively* using the additional I/O bandwidth arising during the conversion process for storing data into the database such that subsequent queries execute faster. As a result, SCANRAW makes optimal use of the available system resources – CPU cycles and I/O bandwidth – by switching *dynamically* between tasks to ensure that optimal performance is achieved.

We implement SCANRAW in a state-of-the-art database system and evaluate its performance across a variety of synthetic and real-world datasets. Our results show that SCANRAW with speculative loading achieves optimal performance for a query sequence at any point in the processing. Moreover, SCANRAW maximizes resource utilization for the entire workload execution while speculatively loading data and without interfering with normal query processing.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design—*access methods*; H.2.4 [**Database Management**]: Systems—*query processing*

1. INTRODUCTION

In the era of data deluge, massive amounts of data are generated at an unprecedented scale by applications ranging from social networks to scientific experiments and personalized medicine. The vast majority of these read-only data are stored as application-specific files containing hundreds of millions of records. Due to the upfront loading cost and the proprietary file format, *databases* are rarely considered as a storage solution even though they provide enhanced querying functionality and performance [10, 5]. Instead, the standard practice is to write dedicated applications encapsulating the query logic on top of *generic file access libraries* that provide instant access to data through a well-defined API. While a series of applications for a limited set of parametrized queries are provided with the library, new queries typically require the implementation of a completely new application even when there is significant logic that can be reused. Relational databases avoid this problem altogether by implementing a declarative querying mechanism based on SQL. This requires though data representation independence—achieved through loading and storing data in a proprietary format.

External tables [25, 17] combine the advantages of file access libraries and the declarative query execution mechanism provided by SQL—data can be queried in the original format using SQL. Thus, there is no loading penalty and querying does not require the implementation of a complete application. There is a price though. When compared to standard database query optimization and processing, external tables use linear scan as the single file access strategy since no storage optimizations are possible—data are external to the database. Every time data are accessed, they have to be converted from the raw format into the internal database representation. As a result, query performance is both constant and poor. Databases, on the other hand, trade query performance for a lengthy loading process. Although time-consuming, data loading is a one-time process amortized over the execution of a large number of queries. The more queries are executed, the more likely is that the database outperforms external tables in response time.

Motivating example. To make our point, let us consider a representative example from genomics. SAM/BAM files [9] – SAM are text, BAM are binary – are the standard result of the next-generation genomic sequence aligners. These files consist of a series of tuples – known as reads – encoding how fragments of the sequenced genome align relative to a reference genome. There are hundreds of millions of reads in a standard SAM/BAM file from the *1000 Genomes* project [2]. Each read contains 11 mandatory fields and a variable number of optional fields. There is one such read on every line in the SAM file—the fields are tab-delimited.

A representative type of processing executed over SAM/BAM files is variant, i.e., genome mutation responsible for causing hereditary diseases, identification [1]. It requires computing the distribution of the CIGAR field [9] across all the reads overlapping a position in the genome where certain patterns occur in at least one read. This can be expressed in SQL as a standard group-by aggregate

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
Copyright 2014 ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2593673.

query and executed inside a database using the optimal execution plan selected by the query optimizer based on data statistics. Geneticists do not use databases though. Their solution to answer this query – and any other query for that matter – is to write application programs on top of generic file access libraries such as SAMtools [9] and BAMTools [7] that provide instant access to the reads in the file through a well-defined API. Overall, a considerably more intricate procedure than writing a SQL query.

Problem statement. We consider the general problem of *executing SQL-like queries in-situ over raw files*, e.g., SAM/BAM, with a database engine. Data converted to the database processing representation at query time can be loaded into the database. Our objective is to design a solution that provides instant access to data and also achieves optimal performance when the workload consists of a sequence of queries. There are two aspects to this problem. First, methods that provide single-query optimal execution over raw files have to be developed. These can be applied both to external table processing as well as standard data loading. And second, a mechanism for query-driven gradual data loading has to be devised. This mechanism interferes minimally – if at all – with normal query processing and guarantees that converted data are loaded inside the database for every query. If sufficient queries are executed, all data get loaded into the database. We assume the existence of a procedure to extract tuples with a specified schema from the raw file and to convert the tuples into the database processing format.

Contributions. The major contribution we propose in this paper is `SCANRAW`—a novel database physical operator for in-situ processing over raw files that integrates data loading and external tables seamlessly while preserving their advantages—optimal performance across a query workload and zero time-to-query. `SCANRAW` has a *parallel super-scalar pipeline architecture* that overlaps data reading, conversion into the database representation, and query processing. `SCANRAW` implements *speculative loading* as a gradual loading mechanism to store converted data inside the database. The main idea in speculative loading is to find those time intervals during raw file query processing when there is no disk reading going on and use them for database writing. The intuition is that query processing speed is not affected since the execution is CPU-bound and the disk is idle.

Our specific contributions can be summarized as follows:

- Design `SCANRAW`, the first parallel super-scalar pipeline operator for in-situ processing over raw data. The stages in the `SCANRAW` pipeline are identified following a detailed analysis of data loading and external table processing.
- Design speculative loading as a gradual data loading mechanism that takes advantage dynamically of the disk idle intervals arising during data conversion and query processing.
- Implement `SCANRAW` in the DataPath [6] database system and evaluate its performance across a variety of synthetic and real-world datasets. Our results show that `SCANRAW` with speculative loading achieves optimal performance for a query sequence at any point in the processing.

Overview. In Section 2 we provide a formal characterization for in-situ data processing over raw files. The `SCANRAW` architecture and operation are introduced in Section 3 while speculative loading is presented in Section 4. Experimental results are shown in Section 5. We conclude with a detailed look at related work (Section 6) and plans for future work (Section 7).

2. RAW FILE QUERY PROCESSING

Figure 1 depicts the generic process that has to be followed in order to make querying over raw files possible. The input to the process is a raw file – SAM/BAM in our running example – a schema, and a procedure to extract tuples with the given schema from the raw file. The output is a tuple representation that can be processed by the execution engine. For each stage, we introduce trade-offs involved and possible optimizations. Before discussing in detail the stages of the conversion process though, we emphasize the generality of the procedure. Stand-alone applications and databases alike have to read data stored in files and convert them to an in-memory representation suitable for processing. They all follow some or all of the stages depicted in Figure 1.

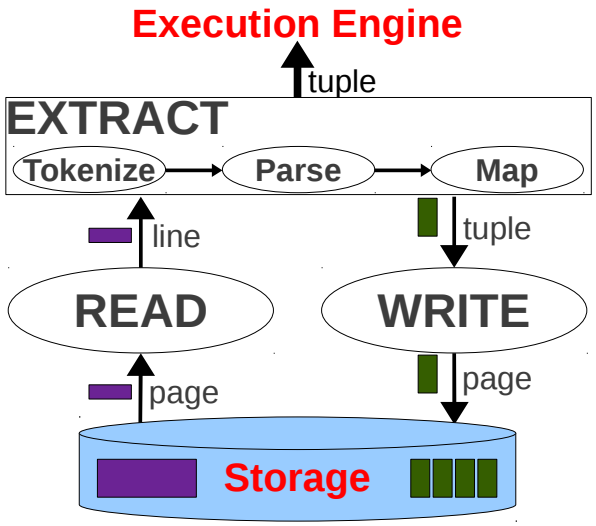


Figure 1: Query processing over raw files.

READ. The first stage of the process requires reading data from the original flat file. Without additional information about the structure or the content – stored inside the file or in some external structure – the entire file has to be read the first time it is accessed. This involves reading the lines of the file one-by-one and passing them to `EXTRACT`. As an optimization – already implemented by the *file system* – multiple lines co-located on the same page are read together. An additional optimization – also implemented by the file system – is the caching of pages in memory buffers such that future requests to the same page can be served directly from memory without accessing the disk. Thus, while the first access is limited by the disk throughput, subsequent accesses can be much faster as long as data are already cached.

Further reading optimizations beyond the ones supported by default by the file system aim at reducing the amount of data – the number of lines – retrieved from disk and typically require the creation of auxiliary data structures, i.e., *indexes*. For example, if the tuples are sorted on a particular attribute range queries over that attribute can be answered by reading only those tuples that satisfy the predicate and a few additional ones used in the binary search to identify the range. Essentially, any type of index built inside a database can be also applied to flat files by incurring the same or higher construction and maintenance costs. In the case of our genomic example, BAI files [7] are indexes built on top of BAM files. Columnar storage and compression are other strategies to minimize the amount of data read from disk—orthogonal to our discussion.

