SCANRAW: A Database Meta-Operator for Parallel In-Situ Processing and Loading

YU CHENG, University of California Merced
FLORIN RUSU, University of California Merced

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 meta-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. We decompose loading and external table processing into atomic stages in order to identify common functionality. We analyze alternative implementations and discuss possible optimizations for each stage. 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*

General Terms: Design, System, Database operator, Performance

Additional Key Words and Phrases: data access operator; external table; data loading; access path

ACM Reference Format:
Yu Cheng and Florin Rusu, 2015. SCANRAW: A Database Meta-Operator for Parallel In-Situ Processing and Loading. *ACM Trans. Datab. Syst.* 0, 0, Article 0 (0), 42 pages.
DOI : http://dx.doi.org/10.1145/0000000.0000000

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 [Idreos et al. 2011; Alagiannis et al. 2012]. 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

This work is supported by a U.S. Department of Energy Early Career Award (DOE Career).
Authors' address: School of Engineering, University of California Merced, 5200 N Lake Rd., Merced, CA 95343, USA. Email: ycheng4@ucmerced.edu; frusu@ucmerced.edu.

DOI : http://dx.doi.org/10.1145/0000000.0000000

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 data representation independence, though—achieved through loading and storing data in a proprietary format.

External tables [Witkowski et al. 2011] 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[1] – SAM [Li et al. 2009] are text, BAM [Barnett et al. 2011] 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[3], i.e., genome mutation responsible for causing hereditary diseases, identification. It requires computing the distribution of the `CIGAR` field 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 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[4] and BAMTools[5], 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. 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 a large enough number of 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.

[1] http://samtools.sourceforge.net/SAMv1.pdf
[2] http://www.1000genomes.org/data
[3] http://www.nih.gov/news/health/sep2013/nhgri-25.htm
[4] http://samtools.sourceforge.net/
[5] http://sourceforge.net/projects/bamtools/

Contributions. The major contribution we propose in this paper is SCANRAW—a novel database meta-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 meta-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.
— Investigate how all the available forms of parallelism supported by modern CPUs can be applied to raw file data processing. Integrate data partitioning, task parallelism and pipelining, and vectorized instructions in SCANRAW, and assess their performance benefits for raw file data processing.
— Design an adaptive scheduling strategy for dynamically assigning worker threads to raw file extraction tasks. The goal of adaptive scheduling is to optimize resource utilization in the system and minimize query execution time, while maximizing the amount of data loaded into the database.
— Design the merge read mechanism for reading data from multiple sources optimally. Merge read groups multiple requests corresponding to the same data source and schedules them together.
— Design speculative loading as a gradual data loading mechanism that dynamically and adaptively takes advantage of the disk idle intervals arising during data conversion and query processing.
— Design the multi-step loading mechanism for storing raw data into the database, without immediate conversion to the internal format. Since data are converted into binary lazily, a significant improvement is achieved in CPU-bound tasks by eliminating parsing of unnecessary columns.
— Implement several instances of the SCANRAW meta-operator, e.g., SCANRAW-CSV, SCANRAW-SAM, SCANRAW-BAM, SCANRAW-FITS, in a state-of-the-art multi-threaded database system [Arumugam et al. 2010; Cheng et al. 2012; Cheng and Rusu 2014a] and evaluate its performance across a variety of synthetic and real-world datasets and data formats. Compare SCANRAW against raw data processing operators in MySQL[6] and Impala [M. Kornacker et al. 2015]—two other data processing systems with support for external tables. Our results show that SCANRAW with speculative loading achieves optimal performance for a query sequence at any point in the processing and outperforms considerably the other systems.

Roadmap. In Section 2, we provide a formal characterization for in-situ data processing over raw files. The forms of parallelism supported by modern computer architectures and how they can be applied to raw file in-situ processing are discussed in Section 3. The SCANRAW architecture and operation – including the thread scheduling algorithms and the merge read mechanism – are introduced in Section 4, while speculative loading is presented in Section 5. The multi-step loading mechanism is presented in Section 6. Detailed experimental results targeting all the aspects of the SCANRAW operator are presented in Section 7. We conclude with a detailed look at related work (Section 8) and plans for future work (Section 9).

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,

[6]http://dev.mysql.com/doc/refman/5.7/en/csv-storage-engine.html

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.

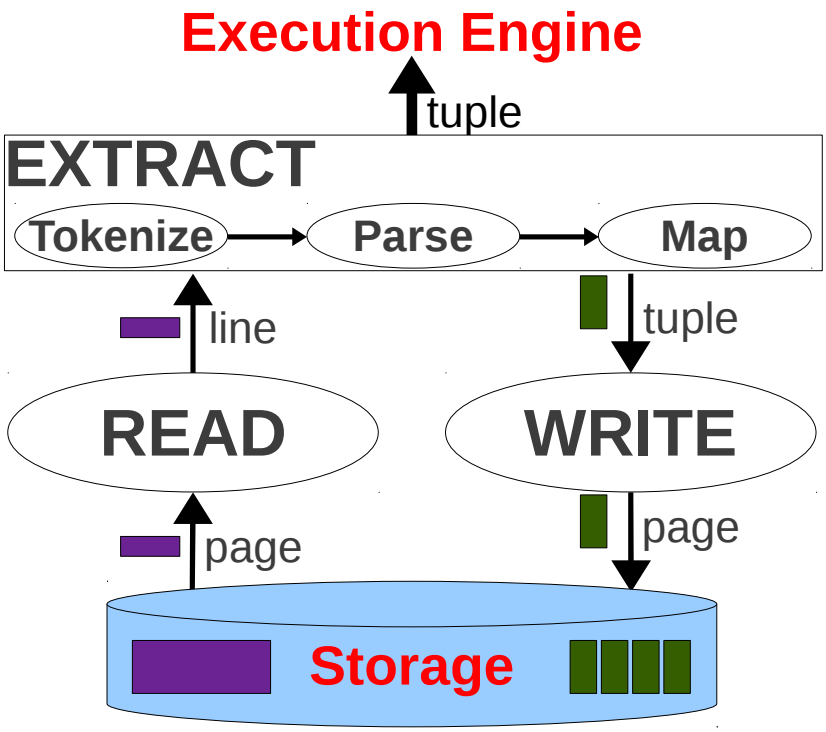


Fig. 1: Query processing over raw files.

2.1. 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 [Barnett et al. 2011] are indexes built on top of BAM files. Columnar storage [Abadi et al. 2013] and compression [Raman et al. 2008] are other strategies to minimize the amount of data read from disk—orthogonal to our discussion.

2.2. 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 [Idreos et al. 2011]. 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 [Alagiannis et al. 2012], 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.

2.3. 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 [Idreos et al. 2011] 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 [Alagiannis et al. 2012] 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.

2.4. 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 [Alagiannis et al. 2012] 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.

2.5. 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 complex 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 extremely negative though. The situation gets less complicated in column-oriented databases, e.g., MonetDB [Idreos et al. 2012], 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. PARALLEL RAW FILE QUERY PROCESSING

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently. In this section, we present how to utilize parallelism to speed-up in-situ data processing. We recognize three general types of parallelization [DeWitt and Gray 1991]: data parallelism, task parallelism, and pipelining. In the following sections, we introduce these parallelization methods and show how they apply to in-situ data processing and loading.

3.1. Data Parallelism

Data parallelism is a form of parallelization across multiple processors or cores in parallel computing environments. Data parallelism focuses on distributing the data across different processors or cores. It emphasizes the distributed nature of the data, as opposed to processing.

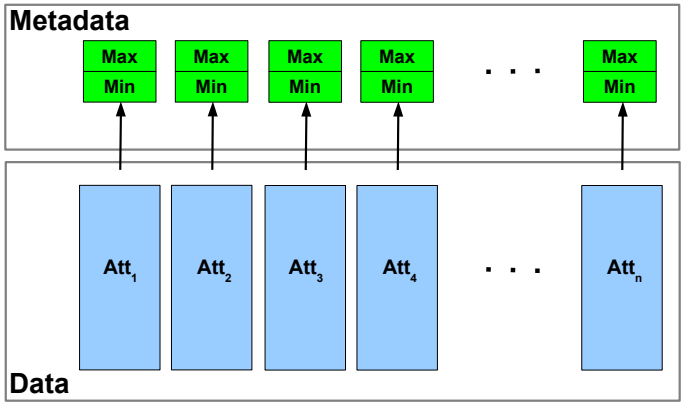


Fig. 2: Chunk structure for internal processing.