TOKENIZE. Abstractly, `EXTRACT` transforms a tuple from text format into the processing representation based on the schema provided and using the extraction procedure given as input to the process. We decompose `EXTRACT` into three stages – `TOKENIZE`, `PARSE`, and `MAP` – with independent functionality. Taking a text

line corresponding to a tuple as input, TOKENIZE is responsible for identifying the attributes of the tuple. To be precise, the output of TOKENIZE is a vector containing the starting position for every attribute in the tuple. This vector is passed along with the text into PARSE. The implementation of TOKENIZE is quite simple. Iterate over the text line character-by-character, identify the delimiter character that separates the attributes, and store the corresponding position in the output vector. To avoid copying, the delimiter can be replaced with the end-of-string character. Overall, a *linear scan* over the text line with little opportunities for optimization.

A first optimization is aimed at *reducing the size of the linear scan* and is applicable only when a subset of attributes have to be converted in the processing representation, i.e., selective tokenizing and parsing [5]. The idea is to stop the linear scan over the text as soon as the end of the last attribute to be converted is identified. Maximum reductions are obtained when the length of the text is large and the attributes are located at the edges—we can go for a backward scan if the length is shorter. A second optimization is targeted at *saving the work done*, i.e., the vector of positions or positional map [5], from one conversion to another. Essentially, when the vector is passed to PARSE, it is also cached in memory. The positional map can be complete or partially filled—when combined with adaptive tokenizing. While a complete map allows for immediate identification of the attributes, a partial map can provide significant reductions even for the attributes whose positions are not stored. The idea is to find the position of the closest attribute already in the map and scan forward or backward from there.

PARSE. In PARSE, attributes are converted from text format into the binary representation corresponding to their type. This typically involves the invocation of a function that takes as input a string parameter and returns the attribute type, e.g., atoi. The input string is part of the text line. Its starting position is determined in TOKENIZE and passed along in the positional map. Intuitively, the higher the number of function invocations, the higher the cost of parsing. Since the only direct optimization – implement faster conversion functions – is a well-studied problem with clear solutions, alternative optimizations target other aspects of parsing. Selective parsing [5] is an immediate extension of selective tokenizing aimed at *reducing the number of conversion function invocations*. Only the attributes required by the current processing are converted. If processing involves selections, the number of conversions can be reduced further by first parsing the attributes which are part of selection predicates, evaluating the condition, and, only if satisfied, parsing the remaining attributes. In the case of highly selective predicates and queries over a large number of attributes, this push-down selection technique [5] can provide significant reductions in parsing time. The other possible optimization is to *cache the converted attributes in memory* such that subsequent processing does not require parsing anymore since data are already in memory in binary format.

MAP. The last stage of extraction is MAP. It takes the binary attributes converted in PARSE and organizes them in a data structure suitable for processing. In the case of a row-store execution engine, the attributes are organized in a record. For column-oriented processing, an array of the corresponding type is created for each attribute. Although not a source of significant processing, this reorganization can become expensive if not implemented properly. Copying data around has to be avoided and replaced with memory mapping whenever possible.

At the end of EXTRACT, data are loaded in memory and ready for processing. Multiple paths can be taken at this point. In external tables, data are passed to the execution engine for query processing and discarded afterwards. In NoDB [5] and in-memory databases, data are kept in memory for subsequent processing. READ and EXTRACT do not have to be executed anymore as long as data are already cached. In standard database loading, data are first written to disk and only then query processing can begin. This typically requires reading data again—from the database though. It is important to notice that these stages have to be executed for any type of processing and for any type of raw data, not only text. In the case of binary raw data though the bulk of processing is very likely to be concentrated in MAP instead of TOKENIZE and PARSE.

WRITE. WRITE is present only in database loading. Data converted in the processing representation is stored in this format such that subsequent accesses do not incur the tokenization and parsing cost. The price is the storage space and the time to write data to disk. Since READ and WRITE contend for I/O throughput, their disk access has to be carefully synchronized in order to minimize the interference. The typical sequential solution is to read a page, convert the tuples from text to binary, write them as a page, and then repeat the entire process for all the pages in the input raw file. This READ-EXTRACT-WRITE pattern guarantees non-overlapping access to disk. An optimization that is often used in practice is to buffer as many pages with converted tuples as possible in memory and to flush them at once when the memory is full.

The interaction between WRITE and the various optimizations implemented in TOKENIZE and PARSE raises some complicated trade-offs. If query-driven partial loading is supported, the database has to provide mechanisms to store incomplete tuples inside a table. A simple solution – the only available in the majority of database servers – is to implement loading with INSERT and UPDATE SQL statements. The effect on performance is very negative though. The situation gets less complicated in column-oriented databases, e.g., MonetDB [21], which allow for efficient schema expansion by adding new columns. Loading new attributes reduces to writing the pages with their binary representation in this case. Push-down selection in PARSE complicates everything further since only the tuples passing the selection predicate end-up in the database. To enforce that a tuple is processed only once – either from the raw file or from the database – detailed bookkeeping has to be set in place. While the effect on a single query might be positive, it is very likely that the overhead incurred across multiple queries is too high to consider push-down selection in PARSE as a viable optimization. This is true even without loading data into the database.

3. THE SCANRAW OPERATOR

In this section, we consider *single query execution* over raw data. Given a set of raw files and a SQL-like query, the objective is to minimize query execution time. The fundamental research question we ask is how to design a parallel in-situ data processing operator targeted at the current many- and multi-core processors? What architectural choices to make in order to take full advantage of the available parallelism? How to integrate the operator with a database server? We propose SCANRAW, a novel physical operator implementing query processing over raw data based on the decomposition presented in Section 2. Our major contribution is a *parallel super-scalar pipeline architecture* that allows SCANRAW to overlap the execution of independent stages. SCANRAW overlaps reading, tokenizing, and parsing with the actual processing across data partitions in a pipelined fashion, thus allowing for multiple partitions to be processed in parallel both across stages and inside a conversion stage. Each stage can itself be sequential or parallel.

To the best of our knowledge, SCANRAW is the first operator that provides generic query processing over raw files using a fully parallel super-scalar pipeline implementation. The other solutions proposed in the literature are sequential or, at best, use data partition-

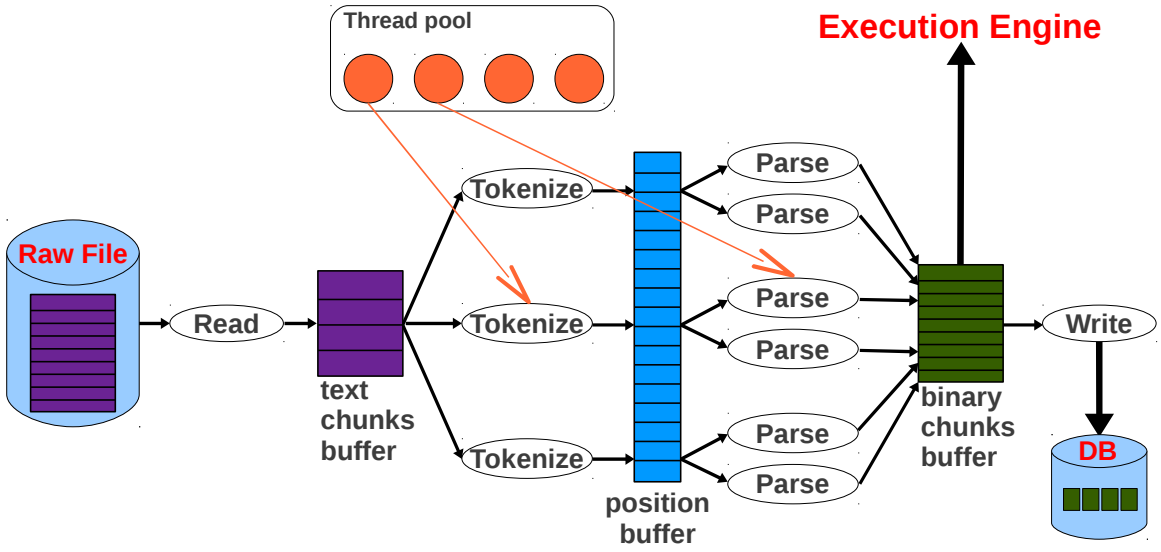


Figure 2: SCANRAW architecture.

ing parallelism [8]—also implemented in SCANRAW. Some solutions follow the principle READ-EXTRACT-PROCESS, e.g, external tables and NoDB [10, 5], while others [4] operate on a READ-EXTRACT-LOAD-PROCESS pattern. In SCANRAW, the processing pattern is dynamic and is determined at runtime based on the available system resources. By default, SCANRAW operates as a parallel external table operator. Whenever I/O bandwidth becomes available during processing – due to query execution or to conversion into the processing representation – SCANRAW switches automatically to partial data loading by overlapping conversion, processing, and loading. In the extreme case, all data accessed by the query are loaded into the database, i.e., SCANRAW acts as a query-driven data loading operator.

3.1 Architecture

The super-scalar pipeline architecture of SCANRAW is depicted in Figure 2. Although based on the abstract process representation given in Figure 1, there are significant structural differences. Multiple TOKENIZE and PARSE stages are present. They operate on different portions of the data in parallel, i.e., data partitioning parallelism [8]. MAP is not an independent stage anymore. In order to simplify the presentation, we consider it is contained in PARSE. The scheduling of these stages is managed by a scheduler controlling a pool of worker threads. The scheduler assigns worker threads to stages dynamically at runtime. READ and WRITE are also controlled by the scheduler thread in order to coordinate disk access optimally and avoid interference. The scheduling policy for WRITE dictates the SCANRAW behavior. If the scheduler never invokes WRITE, SCANRAW becomes a parallel external table operator. If the scheduler invokes WRITE for every tuple – or group of tuples, to be precise – SCANRAW degenerates into a parallel Extract-Transform-Load (ETL) operator. While both these scheduling policies are supported in SCANRAW, we propose a completely different WRITE behavior—*speculative loading* (Section 4). The main idea is to trigger WRITE only when READ is blocked due to the text chunks buffer being full. Remember that our objective is to minimize execution time not to maximize the amount of loaded data.

Dynamic structure. The structure of the super-scalar pipeline can be static – the case in CPU design – or dynamic. In a static structure, the number of stages and their interconnections are set ahead of operation and they do not change. Since the optimal pipeline structure is different across datasets, each SCANRAW instance has to be configured accordingly. For example, a file with 200 numeric attributes per tuple requires considerably more PARSE stages than a file with 200 string attributes per tuple. SCANRAW avoids this problem altogether since it has a dynamic pipeline structure [20] that configures itself according to the input data. Whenever data become available in one of the buffers, a thread is extracted from the thread pool and is assigned the corresponding operation and the data for execution. The maximum degree of parallelism that can be achieved is equal to the number of threads in the pool. The number of threads in the pool is configured dynamically at runtime for each SCANRAW instance. Data that cannot find an available thread are stored in the corresponding buffer until a thread becomes available. This effect is back-propagated through the pipeline structure downto READ which stops producing data when no empty slots are available in the text chunk buffer.

Buffers. Buffers are characteristic to any pipeline implementation and operate using the standard producer-consumer paradigm. The stages in the SCANRAW pipeline act as producers and consumers that move chunks of data between buffers. The entire process is regulated by the size of the buffers which is determined based on memory availability. The *text chunk buffer* contains text fragments read from the raw file. The file is logically split into horizontal portions containing a sequence of lines, i.e., chunks. Chunks represent the reading and processing unit. The *position buffer* between TOKENIZE and PARSE contains the text chunks read from the file and their corresponding positional map computed in TOKENIZE. Finally, *binary chunks buffer* contains the binary representation of the chunks. This is the processing representation used in the execution engine as well as the format in which data are stored inside the database. In binary format, tuples are vertically partitioned along columns represented as arrays in memory. When written to disk, each column is assigned an independent set of pages which can be directly mapped into the in-memory array representation. It is important to emphasize that not all the columns in a table have to be present in a binary chunk.

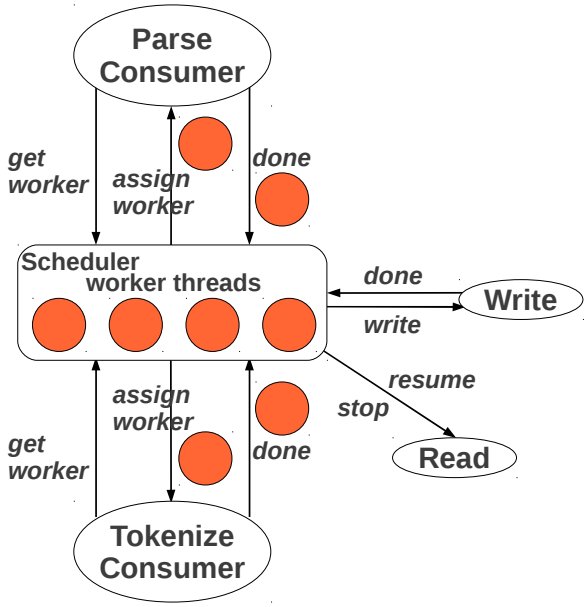


Figure 3: SCANRAW threads and control messages.

Caching. While each of the buffers present in the SCANRAW architecture can act as a cache if the same instance of the operator is employed across multiple query plans, the only buffer that makes sense to operate as a cache is the binary chunks buffer. There are two reasons for this. First, caching raw file chunks takes memory space from the binary chunks cache. Why do not cache more binary chunks, if possible? Moreover, the file system buffers act as an automatic caching mechanism for the raw file. Second, the other entity that can be cached is the positional map generated by TOKENIZE and stored in the position buffer. While this also takes space from the binary chunks cache, the main reason it has little impact for SCANRAW is that it cannot avoid reading the raw file and parsing. These two stages are more likely to be the bottleneck than TOKENIZE which requires only an adequate degree of parallelism to be fully optimized.

The binary chunks buffer provides caching for the converted chunks. Essentially, all the chunks in the raw file end up in the binary chunks cache—not necessarily at the same time. From there, the chunks are passed into the execution engine – external tables processing – or to WRITE for storing inside the database—data loading. What makes SCANRAW special is that in addition to executing any of these tasks in isolation, it can also combine their functionality. The binary chunks cache plays a central role in configuring the SCANRAW functionality. By default, all the converted binary chunks are cached, i.e., they are not eliminated from the cache once passed into the execution engine or WRITE. If all the chunks in the raw file can be cached, SCANRAW simply delivers the chunks to the execution engine and the database becomes an in-memory database. Chunks are expelled from the cache using the standard LRU cache replacement policy biased toward chunks loaded inside the database, i.e., chunks stored in binary format are more likely to be replaced.

Pre-fetching. SCANRAW functions as a self-driven asynchronous process with the objective to produce chunks for the execution engine as fast as possible. It is not a pull-based operator that strictly satisfies requests. Essentially, SCANRAW pre-fetches chunks continuously and caches them in the binary chunks buffer. This process stops only when the buffer is full with chunks not already processed by the execution engine. Processed chunks are replaced using the cache replacement policy. They can be either dropped altogether or stored in the database—if the necessary I/O throughput is available. Notice that pre-fetching works both for raw chunks as well as for chunks already loaded in the database and it is regulated by the operation of the binary chunks cache, i.e., SCANRAW and the execution engine synchronize through the binary chunks cache.

3.2 Operation

Given the architectural components introduced previously, in this section we present how they interact with each other. At a high level, SCANRAW consists of a series of asynchronous stand-alone threads corresponding to the stages in Figure 2. The stand-alone threads – depicted in Figure 3 – communicate through control messages while data are passed through the architectural buffers. Notice that these threads – READ, WRITE, TOKENIZE, PARSE, and SCHEDULER – are separate from the thread pool which contains worker threads that are configured dynamically with the task to execute. In the following, we first present each of the stand-alone threads and their operation. Then we discuss the types of work performed by the thread-pool workers.

3.2.1 Stand-Alone Threads

READ thread. The READ thread reads chunks asynchronously from the raw file and deposits them in the text chunks buffer. READ stops producing chunks when the buffer is full and restarts when there is at least an empty slot. The scheduler can force READ to stop/resume in order to avoid disk interference with WRITE. If the raw file is read for the first time, sequential scan is the only alternative. If the file was read before, a series of optimizations can be applied. First, chunks can be read in other order than sequential or they can be ignored altogether if the selection predicate cannot be satisfied by any tuple in the chunk. This can be checked from the minimum/maximum values stored in the metadata. Second, cached chunks can be processed immediately from memory. And third, chunks loaded inside the database can be read directly in the binary chunks buffer without any tokenizing and parsing. When all the optimizations can be applied, SCANRAW delivers the chunks to the execution engine in the following order. First, the cached chunks, followed by the chunks loaded in the database, and finally the chunks read from the raw file.