Data partitioning. Horizontal data partitioning or chunking is one strategy of data parallelism. When executing a query, the partitions are independently assigned to different execution entities for processing. Since each processing entity works on a considerably smaller dataset, a speed-up proportional to the number of processing workers can be obtained in optimal conditions. Data partitioning can be applied both to the raw files as well as to the internal processing representation. We apply the simple data partitioning strategy described in [DeWitt and Gray 1991] by breaking the raw file into multiple segments of fixed size—in the order of tens to hundreds of megabytes. This has the potential to increase the length of sequential scans and reduce the number of disk seeks. The segment, i.e., chunk, is both the read/write and processing unit.

Figure 2 depicts the generic structure of an internal chunk containing metadata to support range-based data partitioning. The metadata contains the minimum and maximum values for each attribute and are stored in the system catalog. They represent a primitive form of indexing. Besides, we further apply column-based storage inside chunks. This type of storage structure vertically partitions columns inside chunks, associated with an array of pointers to all the columns. The actual data are vertically partitioned, with each column stored in a separate set of disk blocks. This design can improve the performance for accessing selective attributes and allows only for the required columns to be read for each query, thus minimizing the I/O bandwidth. However, the impact of the (Min, Max) ranges on attributes is not always significant since there is no guarantee that attribute values are clustered.

Vectorization. Modern CPUs are highly parallel processors with different levels of parallelism, from the parallel execution units in a CPU core, up to the SIMD (Single Instruction Multiple Data) instruction set, and the parallel execution of multiple threads across cores. Vectorization is the representative instance of SIMD data parallelism. Vectorized instructions operate on multiple data elements in one instruction and make use of wide registers to store both the operands and the result. The Intel `SSE` instruction set[7], which is an extension to the `x86` architecture, is the standard for vectorized processing. `SSE 4.2` includes byte-comparison instructions for string and text processing, which can be used to accelerate string operations.

How is vectorization supported by modern compilers? It is the unrolling of a loop combined with the generation of packed SIMD instructions by the compiler. Because the packed instructions operate on more than one data element at a time, the loop can be executed more efficiently. Modern compilers, such as `GCC`[8] and `LLVM`[9], detect vectorization opportunities automatically whenever

[7]https://software.intel.com/en-us/articles/extending-the-worlds-most-popular-processor-architecture
[8]https://gcc.gnu.org/
[9]http://www.llvm.org/

default optimization (`-O2` or higher) is enabled. However, the compiler is not always capable of taking advantage of the SIMD instructions without the programmer having to explicitly re-write the code following specific criteria.

ALGORITHM 1: Find Next Delimiter

Input: source string $input$; delimiters SC
Output: index of next delimiter in $input$
$searchIdx$ = 0;
$mode = \_SIDD\_CMP\_EQUAL\_ANY$;
$\_\_m128i\ data$;
$\_\_m128i\ pattern$ = $\_mm\_set\_epi8(SC)$;
while *!IsEnd(input)* **do**
  $data$ = $\_mm\_loadu\_si128(input)$;
  $searchIdx$ = $\_mm\_cmpistri(pattern, data, mode)$;
  if $(searchIdx < 16)$ **then**
    //processing delimiter
  end
  $input$ = $input$ + 16;
end

Vectorization can be used to speed-up the tokenization process, as shown in [Mühlbauer et al. 2013]. The `SSE 4.2` instruction set works on 128-bit registers and contains instructions for the comparison of two 16 bytes operands of explicit or implicit lengths. Instead of finding the delimiter character by character, the $\_\_mm\_cmpistrm$ intrinsic can be applied to check 16 bytes at a time. Algorithm 1 illustrates the method. $input$ contains the string that needs to be handled. A 128-bit register, denoted as SC, is used to store the delimiters. At each iteration, 16 bytes of data from $input$ are loaded into another 128-bit register and are checked whether any is equal to a delimiter in SC. The return value of the $\_\_mm\_cmpistrm$ intrinsic indicates the result. If a delimiter is found, the return value equals its index position. Otherwise, the return value is 0.

3.2. Task Parallelism

Task parallelism[10] – also known as function parallelism or control parallelism – is a form of parallelization of computer code across multiple processors in parallel computing environments. Task parallelism focuses on distributing execution processes – or threads – across different parallel computing nodes. Task parallelism can be applied to raw file query processing by assigning the stages identified in Figure 1 – `READ`, `TOKENIZE`, `PARSE`, `MAP`, and `WRITE` – to separate processes or threads. In modern multi-core CPUs, different stages – and multiple instances of the same stage – can be executed concurrently. Moreover, query processing can be viewed as another task that can be also included in the parallel task assignment process.

3.3. Pipelining

Pipeline parallelism is a special form of task parallelism where a problem is divided into sub-problems, which can each be operated on independently, and where there are multiple problem instances to be solved at a given instant in time. Compared to data parallelism, this approach causes shorter latency, less buffering, and good locality. The potential benefits of pipeline parallelism are easy to quantify. Assuming an application is divided into n stages, let t_i denote the processing time for each stage. Then, the execution time and throughput for a non-pipelined program are $T_{\textit{no-pipeline}} = \sum_{i=1}^{n} t_i$ and $1/T_{\textit{no-pipeline}}$, respectively. When pipeline parallelism is active, suppose $t_m = \max\{t_1, t_2, \ldots, t_n\}$ represents the execution time of the slowest stage in the pipeline. Then,

[10] http://en.wikipedia.org/wiki/Task_parallelism

the pipeline throughput is $1/t_m$, since a result is produced at every $T_{pipeline} = t_m$ time instances. When executing a set of L tasks, the speedup rate is given by:

$$\eta = \frac{L \cdot T_{no\text{-}pipeline}}{T_{no\text{-}pipeline} + (L-1) \cdot t_m} \quad (1)$$

When the number of tasks L is extremely large, i.e., $L \to \infty$, the speedup approaches $T_{no\text{-}pipeline}/t_m$, which means the pipeline throughput is decided entirely by the slowest stage.

As shown in Figure 1, raw file query processing has been split into multiple stages which are of high cohesion and low coupling. Pipeline parallelism can be exploited by mapping clusters of producers and consumers to different stages, connected through buffers. Buffering allows storing results of a stage temporarily before forwarding them to the subsequent stages. It is essential in smoothing out the flow of a computational process when the timing for each stage is variable.

4. 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 meta-operator implementing query processing over raw data based on the decomposition presented in Section 2 and implementing the parallelization techniques discussed in Section 3. Our major contribution is a *parallel super-scalar pipeline architecture* [Patterson et al. 1996] 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 partitioning parallelism [DeWitt and Gray 1991]—also implemented in SCANRAW. Some solutions follow the principle READ-EXTRACT-PROCESS, e.g, external tables and NoDB [Idreos et al. 2011; Alagiannis et al. 2012], while others [Abouzied et al. 2013] 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.

4.1. Architecture

The super-scalar pipeline architecture of SCANRAW is depicted in Figure 3. 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 [DeWitt and Gray 1991]. 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 chunk, SCANRAW converts into a parallel Extract-Transform-Load (ETL) operator. While both these scheduling policies are supported in SCANRAW, we propose a completely different WRITE

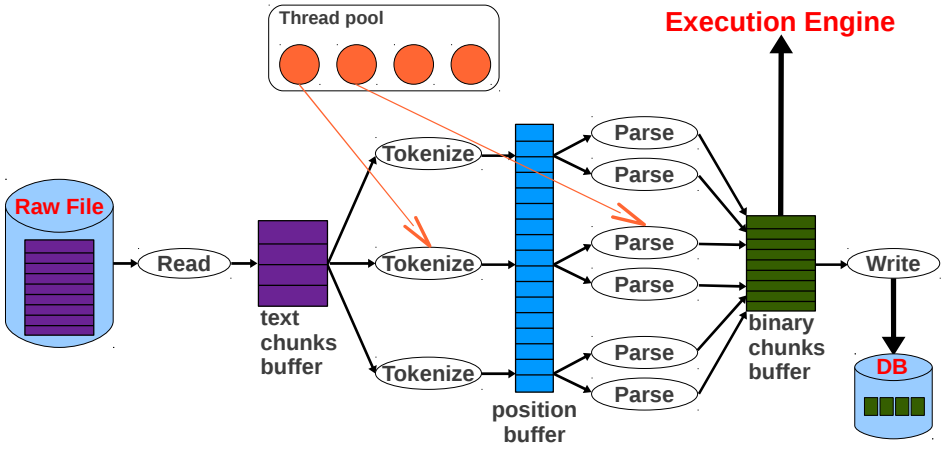


Fig. 3: SCANRAW architecture.

behavior—*speculative loading* (Section 5). 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.