An interesting situation arises when only some of the columns required for query processing are loaded either in cache or in the database. Since raw file access is required in this case to read the remaining columns, SCANRAW has to decide what is optimal: use extra CPU cycles to tokenize and parse all the required columns? or read the already loaded columns from the database and convert only the additional columns? While the second alternative is more likely to be optimal when the chunk is cached or when a large part of the required columns are loaded in the database, it imposes severe restrictions on cache operation since that is where chunks are assembled. If the execution is I/O-bound converting all the columns from the raw file is the optimal choice since it avoids additional reading. This is the implementation used throughout our experiments which are I/O-bound when sufficient threads are available.

WRITE thread. The WRITE thread is responsible for storing binary chunks inside the database. Essentially, WRITE extracts chunks from the binary chunks buffer and materializes them to disk in the database representation. It also updates the catalog metadata accordingly. SCANRAW has to enforce that only one of READ or WRITE accesses the disk at any particular instant in time. This is necessary in order to reduce disk interference and maximize I/O throughput.

Consumer threads. A consumer thread monitors each of the internal buffers in the SCANRAW architecture. TOKENIZE consumer monitors the text chunks buffer while PARSE consumer monitors the position buffer, respectively. Whenever a chunk becomes available in any of these buffers, work has to be executed by one of the workers in the thread pool. The consumer thread is responsible for acquiring the worker thread, scheduling its execution on a chunk, and moving the result data in the subsequent buffer. It is important to emphasize that chunk processing is executed by the worker thread not the consumer thread. For example, the TOKENIZE consumer makes a request to the thread pool whenever a chunk is ready for tokenizing. Multiple such requests can be pending at the same time. Once a worker thread is allocated, the requesting chunk is extracted from the buffer and sent for processing. This triggers READ to produce a new chunk if the buffer is not full. When the processing is done, the TOKENIZE consumer receives back the worker thread and the chunk and its corresponding positional map. It releases the worker thread and inserts the result data into the position buffer. PARSE consumer follows similar logic.

Scheduler thread. The scheduler thread is in charge of managing the thread pool and satisfying the requests made by the consumer threads monitoring the buffers. Whenever a request can be satisfied, the scheduler extracts a thread from the pool and returns it to the requesting consumer thread. Notice that even if a thread is available, it can only be allocated if there is empty space in the destination buffer. Otherwise, the result chunk cannot move forward. For example, a request from the PARSE consumer can be accomplished only if there is empty space in the binary chunks buffer. The scheduler requires access to all the buffers in the architecture in order to take the optimal decision in assigning worker threads. The objective is to have all the threads in the pool running while moving chunks fast enough through the pipeline such that the execution engine is always busy. At the same time, the scheduler has to make sure that progress is always possible and the pipeline does not block. While designing a scheduling algorithm that guarantees progress is an achievable task, designing an optimal algorithm is considerably more complicated. For this reason, it is common in practice to develop heuristics that guarantee correctness while providing some kind of optimality in certain conditions. This is exactly the approach we take in SCANRAW. In our implementation, the scheduler uses a best-effort heuristic and satisfies requests in the order in which they are received while guaranteeing progress.

3.2.2 Worker Threads

Stand-alone threads are static. The task they perform is fixed at implementation. Worker threads, on the other hand, are dynamically configured at runtime with the task they perform. As a general rule, stand-alone threads perform management tasks that control the data flow through the pipeline while worker threads perform the actual data processing. Since the entire process of assigning threads incurs overhead, it has to be the case that the time taken by data processing offsets the overhead. This is realized by making tasks operating over chunks or vectors of tuples rather than individual tuples. As depicted in Figure 2, two types of tasks can be assigned to worker threads—TOKENIZE and PARSE. The operations that are executed by each of them and possible optimizations are discussed in Section 2. The scheduler selects the task to assign to each worker from a set of requests made by the corresponding consumer threads.

3.3 Integration with a Database

Query processing. At a high level, SCANRAW is similar to the database *heap scan* operator. Heap scan reads the pages corresponding to a table from disk, extracts the tuples, and maps them in the internal processing representation. Relative to the process depicted in Figure 1, the extract phase in heap scan consists of MAP only. There is no TOKENIZE and PARSE. It is natural then for the database engine to treat SCANRAW similar to the heap scan operator and place it in the leaves of query execution plans. Moreover, SCANRAW morphs into heap scan as data are loaded in the database. The main difference between SCANRAW and heap scan though – and any other standard database operator for that matter – is that SCANRAW is not destroyed once a query finishes execution. This is because SCANRAW is not attached to a query but rather to the raw file it extracts data from. As such, the state of the internal buffers is preserved across queries in order to guarantee improved performance—not the case for the standard heap scan. When a new query arrives, the execution engine first checks the existence of a corresponding SCANRAW operator. If such an operator exists, it is connected to the query execution plan. Only otherwise it is created. When is a SCANRAW instance completely deleted then? Whenever it loaded the entire raw file into the database.

Query optimization. To effectively use SCANRAW in query optimization, additional data, i.e., statistics, have to be gathered. This is typically done as a stand-alone process executed at certain time intervals. In the case of SCANRAW, statistics are collected while data are converted in the database representation which is triggered in turn by query processing. Statistics are stored in the metadata catalog. The types of statistics collected by SCANRAW include the position in the raw file where each chunk starts and the minimum/maximum value corresponding to each attribute in every chunk. More advanced statistics such as the number of distinct elements and the skew of an attribute – or even samples – can be also extracted during the conversion stage. The collected statistics are later used for two purposes. First, the number of chunks read from disk can be reduced in the case of selection predicates. For this to be effective though, data inside the chunk have to be clustered. While WRITE can sort data in each chunk prior to loading, SCANRAW does not address the problem of completely sorting and reorganizing data based on queries, i.e., database cracking [22]. The second use case for statistics is cardinality estimation for traditional query optimization.

Resource management. SCANRAW resources are allocated dynamically at runtime by the database resource manager in order to better control the operation and to optimize system utilization. For this to be possible though, measurements accounting for CPU usage and memory utilization have to be taken and integrated in the resource allocation procedure. The scheduler is in the best position to monitor resource utilization since it manages the allocation of worker threads from the pool and inspects buffer utilization. These data are relayed to the database resource manager as requests for additional resources or are used to determine when to release resources. For example, if the scheduler assigns all the worker threads in the pool for task execution but the text chunks buffer is still full – SCANRAW is CPU-bound – additional CPUs are needed in order to cope with the I/O throughput. With respect to memory utilization, it makes sense to allocate more memory to SCANRAW instances that are used more often since this increases the rate of data reuse across queries.

4. SPECULATIVE LOADING

By default, SCANRAW operates as a parallel external table operator. It provides instant access to data without pre-loading. This results in optimal performance for the first query accessing the raw data. What about a workload consisting of a sequence of queries? What is the SCANRAW behavior in that case? The default external

table regime is sub-optimal since tokenizing and parsing have to be executed again and again. Ideally, only useful work, i.e., reading and processing, should be performed starting with the second data access. Traditional databases achieve this by pre-loading the data in their processing format. They give up instant data access though.

Is it possible to achieve both optimal execution time for all the queries in a sequential workload and instant data access for the first query? This is the research question we ask in this section. Our solution is *speculative loading*. In speculative loading, instant access to data is guaranteed. It is also guaranteed that subsequent queries accessing the same data execute faster and faster, achieving database performance at some point—the entire data accessed by the query are read from the database at that time. Moreover, speculative loading achieves database performance the earliest possible while preserving optimal execution time for all the queries in-between. In some cases this is realized from the second query. The main idea in speculative loading is to find those time intervals during raw file query processing when there is no disk reading going on and use them for database writing. The intuition is that query processing speed is not affected since the execution is CPU-bound and the disk is idle. Notice though that a highly-parallel architecture consisting of asynchronous threads capable to detect free I/O bandwidth and overlap processing with disk operations is required in order to implement speculative loading. `SCANRAW` accomplishes these requirements.

There are two solutions in the literature that are related to speculative loading. In invisible loading [4], a fixed amount of data – specified as a number of chunks – are loaded for every query even if that slows down the processing. In fact, invisible loading increases execution time for all the queries accessing raw data. NoDB [5] achieves optimal execution time for all the queries in a workload only when all the accessed data fit in memory. Loading is not considered in NoDB. Only in-memory caching. A possible extension to NoDB – explored in [10] – is to flush data into the database when the memory is full. This results in oscillating query performance, i.e., whenever flushing is triggered query execution time increases.

How does speculative loading work? The central idea in speculative loading is to let `SCANRAW` decide adaptively at runtime what data to load, how much, and when while maintaining optimal query execution performance. These decisions are taken dynamically by the scheduler, in charge of coordinating disk access between `READ` and `WRITE`. Since the scheduler monitors the utilization of the buffers and assigns worker threads for task execution, it can identify when `READ` is blocked. This can happen for two reasons. First, conversion from raw format to database representation – tokenizing and parsing – is too time-consuming. Second, query execution is the bottleneck. In both cases, processing is CPU-bound. At that time, the scheduler signals `WRITE` to load chunks in the database. While the maximum number of chunks to be loaded is determined by the scheduler based on the pipeline utilization, the actual chunks are strictly determined by `WRITE` based on the catalog metadata. In order to minimize the impact on query execution performance, only the "oldest" chunk in the binary cache that was not previously loaded into the database is written at a time. This increases the chance to load more chunks before they are eliminated from the cache. It is important for the scheduler not to allow reading start before writing finishes in order to avoid disk interference. This is realized with the `resume` control message (Figure 3) whenever worker threads become available and `WRITE` returns.

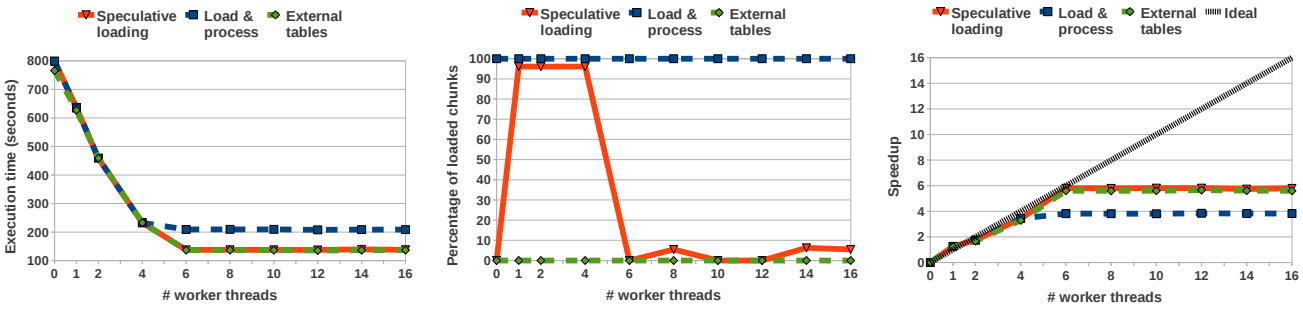
Why does speculative loading not interfere with query execution? Speculative loading is triggered only when there is no disk utilization. Rather than let the disk idle, this "dead" time is used for loading—a task with minimal CPU usage that has little to no impact on the overall CPU utilization. Especially for the modern multi-core processors with a high degree of parallelism. What about memory interference in the binary chunks cache? Something like this can happen only when the chunk being written to disk has to be expelled from the cache. As long as there is at least one other chunk already processed, that chunk can be eliminated instead. The larger the cache size, the higher the chance to find such a chunk.

How do we guarantee that new chunks are loaded for every query? Since speculative loading is entirely driven by resource utilization in the system, there is no guarantee that new chunks will get loaded for every query. For example, if I/O is the bottleneck in query processing, no loading is possible whatsoever. Thus, we have to develop a *safeguard mechanism* that enforces a minimum amount of loading but without decreasing query processing performance. Our solution is based on the following observation. At the end of a scan over the raw file, the binary chunks cache contains a set of converted chunks that are kept there until the next query starts scanning the file. These chunks are the perfect candidates to load in the database. Writing can start as soon as the last chunk was read from the raw file—not necessarily after query processing finishes. Moreover, the next query can be admitted immediately since it can start processing the cached chunks first. Only the reading of new chunks from disk has to be delayed until flushing the cache to disk. This is very unlikely to affect query performance though. If it does, an alternative solution is to delay the admission of the next query until flushing the cache is over—a standard procedure in multi-query processing. It is important to emphasize that the safeguard mechanism is the norm for invisible loading [4] while in speculative loading it is invoked only in rare circumstances. There is nothing stopping us to invoke it for every query though.

How does speculative loading improve performance for a sequence of queries? The chunks written to the database do not require tokenization and parsing. This guarantees improved query performance as long as new data are loaded for every query. The safeguard mechanism enforces chunk loading independent of the system resource utilization. In order to show how speculative loading improves query execution, we provide an illustrative example. Since the amount of data loaded due to resource utilization is non-deterministic – thus hard to illustrate – we focus on the safeguard mechanism. For the purpose of this example, we assume that the safeguard is invoked after every query. Consider a raw file consisting of 8 chunks. The binary cache can contain 2 chunks. The first query that accesses the file reads all the data and converts them to binary representation. For simplicity, assume that chunks are read and processed in sequential order. At the end of the first query, chunk 7 and 8 reside in the cache. Thus, the safeguard mechanism flushes them to the database. Query 2 processes the chunks in the order {7, 8, 1, 2, 3, 4, 5, 6}, with chunk 7 and 8 delivered from the cache. Since fewer chunks are converted from raw format, query 2 runs faster than query 1. At the end of query 2, chunk 5 and 6 reside in the cache and they are flushed to the database. Query 3 processes the chunks in the order {5, 6, 7, 8, 1, 2, 3, 4}. The first two chunks are in the cache, the next two are read from the database without tokenizing and parsing while only the remaining 4 are converted from the raw file. This makes query 3 execute faster than query 2. Repeating the same process, chunk 3 and 4 are loaded in the database at the end of query 3 and by the end of query 4 all data are loaded.

5. EXPERIMENTAL EVALUATION

The objective of the experimental evaluation is to investigate the `SCANRAW` performance across a variety of datasets – synthetic and real – and workloads—including a single query as well as a se-



(a) (b) (c)

Figure 4: Execution time (a), percentage of loaded data (b), and speedup (c) as a function of the number of worker threads.

quence of queries. Additionally, the sensitivity of the operator is quantified with respect to many configuration parameters. Specifically, the experiments we design are targeted to answer the following questions:

- How is parallelism improving SCANRAW performance and what speedup does SCANRAW achieve?
- What is the performance of speculative loading compared to external tables and database loading & processing, respectively, for a single query? For a sequence of queries?
- Where is the time spent in the pipeline? How does the dynamic SCANRAW architecture adapt to data characteristics? Query characteristics?
- What resource utilization does SCANRAW achieve?
- How does SCANRAW performance compare to other file access libraries?

Implementation. SCANRAW is implemented as a C++ prototype. Each stand-alone thread as well as the workers are implemented as pthread instances. The code contains special function calls to harness detailed profiling data. In the experiments, we use SCANRAW implementations for CSV and tab-delimited flat files, as well as BAM [7] binary files. Adding support for other file formats requires only the implementation of specific TOKENIZE and PARSE workers without changing the basic architecture. We integrate SCANRAW with a state-of-the-art multi-thread database system [6] shown to be I/O-bound for a large class of queries. This guarantees that query processing is not the bottleneck except in rare situations and allows us to isolate the SCANRAW behavior for detailed and accurate measurements. Notice though that integration with a different database requires mapping to a different processing representation without changes to SCANRAW architecture.

System. We execute the experiments on a standard server with 2 AMD Opteron 6128 series 8-core processors (64 bit) – 16 cores – 40 GB of memory, and four 2 TB 7200 RPM SAS hard-drives configured RAID-0 in software. Each processor has 12 MB L3 cache while each core has 128 KB L1 and 512 KB L2 local caches. The storage system supports 240, 436 and 1600 MB/second minimum, average, and maximum read rates, respectively—based on the Ubuntu disk utility. The cached and buffered read rates are 3 GB/second and 565 MB/second, respectively. Ubuntu 12.04.3 SMP 64-bit with Linux kernel 3.2.0-56 is the operating system.

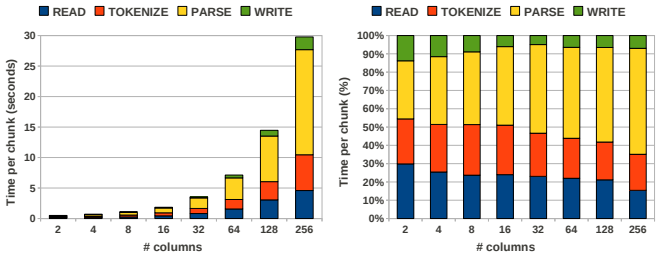
Methodology. We perform all experiments at least 3 times and report the average value as the result. If the experiment consists of a single query, we always enforce data to be read from disk by cleaning the file system buffers before execution. In experiments over a sequence of queries, the buffers are cleaned only before the first query. Thus, the second and subsequent queries can access cached data.