One potential problem with the super-scalar pipeline architecture is that chunks can be passed to the execution engine in a different order than the raw file. This is possible because of the multiple parallel paths a chunk can take. While not a problem in the relational data model, this can be an issue if strict ordering is required. `SCANRAW` can handle this scenario using a similar approach to CPUs—reordering at the binary chunks buffer. Chunks read from the raw file are stamped with a sequential identifier when they are inserted into the text chunks buffer and reordered based on it once they hit the binary chunks buffer. They are subsequently passed to the execution engine in the order they appear in the file.

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 [Avnur and Hellerstein 2000] 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 posi-

tional 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.

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 that have already been written to the database 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 starts to pre-fetch chunks as soon as the query is compiled and caches them in the binary chunks buffer. The goal is to guarantee that the execution engine is fed continuously with data and the delay introduced by the I/O is minimized. Pre-fetching 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.

Metadata. SCANRAW extracts valuable metadata while converting raw chunks into binary. These metadata are stored in the catalog to be used for processing subsequent queries. They also represent an important source in query optimization. The extracted metadata include the position in the raw file where each chunk begins and for every attribute the minimum and maximum value in the chunk. While the starting position provides direct access to a chunk, the minimum/maximum values allow us to identify the chunks required by a given query by evaluating the selection predicates before reading the data. In the best case when data in a column are range-partitioned across chunks this results in significant I/O and CPU savings—no tokenizing and parsing. The positional map computed in TOKENIZE can also be considered metadata. Its size is considerably larger though and its usage is limited to avoiding tokenizing—reading and parsing are still required. The metadata also contains the information about load status. When a chunk has been loaded into the database, the metadata not only record the general information of chunk ID, but also remembers the critical contend of

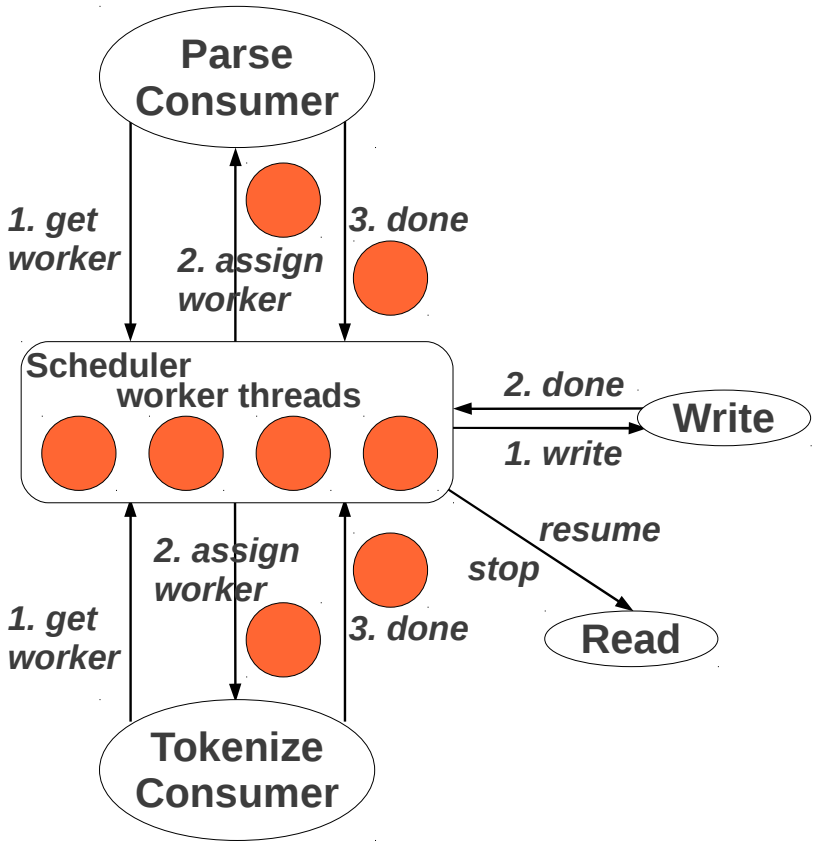


Fig. 4: SCANRAW threads and control messages.

corresponding columns. Therefore, when a query requires a list of attributes, the metadata manager could indicate how many chunks are loaded or not. Besides, for a loaded chunk, it could distinguish the columns already in database with other columns that resides in raw files.

4.2. 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 3. The stand-alone threads – depicted in Figure 4 by ovals – communicate through control messages (arrows) while data are passed through the architectural buffers. The communication patterns involve exactly two threads and they consist of at most three steps. The order is given by the number on the arrow. Notice that these threads – `READ`, `WRITE`, `TOKENIZE`, `PARSE`, and `SCHEDULER` – are separate from the thread pool which contains worker threads – circles in Figure 4 – that are configured dynamically with the task to execute.

4.2.1. Stand-Alone Threads. 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.

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`. These are the only two control messages corresponding to `READ`. 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.

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. The SCHEDULER identifies when writing can occur by monitoring the text chunks buffer and triggers the action by sending the write control message. WRITE extracts a chunk from the binary chunks buffer – the default is the LRU algorithm for chunk replacement – and stores it inside the database. When writing finishes, the control message done is sent back to the SCHEDULER.

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 (control message get worker). Multiple such requests can be pending at the same time. Once a worker thread is allocated (control message assign worker), 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 (control message done) 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 stall. 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 to develop heuristics that guarantee correctness, while providing certain optimality conditions. In Section 4.3, we discuss in detail the strategies used to schedule worker threads in SCANRAW.

4.2.2. 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 3, two types of tasks can be assigned to worker threads—TOKENIZE and PARSE. They correspond to the stages of the pipeline architecture. 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.

4.3. Worker Thread Scheduling

SCANRAW can be viewed as a pipeline parallelism structure. The extraction process is expressed as a set of explicitly divided, concurrent, and independent stages, i.e., READ, TOKENIZE, PARSE, and WRITE, with producer-consumer communication between stages through data queues. Each of these stages has a unique task queue, to which it enqueues newly produced tasks and from which it dequeues tasks to be executed. For instance, the text chunks buffer and position buffer are two respective task queues for the TOKENIZE and PARSE stage, respectively. The pipeline structure has several advantages. Parallelism can be exploited at multiple levels, which allows the programmer to tolerate different dependence patterns. Communication is deterministic, following a producer-consumer pattern between the stages.

If the execution is I/O-bound, applications parallelized using the pipeline structure can be processed optimally – assuming the overhead introduced by moving data between pipeline stages is negligible – since the execution time is determined entirely by the read/write components which cannot be improved by any scheduling algorithm. However, when the execution turns out to be CPU-bound, an efficient scheduling strategy is critical to effective application execution since such applications are sensitive to load balancing. For optimal efficiency, pipelines must avoid "bubbles", i.e., all the stages must process data at all times. Load imbalance is usually caused by workload variation across stages. When the number of threads for every stage is the same, the stage with the largest amount of workload becomes the bottleneck. To address load imbalance, two orthogonal approaches are possible: 1) collapse all the parallel stages into one; 2) use dynamic scheduling to share the load among different stages. Collapsing pipeline stages is applicable only when all the intermediate stages are parallel. Dynamic scheduling is a more general solution. An optimal scheduler satisfies three desirable requirements. First, it keeps the execution units well utilized, performing load balancing if and when needed. Second, it guarantees bounds on resource utilization. In particular, bounding the memory footprint is especially important to avoid out-of-memory conditions. Third, the scheduling overhead is minimal.

According to these requirements, we design and implement two scheduling strategies in SCANRAW—best-effort and adaptive, respectively. They are both based on the dynamic pipeline structure. While these strategies are not novel from a scheduling perspective [Blumofe and Leiserson 1999; Sanchez et al. 2011], their application to a database operator for raw file processing is new. In order to clearly present the scheduling strategies, we formalize the SCANRAW resources as follows. There are N worker threads and the available memory is M. The operator has k stages. The workload, processing time, and memory consumption are denoted as w_i, t_i, and m_i, respectively. At any time, the number of worker threads assigned to different stages is expressed as n_i, where $1 \leq i \leq k$. Then, the expected processing time for a stage i is $\bar{t}_i = t_i/n_i$. In the following, we present the two scheduling policies and their implementation. We provide experimental results to compare their performance in Section 7.