5.1 Micro-Benchmarks

Data. We generate a suite of synthetic CSV files in order to study SCANRAW sensitivity in a controlled setting. There are between 2^{20} and 2^{28} lines in a file in powers of 4 increments. Each line corresponds to a database tuple. The number of columns in a tuple ranges from 2 to 256 in powers of two. Overall, there are 40 files in the suite, i.e., 5 numbers of tuples times 8 numbers of columns. The smallest file contains 2^{20} rows and 2 columns – 20 MB – while the largest is 638 GB in size—2^{28} rows with 256 columns each. The value in each column is a randomly-generated unsigned integer smaller than 2^{31}. The dataset is modeled based on [5, 4]. While we execute the experiments for every file, unless otherwise specified, we report results only for the configuration $2^{26} \times 64$—40 GB in text format.

Query. The query used throughout experiments has the form SELECT SUM($\sum_{j=1}^{K} C_{i_j}$) FROM FILE where K columns C_{i_j} are projected out. By default, K is set to the number of columns in the raw file, e.g., 64 for the majority of the reported results. This simple processing interferes minimally with SCANRAW thus allowing for exact measurements to be taken.

Parallelism. Figure 4 depicts the effect the number of workers in the thread pool has on the execution of speculative loading, query-driven loading and execution – load all data into the database only when queried – and external tables. Notice that all these three regimes are directly supported in SCANRAW with simple modifications to the scheduler writing policy. Zero worker threads correspond to sequential execution, i.e., the chunks go through the conversion stages one at a time. With one or more worker threads, READ and WRITE are separated from conversion—TOKENIZE and PARSE. Moreover, their execution is overlapped. While the general trend is standard – increasing the degree of parallelism results in better performance – there are multiple findings that require clarification. The execution time (Figure 4a) – and the speedup (Figure 4c) – level-off beyond 6 workers. The reason for this is that processing becomes I/O-bound. Increasing the number of worker threads does not improve performance anymore. As expected, loading all data during query processing increases the execution time. What is not expected though is that this is not the case when the number of worker threads is 1, 2, and 4. In these cases, full loading, speculative loading, and external tables have identical execution time. The reason for this behavior is that processing is CPU-bound and – due to parallelism – SCANRAW manages to overlap conver-



(a) (b)

Figure 5: Pipeline execution time: (a) absolute, (b) relative.

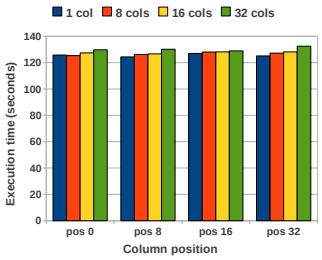


Figure 6: Position and number of columns.

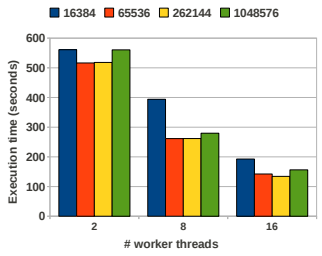


Figure 7: Chunk size.

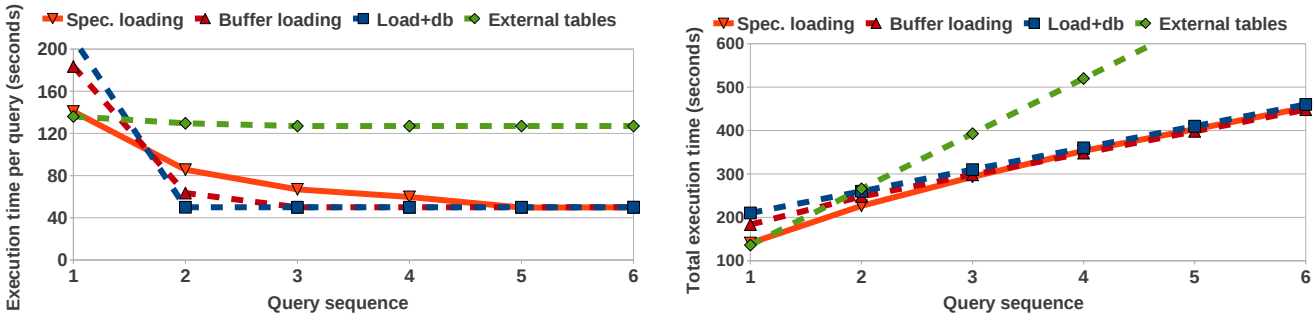
sion to binary and loading into the database completely. Essentially, loading comes for free since the disk is idle. All the unique SCANRAW features – super-scalar pipeline, asynchronous threads, dynamic scheduling – combine together to make loading and processing as efficient as external tables. We are not aware of any other raw file processing operator capable to achieve this performance. The effect of parallel processing on speculative loading is illustrated in Figure 4b. As long as the execution is CPU-bound, speculative loading operates as full loading, writing almost all the converted chunks into the database. This happens for a small number of worker threads, i.e., less than 6. As soon as there are enough workers (6 or more) to handle all data read from disk – the execution becomes I/O-bound – SCANRAW switches to external tables and does not load any chunks at all, i.e., there is no speculative loading. In Figure 4a, the curves for external tables and speculative loading are always overlapped for more than one thread. Independent of the number of workers, SCANRAW minimizes query execution time.

Pipeline analysis. Figure 5 depicts the duration of each stage in the SCANRAW pipeline for all the column sizes considered in the experimental dataset, i.e., 2 to 256 in powers of 2. The number of lines in all the files is 2^{26}. WRITE time is included in these measurements since the experiment is executed with full data loading. We report the average time per chunk in each stage over all the chunks in the file. The absolute time to process a chunk is shown in Figure 5a. As expected, when the number of columns increases, so does the chunk processing time. More exactly, the time doubles with the doubling in the number of columns. For more than 16 columns, PARSE is by far the most time-consuming stage. This is where database processing over binary data outperforms standard external tables. This is also the operation point targeted by SCANRAW with massive parallelism supported by the modern many- and multi-core processors. Essentially, SCANRAW transforms this CPU-bound task into typical database I/O-bound processing (Figure 4a) over raw files, thus making data loading obsolete. Figure 5b gives the relative distribution of the data in Figure 5a. The relative significance of I/O operations – READ and WRITE – drops from 45% for 2 columns to approximately 20% for 256 columns. PARSE doubles from 30% to 60%. This proves that PARSE is the stage to optimize in order to make raw file processing efficient. SCANRAW achieves this through parallelization. The other alternative is saving (caching) the binary conversion.

Position and number of columns. Figure 6 depicts the effect of two parameters on SCANRAW performance—the number of columns projected by the query and the starting position of the first column. In this experiment, we consider that only a continuous subset of the 64 columns are required in the query. The starting position of the subset – the first column – is also a parameter. The purpose is to measure the effect of selective tokenizing and parsing on SCANRAW performance. SCANRAW is configured with 8 worker threads. Increasing the number of columns required in the query results in a slight increase in the conversion time—less than 5%. This is expected since the number of function calls in PARSE increases. The position of the first column in the subset does not impact performance. The reason for this is that the minimal increase in tokenization time is completely hidden by parallel execution. These results confirm once more that PARSE is the stage to optimize in raw file processing.

Chunk size. The chunk size – number of lines in the file processed as a unit – has a dramatic impact on the pipeline efficiency—depicted in Figure 7. The chunk size has to be large enough to hide the overhead introduced by the dynamic allocation of tasks to worker threads. If too large, it takes longer to fill and free the pipeline since the amount of overlap is limited. While the actual chunk size is dependent on the data, we found that between 2^{17} and 2^{19} tuples per chunk are optimal for our datasets. The chunk size used throughout the experiments is $2^{19} \approx 500K$.