Best-effort scheduling. In best-effort scheduling, processing is expressed as a graph of stages which communicate explicitly via data streams. The scheduler assigns a worker thread to a task based on the first-come first-served (FCFS) mechanism. The main idea of this strategy is that all the tasks are viewed equal to each other, so when two different tasks are ready and ask for resources, the scheduler simply chooses one of them randomly, without any calculation to make the decision. Best-effort can be viewed as a stateless mechanism, since it does not need any data or status

information to make the assignment. The scheduling overhead is zero and the scheduler implementation is straightforward. Therefore, best-effort is typically used as a preliminary heuristics choice. Best-effort scheduling works well in most situations and has the potential to maximize resource utilization. It guarantees that all the available threads are working in parallel as much as possible, particularly for computation-intensive tasks, since the system assigns threads without delay. However, it also has weaknesses. Since the scheduler never considers the system status at runtime, it can introduce load imbalance in general-purpose multi-core CPUs, where the workload can vary significantly at runtime, especially when the data footprint is constrained.

As an example, take a query A which has to process C chunks in total. During query execution, `SCANRAW` has to allocate memory in `READ` and `TOKENIZE` to store the raw data from the file and produce the positional map. Although `PARSE` requires memory for the binary chunk, it releases the memory allocated in the former stages, which is considerably larger. Therefore, the memory consumption of `READ` and `TOKENIZE` is positive, while it is negative for `PARSE`. If the scheduler assigns worker threads to `TOKENIZE`, the memory consumption increases. Opposite, if `PARSE` receives worker threads, there is memory that can be released. Too many `TOKENIZE` being processed concurrently can cause the memory usage to go out of the bounds. If most of the threads are in `PARSE` stages, the throughput of this stage can become larger than that of the query execution engine, which introduces stalls between `PARSE` and the execution engine.

Adaptive scheduling. Unlike the stateless best-effort scheduling, the adaptive strategy assigns worker threads according to the runtime system state, which includes the available resource status and statistics on current and past workloads. The resource status includes the number of available worker threads and the used memory capacity. The running time – the main statistic used by the scheduler – is recorded for each chunk in each stage. A timer is started when a worker thread is assigned to the stage and stopped when the thread is reclaimed. Initial values for the parameters are extracted from the historical workload. Once the execution starts, the parameter values are measured for every chunk and updated accordingly.

When the scheduler has to decide which stage to assign the next available worker thread to, it calculates a priority value for all the stages, and the candidate with the highest value receives the thread. Moreover, the scheduler aims to fill the pipeline within the available resource restrictions. It is well known that the pipeline performance is determined by the most time-consuming stage. We use function $f'(i) = \frac{t_i \cdot n_i}{N}$ to express the expected running time for stage i. The stage having the largest f' value requires more time to finish the total work without adding additional system resources. Based on this function, the pipeline stages can be ordered and compared to each other. The stage with the lowest f' value has the highest priority. Therefore, adaptive scheduling can be expressed as the following optimization formulation:

$$\begin{aligned} &\textit{minimize} && \max_{1 \le i \le k} f'(i) - \min_{1 \le j \le k} f'(j) \\ &\textit{subject to} && \sum_{i=1}^{k} m_i \le M \end{aligned} \tag{2}$$

Adaptive scheduling not only guarantees optimal system utilization, but can also adapt automatically to workload imbalance. In order to solve the imbalanced workload problem, the execution time for all the stages has to dynamically change in a short time interval. To detect this situation, when the average execution time is calculated, we assign different weights based on previous stage executions. The degree of responsiveness to the workload can be controlled by the assignment of weight values. If the more recent executions receive higher weight, the faster the adaptation to changes. If the weights of past executions are higher, the scheduler is more conservative. For example, assume the execution times for stage i in the last l executions are $\{t_1, t_2, \ldots, t_l\}$ and the corresponding weights are $\{w_1, w_2, \ldots, w_l\}$, respectively. The condition $w_j > w_{j-1}$ holds for every j such that

$1 < j \leq l$. Then, the average execution time for stage i is calculated as follows:

$$\mathrm{T}_i = \sum_{i=1}^{l} w_i \cdot t_i \tag{3}$$

If the workload does not change much, then the T_i values also stay stable. However, when the workload varies suddenly, the scheduler can quickly update the T_i values since the latest executions have the highest weight. Thus, T_i can quickly react to changes in the workload.

4.4. Merge Read Mechanism

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 – called *raw read* – or read the already loaded columns from the database and convert only the additional columns—called *merge read*.

To analyze this problem, we introduce the following notation. The size of the raw file is defined as S. There are n attributes in the raw file. The read throughput is r, while the processing rate is denoted as p. The query requires y attributes from the raw file and z attributes already loaded in the database. Then, the query workload consists of the following components. Let function $f(n, y)$ represent the cost to process attributes from the raw file. The values of function f can be computed from previous accesses to the raw file. Remember that this is not the first time we access the file since some of the data have already been loaded into the database. The workload for data loaded into the database – denoted $R(z)$ – is only from reading, without any extraction operation. For simplicity, the workload to access cached attributes is neglected. Therefore, the execution time for *raw read*, i.e., $T_{\textit{raw read}}$, can be expressed as the following equation:

$$T_{\textit{raw read}} = \textit{max}\left(\frac{S}{r}, \frac{f(n, y+z)}{p}\right) \tag{4}$$

Since SCANRAW is a parallel pipeline operator that supports concurrent reading and extraction, query execution time is decided by the most time-consuming part. This rule is applied in Eq. (4) to obtain the execution time for the query under the assumption that the entire raw file has to be read in order to extract the required attributes. Using the same notation, the execution time for *merge read*, i.e., $T_{\textit{merge read}}$, is calculated in Eq. (5). Compared to *raw read*, *merge read* has to generate data both from the raw file as well as the database. δt represents the overhead caused by the interference introduced by reading from two sources.

$$T_{\textit{merge read}} = \textit{max}\left(\frac{S + R(z)}{r} + \delta t, \frac{f(n, y)}{p}\right) \tag{5}$$

According to which part of the execution is more time-consuming, we classify queries into two categories. When the execution time is dominated by the I/O performance, then the query is I/O-bound. Opposite, if processing takes most of the execution time, the query is CPU-bound. Based on this categorization, the answer to the question *which reading strategy is better?* is determined as follows. The main difference between the two strategies is that a portion of the *raw read* computation is transformed into I/O work in *merge read*. Therefore, if the execution is I/O-bound for *raw read*, it is still I/O-bound for *merge read*, which incurs even more I/O work. If the system has plenty of CPU resources to support the extra tokenize and parse tasks while still fully overlapping with the I/O operation, then *raw read* is the better choice. Similarly, if the execution is CPU-bound for *merge read*, then it is also CPU-bound for *whole read*, which incurs even more work, i.e., $f(n, y+z) > f(n, y)$. This situation corresponds to a system with a fast I/O component, where most of the time is spent for processing. In-memory databases are a good example that illustrates this scenario. The most complex situation arises when the execution is CPU-bound for *raw read* and I/O-bound for

merge read. In this case, the answer is determined by comparing $\frac{f(n,z)}{p}$ and $\frac{R(z)}{r} + \delta t$. The overhead δt is hard to measure, since it depends on the system configuration and data organization. A simple solution is to switch to *merge read* whenever the execution in *raw read* is CPU-bound. Hence, *merge read* is more likely to be chosen when a large part of the required columns are loaded in the database. However, this imposes severe restrictions on cache operation since that is where chunks are assembled. Then, the question becomes: Can SCANRAW efficiently support *merge read*? We argue that the answer is yes and we provide the details of our solution in the following.