Speculative loading for a query sequence. Figure 8 depicts SCANRAW performance for a query sequence consisting of 6 standard queries, i.e., SELECT SUM($\sum_{i=1}^{64} C_i$) FROM $2^{26} \times 64$. Executing instances of the same query guarantees that the same data are accessed in every query. This allows us to detect and quantify the effect the data source has on query performance. The methods we compare are database loading, buffered loading, i.e., data are written to the database only when the binary cache buffer is full, external tables, and speculative loading. The size of the binary cache used in buffered and speculative loading, respectively, is 32 chunks. Since SCANRAW is configured with 16 worker threads, speculative loading behaves similar to external tables. This allows us to verify the effectiveness of the safeguard mechanism. Since the number of chunks loaded during query processing is non-deterministic, it is more difficult to observe. Figure 8a shows the execution time for every query in the sequence. As expected, this is (almost) constant for external tables. Data are always read from the raw file, tokenized, and parsed before being passed to the execution engine. The same is true for database execution starting from the second query—the first query incurs the entire loading time, thus it takes significantly longer. The difference is that database execution is considerably faster than external tables—a factor of 2.5. In SCANRAW, this is entirely due to the difference in size between text and binary format—40 GB and 16 GB, respectively. Buffered loading distributes the loading time over the first two queries since not all data fit in memory. Every chunk expelled from the cache is automatically written to the database. As a result, there is a decrease in



(a) (b)

Figure 8: Execution time for a sequence of queries. (a) Execution time for query i. (b) Overall execution time up to query i.

runtime for the first query when compared to standard loading. For the second query though, execution time is larger. Speculative loading exhibits a considerably more interesting behavior. It has exactly the same execution time as external tables for the first query – this is the absolute minimum that can be achieved – and then converges to the database execution time after a number of queries. According to Figure 8a, it takes only 5 queries. This is expected since the size of the cache is $1/4^{\text{th}}$ of the number of chunks accessed by the query. Even though speculative loading operates in external tables mode, it manages to load additional chunks – 2-4 chunks, to be precise – into the database in the interval between reading finishes and query execution completes. This is possible only because of the asynchronous multi-thread SCANRAW architecture.

Figure 8b shows the overall execution time after i queries in the sequence, where i goes from 1 to 6. At least two important observations can be drawn. First, after only two queries data loading already pays off since the database performance is equal to external tables. This proves that SCANRAW loading is optimal. Second, speculative loading is always more efficient than database processing. This is somehow unexpected since database processing is supposed to be optimal when a large enough number of queries are executed. The reason this is happening is because even though speculative loading goes multiple times to the raw file, it only reads data not cached or loaded. The difference becomes even larger when the text file has a size comparable to the binary format. Moreover, speculative loading achieves optimal performance at any point in the query sequence—including the first query. This is not true for buffered loading even though not all data are loaded into the database. It is important to notice that speculative loading has similar behavior as buffered loading when all data fit in memory. The only difference is that speculative loading materializes cached data into the database proactively, when resources are available.

Resource utilization. In Figure 9, we display the SCANRAW CPU and I/O utilization for processing a 256 column raw file with speculative loading. In this situation, the execution is CPU-bound even for 8 worker threads—the reason why CPU utilization goes to 800. The interesting aspect to observe here is how the SCANRAW scheduler alternates between READ and WRITE in order to utilize resources optimally. Whenever the CPU is fully-utilized and no reading is executed, WRITE is triggered to load data into the database. This results in a temporary decrease in disk utilization since writing is done one chunk at a time. As soon as worker threads become available, the scheduler resumes reading and disk utilization goes back to 100% since a sequence of chunks are typically read at a time.

5.2 Real Data

Method	Execution time (sec)
External tables (SAM)	370
External tables (BAM + BAMTools)	2714
Data loading (SAM)	945
Database processing	122
Speculative loading (SAM)	370

Table 1: SCANRAW performance on SAM/BAM data.

In order to evaluate SCANRAW on a real dataset, we use genomic sequence alignment data from the *1000 Genomes* project [2]. These data come in two formats—SAM is text while BAM is compressed binary. We use the file corresponding to individual NA12878 containing more than 400 million reads. SAM is 145 GB in size while BAM is 26 GB. As for processing, we compute the distribution of the CIGAR field at positions in the genome where reads exhibit a certain pattern. The SQL equivalent is a group-by aggregate query with a pattern matching predicate. Table 1 shows the results we obtain for different SCANRAW configurations. In all the queries over SAM files, we use a SCANRAW implementation for processing tab-delimited text files. The tokenizing and parsing are handled inside SCANRAW. For BAM file processing, we use BAMTools [7] to extract the tuples from binary and implement only MAP in SCANRAW. There is a single operation executed in MAP—convert the BAMTools internal representation to SCANRAW. While the results are standard – database processing is fastest, followed by external tables, and data loading – the comparison between SAM and BAM processing is surprising. SCANRAW takes more than 7 times less to process a file more than 5 times larger. After careful investigation, we found the problem to be BAMTools. The SAM implementation in SCANRAW parallelizes tokenizing and parsing such that processing becomes I/O-bound. For BAM, file data access and decompression are sequential and handled inside BAMTools. The process is heavily CPU-bound. While we did not modify the BAMTools code, we parallelized MAP—without any performance gains.

5.3 Discussion

The experimental results confirm the benefits of the SCANRAW super-scalar pipeline architecture for in-situ data processing. Parallel execution at chunk granularity results in linear speedup for CPU-bound tasks. SCANRAW with speculative loading achieves

optimal performance across a sequence of queries at any point in the execution. It is similar to external tables for the first query and more efficient than database processing in the long run. Moreover, SCANRAW makes full data loading efficient to the point where database processing – with pre-loading – achieves better overall execution time than external tables even for a two-query sequence. While the time distribution is split almost equally between I/O and CPU-intensive pipeline stages when the number of columns in the file is small, CPU-intensive stages – TOKENIZE and PARSE – account for more than 80% of the time to process a chunk when the raw file contains a large number of numeric attributes. By overlapping processing across multiple chunks and between stages, SCANRAW makes even this type of execution I/O-bound. This guarantees optimal resource utilization in the system. Due to parallel conversion from text to binary, SCANRAW outperforms BAMTools by a factor of 7 while processing a file 5 times larger.

6. RELATED WORK

Several researchers [13, 3, 23, 16, 15] have recently identified the need to reduce the analysis time for processing tasks operating over massive repositories of raw data. In-situ processing [14] has been confirmed as one promising approach. At a high level, we can group in-situ data processing into two categories. In the first category, we have extensions to traditional database systems that allow raw file processing inside the execution engine. Examples include external tables [25, 18, 17] and various optimizations that eliminate the requirement for scanning the entire file to answer the query [10, 5, 11]. The second category is organized around the MapReduce programming paradigm [12] and its implementation in Hadoop. While some of the data extraction is implemented by adapters that convert data from various representations into the Hadoop internal format [19], the application is still responsible for a significant part of the conversion, i.e., the Map and Reduce functions contain large amounts of tokenizing and parsing code. The work in this category focuses on eliminating the conversion code by caching the already converted data in memory or storing it in binary format inside a database [4].

External tables. Modern database engines, e.g., Oracle [25] and MySQL [17], provide external tables as a feature to directly query flat files using SQL without paying the upfront cost of loading the data into the system. External tables work by linking a database table with a specified schema to a flat file. Whenever a tuple is required during query processing, it is read from the flat file, parsed into the internal database representation, and passed to the execution engine. Our work can be viewed as a parallel pipelined implementation of external tables that takes advantage of the current multi-core processors for improving performance significantly when mapping data into the processing representation is expensive. As far as we know, SCANRAW is the first parallel pipelined solution for external tables in the literature. Moreover, SCANRAW goes well beyond the external tables functionality and supports speculative loading, tokenizing, and parsing.

Adaptive partial loading [10]. The main idea in adaptive partial loading is to avoid the upfront cost of loading the entire data into the database. Instead, data are loaded only at query time and only the attributes required by the query, i.e., push-down projection. An additional optimization aimed at further reducing the amount of loaded data is to push the selection predicates into the loading operator, i.e., push-down selection, such that only the tuples participating in the other query operators are loaded. The proposed adaptive loading operator is invoked whenever columns or tuples required in the current query are not stored yet in the database. It is important to notice that the operator executes a partial data loading before query execution can proceed. While SCANRAW supports adaptive partial loading, it avoids loading all the data into the database the first time they are accessed. Query processing has higher priority. Data are loaded only if sufficient I/O bandwidth is available.