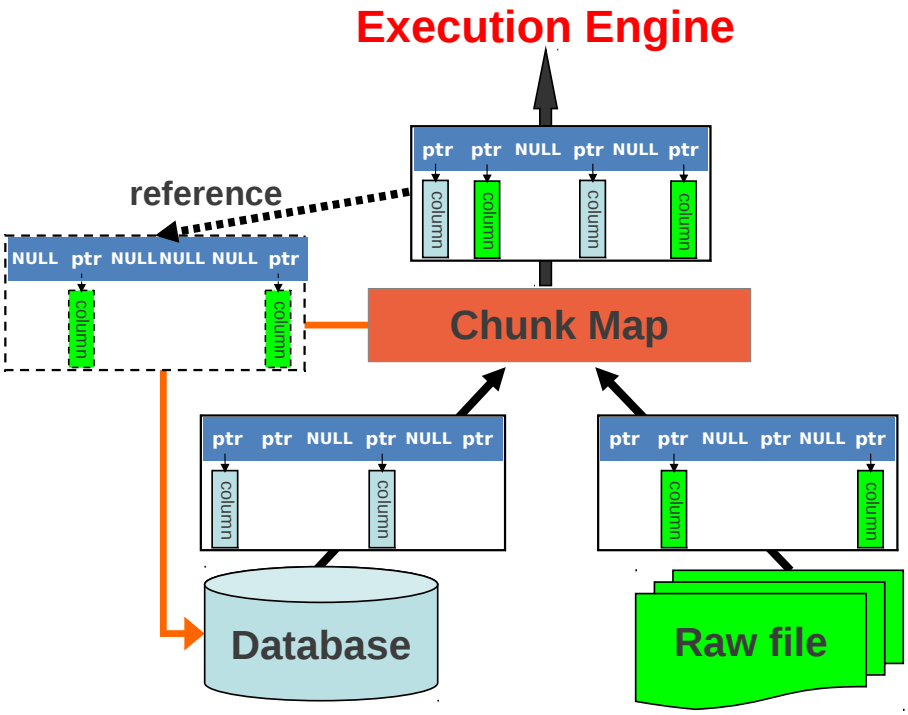


Fig. 5: Merge read mechanism workflow.

Workflow. When SCANRAW receives a query with the list of attributes for a relation, the first optimization mechanism is to utilize the range index pair (Min, Max) to eliminate the unnecessary chunks. A list of chunks that have to be read is generated. Since the loading status for every chunk is different, when reading a chunk, SCANRAW inspects the metadata to divide the list of attributes into two sets. One set contains the columns that are loaded into the database, while the other set contains the remaining columns that can be accessed only from the raw file. Figure 5 shows the *merge read* workflow in SCANRAW. This workflow is based on the chunk structure and column-based storage schema depicted in Figure 2. Chunks that have columns loaded in the database generate two types of read requests—one request to the database and another to the raw file. The cost to execute these requests is quite different. Generating data from the database requires only reading since data are already in binary format. Extracting data from the raw file is a complicated process with multiple stages, as shown in Figure 1. Therefore, when the READ thread receives these two requests, it first starts to read data from the raw file since the conversion to binary format can be overlapped with data reading from the database. Columns are cached in memory until all the columns corresponding to

the chunk are read from disk. Moreover, the columns from the raw file are soft-copied by SCANRAW, as shown in Figure 5, and are sent to the WRITE thread for speculative loading.

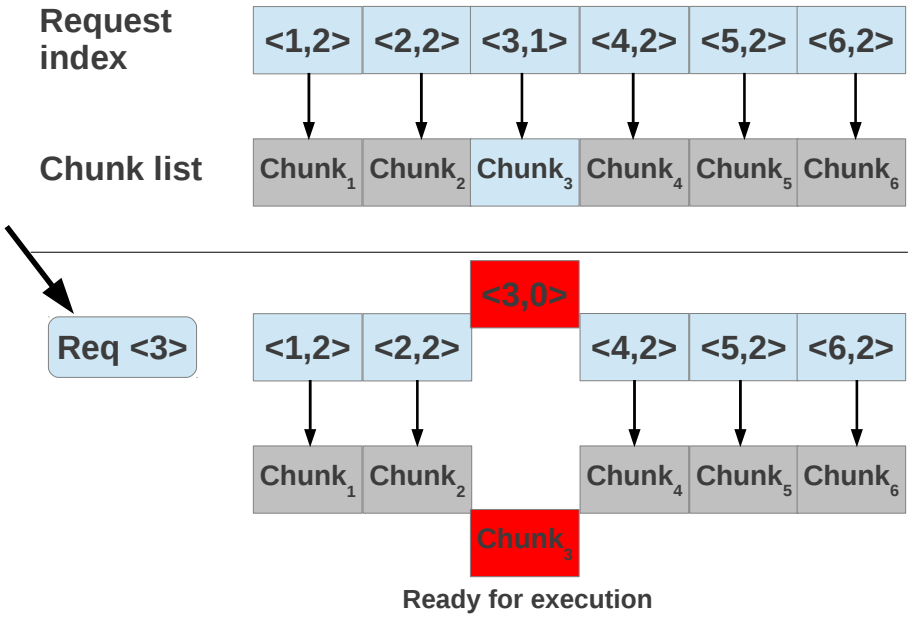


Fig. 6: Example of a Chunk Map instance in merge read.

Chunk Map data structure. All the threads are executed concurrently and asynchronously in SCANRAW, hence chunk requests having multiple data sources have to be synchronized. The memory-resident chunk map data structure (Figure 6) is designed in order to synchronize these requests and trace the status of each chunk. It consists of two entities—the request index and the chunk list. The request index is used to trace the completion status of every chunk and test whether the chunk is ready to be sent to the execution engine for processing. The chunk list is used to store the chunk columns. All the columns have to be loaded into the chunk list before the chunk can be processed. When generating a chunk, a key-value item is created in the request index. The key is the chunk id, while the value is the number of data sources containing data for this particular chunk. If the value is initialized with value 1, it means the chunk does not require synchronization since it is stored either into the database or the raw file. However, if the value is set to 2, it indicates that the chunk has to wait and merge data both from the database and the raw file. When either of the requests is returned, SCANRAW first finds the corresponding item, then updates the value, and lastly checks whether the chunk is complete. For example, in Figure 6, the index value of $chunk_3$ is 1, which means $chunk_3$ only needs one step to be generated. When $Req < 3 >$ arrives, the value for $chunk_3$ drops to 0. The chunk is complete, thus both the index request and the chunk can be removed from the chunk map. The chunk is further sent to the execution engine for processing.

Group reading. It is well-known that sequentially scanning large amounts of data maximizes the disk I/O throughput. However, in the case of *merge read*, when many chunks have more than one data source, SCANRAW has to read data from non-contiguous disk blocks. If the chunks are generated sequentially, the disk driver cannot utilize the sequential reading pattern. Hence, we design the *group reading* mechanism which, instead of generating chunk requests sequentially, splits the chunk list into multiple groups having similar size. For every group, columns from different chunks are

generated together, according to their data source. The chunks are generated group by group. Since all the chunks in a group have to reside in memory before they are sent to the execution engine, the size of the group is determined by the capacity of the system memory. Take the query shown in Figure 6 as an example, and consider that all the chunks have two data sources. The overall number of chunks is 100, with 500 MB per chunk. Moreover, the memory capacity is 10 GB. Then, the maximum number of chunks residing in memory at the same time is $10,000/500 = 20$, which is the group size. Thus, for every 20 chunks, SCANRAW generates two lists of requests. One request is for reading the columns corresponding to all the chunks in the group from the raw file, while the other request is responsible for retrieving the remaining columns from the database. Notice that this separation is not essential since the disk controller already optimizes concurrent requests. In order to minimize memory consumption, the requests are ordered based on the size of the retrieved data. The requests with smaller data footprint are processed first since this minimizes the duration of the memory occupancy.

4.5. 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.

Generalization to any raw file format. In order to add support for processing any type of raw file in SCANRAW, all is required to implement are new TOKENIZE and PARSE functions – and MAP when the separation makes sense – specific to that file type. Everything else can be kept the same. Only in the extreme case of an empty task – for example, binary raw files such as BAM do not require TOKENIZE and PARSE becomes exclusively MAP, transforming data from the format returned by the file access library to the internal processing format – architectural modifications are required, i.e., the corresponding buffer and the consumer thread monitoring it are removed from the pipeline altogether. It is this generalization to any type of raw data that gives SCANRAW its meta-operator characteristic. While the optimizations in TOKENIZE and PARSE are not applicable to binary data, the optimizations in all the other pipeline stages are still applicable—pre-fetching in READ and caching in MAP. Moreover, the SQL-like interface to process raw data without loading is probably the most important benefit of SCANRAW when compared to file access libraries which require writing an entirely new program for every query.

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 [Idreos et al. 2011]. 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.

5. SPECULATIVE LOADING

By default, SCANRAW operates as a parallel external table operator. It provides instant access to data without pre-loading. This results in zero time-to-query 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.

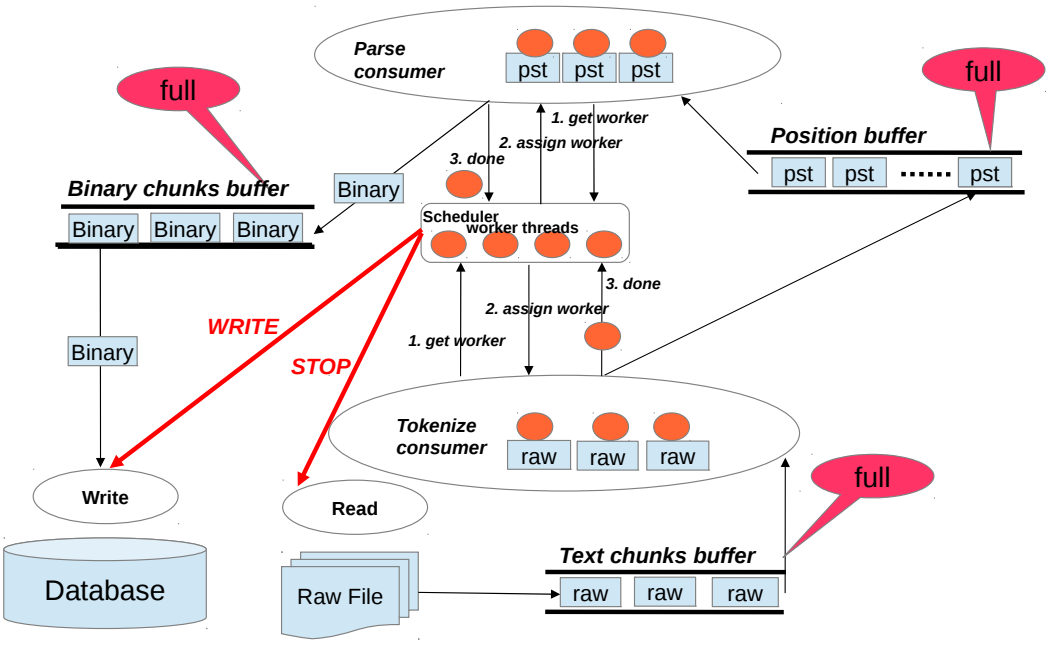


Fig. 7: Detection mechanism for triggering speculative loading.

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 several solutions in the literature that are related to speculative loading. In invisible loading [Abouzied et al. 2013], 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 [Alagiannis et al. 2012] 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 [Idreos et al. 2011] – 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 (Figure 7). 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 4) whenever worker threads become available and `WRITE` returns.

Why does speculative loading interfere minimally 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 [Abouzied et al. 2013] 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. Since the number of chunks that have to be converted from raw format into the database representation decreases with each query, subsequent queries run faster than the previous ones. Until all data are loaded into the database.

6. MULTI-STEP LOADING (MSL)

Using speculative loading, `SCANRAW` could instantly access to data and utilize the spare I/O resource to load data into database without affecting the query execution.

Is it possible to find another solution even faster than speculative loading. This is the research question we ask in this section. Our solution is *multi-step loading*. It contains many advantages just like speculative-loading, such as instant access to the data, speeding up the subsequent queries over the same data set. But in some cases, the MSL could achieve better execution time. The main idea in MSL is to find the computation bottleneck and try to postpone unnecessary workload in order to speed up the current query execution. The intuition is that query processing time is decided by the size of computation workload when the execution is CPU-bound.

How does multi-step loading work. To MSL, the concerning attributes in a query are not the same. Those involving numerical operations, such as `+`, `-`, `*`, `/`, or aggregate functions, such as `SUM`, `AVG`, `MIN`, `MAX`, are viewed as "binary columns", which means they have to be transformed into binary format before query execution. All the other attributes, even though their type is not `string`, can be represented as text in the final query result. Thus, we call them text attributes since they do not require parsing. The central idea in multi-step loading is to let `SCANRAW` decide

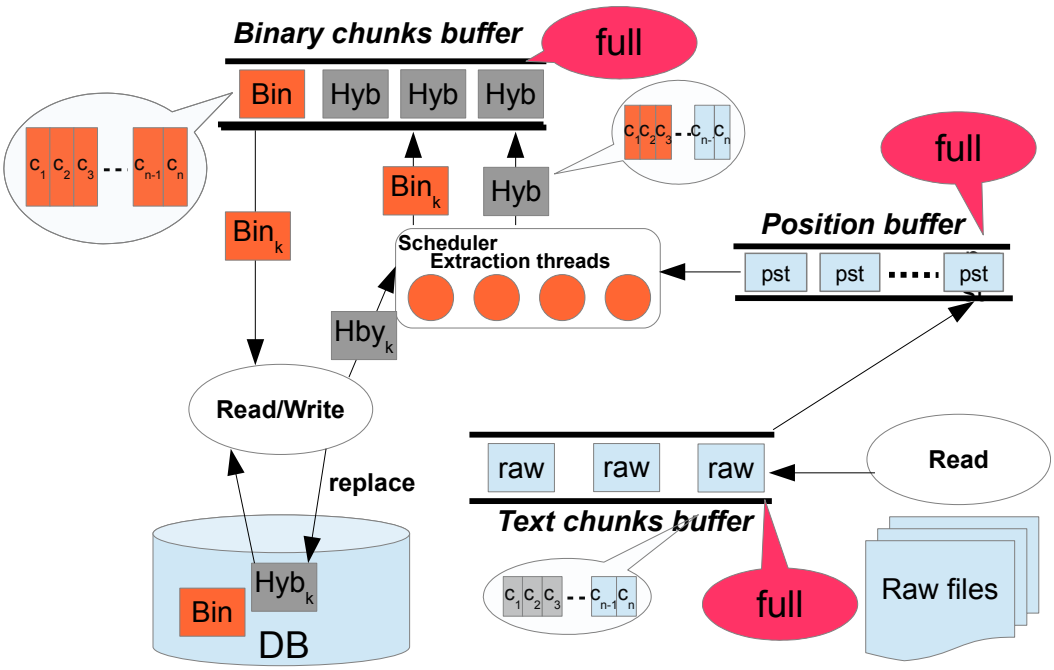


Fig. 8: Detection mechanism for triggering multi-step loading.

adaptively at runtime the access plan for these "text columns" while maintaining optimal query execution performance.

In order to illustrate the MSL workflow, let us consider an example based on table `lineitem` in the TPC-H benchmark. `lineitem` has 16 attributes with different data types, such as `integer`, `decimal`, and `string`. Consider the following two queries executed over raw text data generated by the TPC-H generator:

Q_1 : **SELECT** $l\_commitdate$, $l\_orderkey$, $l\_linenumber$, $l\_discount$, $l\_extendedprice$, l_tax
FROM `lineitem`
Q_2 : **SELECT** $l\_commitdate$, $l\_orderkey$, $(l\_extendedprice * l\_discount * (1 + l\_tax))$
FROM `lineitem` **WHERE** $l\_linenumber \geq 3$

Attributes in Q_1 are all "text columns", since there is no any numerical operation on any columns. However in Q_2 $l\_extendedprice$, $l\_linenumber$, $l\_discount$, and l_tax, are binary columns and they have to be parsed before query execution.

How does multi-step loading work for the first query. MSL supports instant access to the data. Since the scheduler monitors the utilization of the buffers and assigns worker threads for task execution, it can identify the status of `READ` through text chunks buffer (Figure 8). When the buffer is empty the processing is I/O-bound, the execution time is decided entirely by the read/write throughput. In this case, all the attributes in the query would be parsed due to the plenty of CPU resources. MSL works exactly the same as speculative loading. When the `READ` is blocked, the processing becomes CPU-bound. At this time, MSL generate the query access plan to delay parsing work as much as possible. In our example, if Q_1 is the first query to be processed, MSL reads, tokenizes, and stores the 6 attributes in the database as `string`, without parsing them at all. However, if Q_2 becomes the first query, only columns $l\_commitdate$ and $l\_orderkey$ can be delayed for parsing, since $l\_linenumber$, $l\_discount$, and l_tax are all "binary columns". Both in Q_1 and Q_2, some

parsing workload could be delayed to subsequent queries, which improves the execution time for the current query. A possible question is how many columns should be delayed for parsing? This is dynamically controlled by SCANRAW. Either there are no other columns can be delayed for, or the query becomes I/O-bound. If all text columns have been delayed for parsing and the query is still CPU-bound, then MSL can use the spare I/O resource to load data into the database, similar to speculative loading. But text attributes are loaded as `string`, even though they have a different type. The more attributes are loaded into the database, the higher the probability that SCANRAW can avoid reading the raw file.