NoDB [5]. NoDB never loads data into the database. It always reads data from the raw file thus incurring the inherent overhead associated with tokenizing and parsing. Efficiency is achieved by a series of techniques that address these sources of overhead. Caching is used extensively to store data converted in the database representation in memory. If all data fit in memory NoDB operates as an in-memory database, without accessing the disk. Whenever data have to be read from the raw file, tokenizing and parsing have to be executed. This is done adaptively though. A positional map with the starting position of all the attributes in all the tuples completely eliminates tokenizing. The positional map is built incrementally, from the executed queries. Moreover, only the attributes required in the current query are parsed. When the positional map, selective parsing, and caching are put together, NoDB achieves performance comparable – if not better – to executing queries over data stored in the database. This happens though only when NoDB operates over cached data, as an in-memory database. The main difference between SCANRAW and NoDB is that SCANRAW still loads data into the databases—without paying any cost for it though. Additionally, SCANRAW implements a parallel pipeline for data conversion. This is not the case in NoDB which is implemented as a PostgreSQL extension.

Data vaults [11]. Data vaults apply the same idea of query-driven just-in-time caching of raw data in memory. They are used in conjunction with scientific repositories though and the cache stores multi-dimensional arrays extracted from various scientific file formats. Similar to NoDB, the cached arrays are never written to the database. The ability to execute queries over relations stored in the database, cached arrays, and scientific file repositories using SciQL as a common query language is the main contribution brought by data vaults.

Invisible loading [4]. Invisible loading extends adaptive partial loading and caching to MapReduce applications which operate natively over raw files stored in a distributed file system. The database is used as a disk-based cache that stores the content of the raw file in binary format. This eliminates the inherent cost of tokenizing and parsing data for every query. Notice though that processing is still executed by the MapReduce framework, not the database. Thus, the database acts only as a more efficient storage layer. In invisible loading, data converted into the MapReduce internal representation are first stored in the database and only then are passed for processing. While this is similar to adaptive partial loading [10], an additional optimization is aimed at reducing the storing time. Instead of saving all the data into the database, only a pre-determined fraction of a file is stored for every query. The intuition is to spread the cost of loading across multiple queries and to make sure that loaded data are indeed used by more than a single query. The result is a smooth decrease in query time instead of a steep drop after the first query—responsible for loading all the required data. The proposed speculative loading implemented in SCANRAW brings two novel contributions with respect to invisible loading. First, the amount of data loaded for every query changes dynamically based on the available system resources. Speculative loading degenerates to invisible loading only in the case when no I/O bandwidth is available. And second, speculative loading overlaps entirely with query processing without having any negative effects on query performance. This is the result of the SCANRAW pipelined architecture.

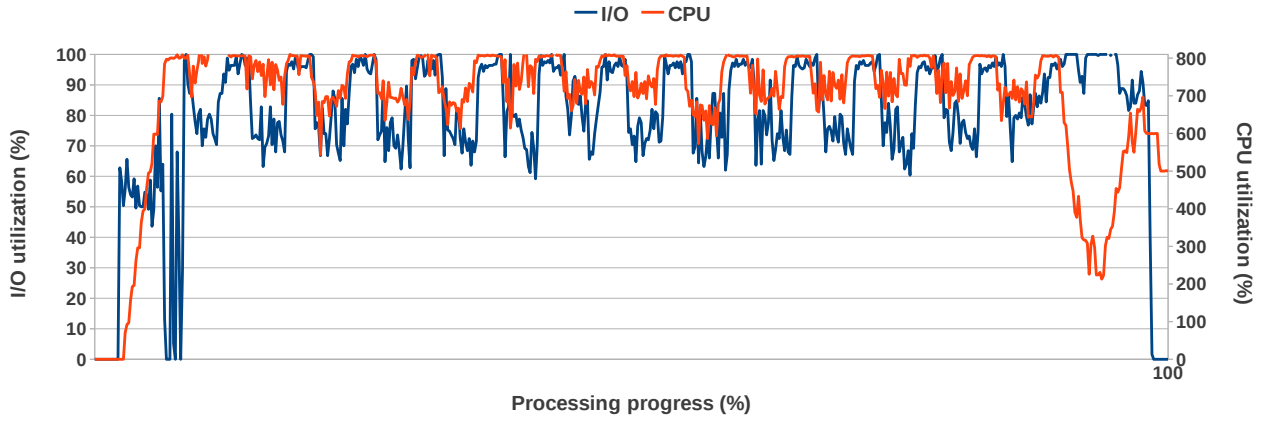


Figure 9: Resource utilization in SCANRAW.

Instant loading [24]. Instant loading proposes scalable bulk loading methods that take full advantage of the modern super-scalar multi-core CPUs. Specifically, vectorized implementations using SSE 4.2 SIMD instructions are proposed for tokenizing and parsing. These stages are still executed sequentially though for every data partition—there is no pipeline parallelism. Moreover, instant loading does not support raw file query processing. SCANRAW on the other hand overlaps the execution of tokenizing and parsing both across data partitions and for each partition individually. And with the actual query processing and/or loading. Overall, instant loading introduces faster algorithms for tokenizing and parsing that can be immediately integrated in SCANRAW. We plan to do so in the future and investigate the performance of SCANRAW with tokenizing and parsing implemented using the vectorized techniques proposed in [24].

7. CONCLUSIONS AND FUTURE WORK

In this paper, we propose SCANRAW—a novel database physical operator for in-situ processing over raw files that integrates data loading and external tables seamlessly while preserving their advantages. SCANRAW supports single-query optimal execution with a *parallel super-scalar pipeline implementation* that overlaps data reading, conversion into the database representation, and query processing. SCANRAW implements *speculative loading* as a gradual loading mechanism to store converted data inside the database. We implement SCANRAW in a state-of-the-art database system and evaluate its performance across a variety of synthetic and real-world datasets. Our results show that SCANRAW with speculative loading achieves optimal performance for a query sequence at any point in the processing. In future work, we plan to focus on extending SCANRAW with support for multi-query processing over raw files.

8. REFERENCES

[1] New NIH-Funded Resource Focuses on Use of Genomic Variants in Medical Care, 2013. http://www.nih.gov/news/health/sep2013/nhgri-25.htm.

[2] 1000 Genomes. http://www.1000genomes.org/data.

[3] A. Ailamaki, V. Kantere, and D. Dash. Managing Scientific Data. *Commun. ACM*, 53, 2010.

[4] A. Abouzied, D. Abadi, and A. Silberschatz. Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems. In *EDBT/ICDT 2013*.

[5] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD 2012*.

[6] S. Arumugam, A. Dobra, C. Jermaine, N. Pansare, and L. Perez. The DataPath System: A Data-Centric Analytic Processing Engine for Large Data Warehouses. In *SIGMOD 2010*.

[7] BAMTools. http://sourceforge.net/bamtools/.

[8] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of Database Processing or a Passing Fad? *SIGMOD Rec.*, 19, 1991.

[9] H. Li et al. The Sequence Alignment/Map Format and SAMtools. *Bioinformatics*, 25(16), 2009.

[10] S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. Here Are My Data Files. Here Are My Queries. Where Are My Results? In *CIDR 2011*.

[11] M. Ivanova, M. L. Kersten, and S. Manegold. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *SSDBM 2012*.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1), 2008.

[13] J. Gray et al. Scientific Data Management in the Coming Decade. *SIGMOD Rec.*, 34, 2005.

[14] K. Lorincz et al. Grep versus FlatSQL versus MySQL: Queries using UNIX tools vs. a DBMS, 2003. Harvard University.

[15] M. Kersten, S. Idreos, S. Manegold, and E. Liarou. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *PVLDB*, 4, 2011.

[16] M. Stonebraker et al. Requirements for Science Data Bases and SciDB. In *CIDR 2009*.

[17] MySQL CSV Storage Engine. http://dev.mysql.com/doc/refman/5.0/en/csv-storage-engine.html.

[18] N. Alur et al. *IBM DataStage Data Flow and Job Design*. 2008.

[19] Optiq. https://github.com/julianhyde/optiq.

[20] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *SIGMOD 2000*.

[21] S. Idreos et al. MonetDB: Two Decades of Research in Column-Oriented Database Architectures. *IEEE Data Eng. Bull.*, 35(1), 2012.

[22] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB*, 4, 2011.

[23] T. Hey, S. Tansley, and K. Tolle. The Fourth Paradigm: Data-Intensive Scientific Discovery, 2009. Microsoft Research.

[24] T. Muhlbauer et al. Instant Loading for Main Memory Databases. *PVLDB*, 6(14), 2013.

[25] A. Witkowski, M. Colgan, A. Brumm, T. Cruanes, and H. Baer. Performant and Scalable Data Loading with Oracle Database 11g, 2011.