How does multi-step loading process subsequent queries. After the first query has been processed, some data are loaded into the database. However, the format of the data in the database is not fixed. There are attributes loaded in binary format and attributes loaded in text format. How to generate the optimal access plan for subsequent queries is the problem to solve. In multi-step loading, data can be in three formats: text format in the raw file, text format in the database, and binary format in the database. Let us assume that Q_1 is executed first and consider the access plan for Q_2. If all the attributes can be retrieved from the database in binary format, then Q_2 is executed as a standard database query. In this situation, SCANRAW has no additional work to execute. Another possibility is that all the attributes are retrieved from the database, but not all of them are in binary format. For example, assume attributes $l\_linenumber$, $l\_discount$, and l_tax are stored as text inside the database. Then, when SCANRAW starts to process Q_2, it should read data from the database since it has smaller size. However, these attributes have to be parsed before they can be processed by the execution engine. The same conversion pipeline is used for this purpose, albeit with a different data source. After the attributes are transformed into binary format, they can be loaded into the database in binary format, to replace the former text representation, following the speculative loading mechanism. In the last scenario, columns are distributed across all the formats. For instance, in Q_2, $l\_linenumber$ is loaded as `integer` into the database, $l\_orderkey$ and $l\_discount$ are saved as text, and l_tax is still in the raw file. At this point, reading the raw file and tokenize and parse l_tax is inevitable. If the extraction process is I/O-bound, then we can extract more attributes instead of reading them from the database, until the processing reaches a balance between CPU and I/O utilization. If the extraction is CPU-bound, then reading binary data decreases the tokenizing and parsing work, while reading only text from database eliminates tokenizing. These dynamic changes can move the execution between CPU-bound and I/O-bound status. SCANRAW can adapt to the changes through the thread pool mechanism and remain optimal.

7. 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 sequence 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? What speedup does SCANRAW achieve?
— What is the performance of speculative loading and multi-step loading compared to external tables and database loading and 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?
— How fast can SCANRAW extract tuples from the raw file?
— How does the scheduling algorithm affect query execution performance?
— What resource utilization does SCANRAW achieve?
— How does the SCANRAW performance compare to other file access libraries and systems that support raw file processing?

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 SAM and BAM 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 [Arumugam et al. 2010; Cheng et al. 2012; Cheng and Rusu 2014a] 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 only mapping to a different processing representation, without changes to the SCANRAW architecture.

System. We execute the experiments on a standard server with 2 AMD Opteron 6128 series 8-core processors (64 bit) – 16 cores – 64 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. According to hdparm, 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.

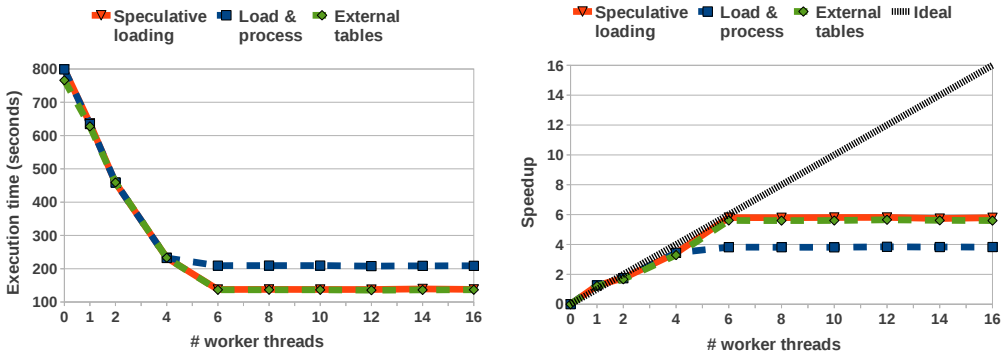
Methodology. We perform all experiments at least 3 times and report the average value as the result. If the experiment consists of a single query, we always enforce data to be read from disk by cleaning the file system buffers before execution. In experiments over a sequence of queries, the buffers are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

7.1. 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 [Alagiannis et al. 2012; Abouzied et al. 2013]. 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.

7.1.1. Parallelism and speedup. Figure 9 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 9a) – and the speedup (Figure 9b) – 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



(a) (b)

Fig. 9: Execution time (a) and speedup (b) as a function of the number of worker threads.

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 conversion to binary and loading into the database completely. Essentially, loading comes for free since the disk is idle. In Figure 9a, 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. 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.

7.1.2. Percentage of loaded data. The effect of parallel processing on speculative loading is illustrated in Figure 10. 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.

7.1.3. Vectorization. Figure 11a depicts the comparison for tokenizing a chunk between an implementation with SSE vectorized instructions and a standard non-vectorized implementation. As the number of worker threads increases, the performance of the vectorized version stays constant and outperforms the non-vectorized implementation by a factor of 2. Figure 11b depicts the overall query execution time for the two mechanisms. When the number of worker threads increases, the difference in execution time drops to the point where they are identical—for 12 or more threads. There are two reasons for this. First, the advantages of vectorization are diluted by multi-threading execution. Assume s to be the time for tokenizing a chunk. Then, when processing n chunks with a single thread, the vectorization mechanism can save at most $(n-1)\cdot s$ time from execution. However, if there are m threads working in parallel, the overall execution time reduces by a factor of m. The savings due to vectorization are upper-bounded by the number of chunks assigned to a thread—only n/m in this case. The second reason is that tokenization is a relatively small portion from the overall query execution time, as proved by the detailed pipeline analysis presented in the following.

7.1.4. Pipeline analysis. Figure 12 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 increments.

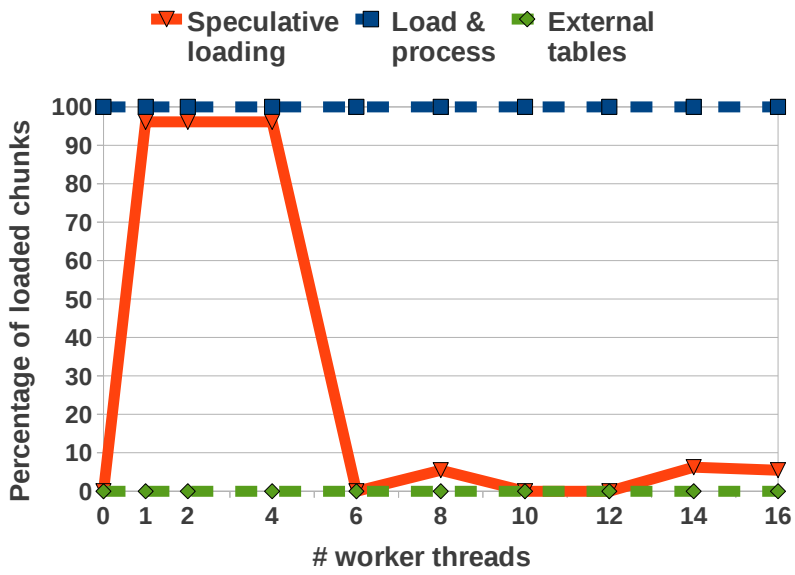
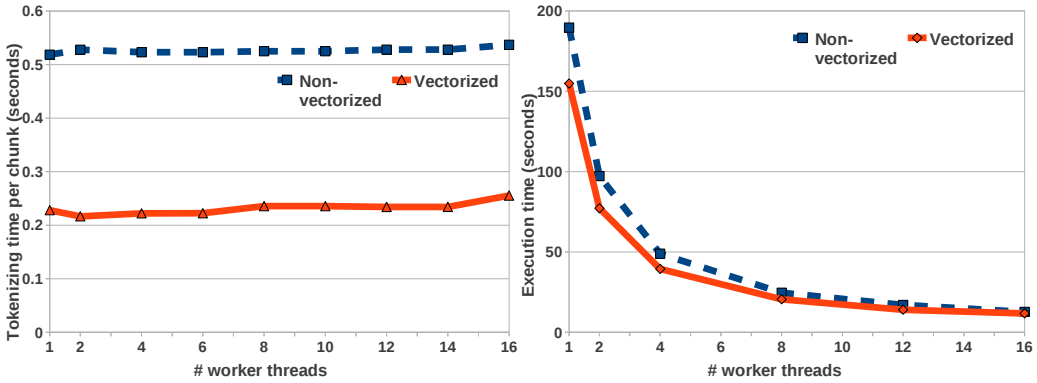


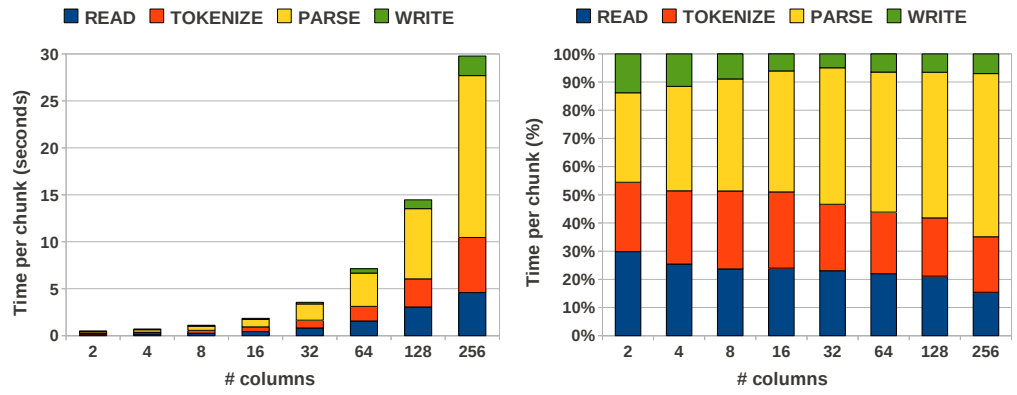
Fig. 10: Percentage of loaded data as a function of the number of worker threads.



(a) (b)

Fig. 11: The effect vectorization has on tokenization (a) and overall query execution time (b).

The number of lines in all the files is 2^{26}. The `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 12a. As expected, when the number of columns increases, so does the chunk processing time. Specifically, 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 regime 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 9a)



(a) (b)

Fig. 12: Pipeline execution time: (a) absolute, (b) relative.

over raw files, thus making data loading obsolete. Figure 12b gives the relative distribution of the data in Figure 12a. 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 experiment illustrates two important aspects. First, it proves that PARSE is the stage to optimize in order to make raw file processing efficient. And second, the workload distribution across the extraction stages varies significantly with the number of attributes required by the query. SCANRAW avoids the problems generated by this imbalance by using a dynamic super-scalar pipeline architecture.

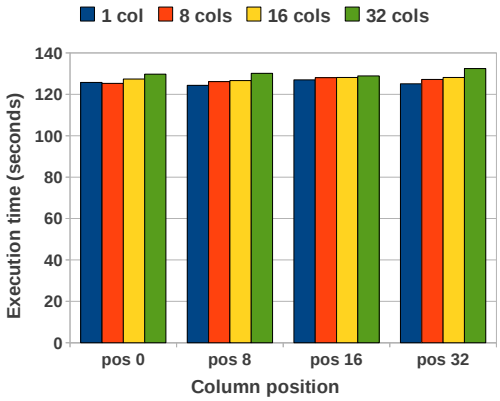


Fig. 13: Position and number of columns.

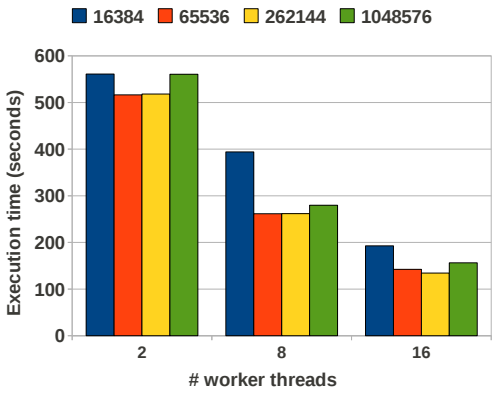


Fig. 14: Chunk size.

7.1.5. Position and number of columns. Figure 13 depicts the effect of two parameters on the 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. PARSE becomes the most time-consuming stage in the extraction and determines the overall pipeline performance. The position of the first column in the subset does not impact performance at all. The reason for this is that the minimal increase in tokenization time is completely hidden in the parallel execution with pipeline structure. These results confirm once more that PARSE is the stage to optimize in raw file processing.

7.1.6. 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 14. The chunk size has to be in the proper range. If the size is not large enough, the overhead introduced by the dynamic allocation of tasks to worker threads impacts heavily the performance. 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 $2^{19} \approx 500K$.

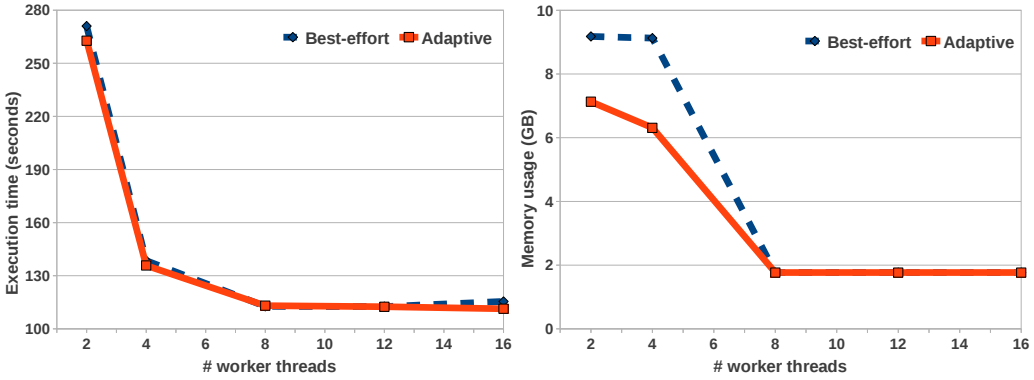


Fig. 15: Thread scheduling algorithm comparison: (a) execution time, (b) memory usage.

7.1.7. Thread scheduling. In this experiment, we compare the performance of the two scheduling algorithms introduced in Section 4.3—best-effort and adaptive. We vary the number of worker threads in the pool from 2 to 16. The execution migrates from CPU-bound to I/O-bound, accordingly. The goal of the experiment is to investigate the behavior of the two scheduling algorithms under the two types of execution regimes. During query processing, the chunk size changes when converting from text to binary, which causes the imbalanced workload. Thus, we also evaluate the amount of memory saved by employing the adaptive algorithm when compared to the best-effort strategy.

The results are depicted in Figure 15. In Figure 15a, on the left, the vertical axis is the execution time, while in Figure 15b, on the right, the vertical axis represents the memory usage in GB. When the number of worker threads is less than 8, the execution is CPU-bound. While the processing speed of adaptive is slightly faster, its memory usage is considerably smaller than for best-effort. The reason is that the adaptive algorithm changes its assignment configuration dynamically, according to the imbalanced workload, to guarantee that the pipeline is fully utilized. Moreover, adaptive optimizes memory consumption at the same time. However, when the number of worker threads increases beyond 8, the execution becomes I/O-bound, which means the workload can always be processed immediately. Thus, there is no difference between the two algorithms according to both

comparison criteria. Notice also that the workload can be processed efficiently and with minimal memory consumption.

7.1.8. Merge read mechanism. The merge read mechanism (Section 4.4) is used to merge necessary query data from the database and the raw file. Database data are in the internal processing format and do not require conversion. Raw file data have to be extracted and mapped into the internal processing representation. In this experiment, we investigate the performance of merge read. In particular, we investigate how the behavior of merge read changes as the processing workload varies and when merge read is the optimal choice. In order to illustrate the characteristics of the merge read mechanism, we run the raw read strategy as a comparison.

The experiments presented in this section, use a dataset of 40 GB, containing 2^{26} tuples. Each tuple contains 64 attributes with integers distributed randomly in the range $[0 - 10^9)$. Half of the dataset, i.e., attributes $1-32$, are also loaded into the database. The experimental setup is as follows. We create nine `SELECT-PROJECT` queries, denoted $\{Q_1, Q_2, \ldots, Q_9\}$. The queries access 32 columns grouped into continuous ranges. Selectivity is 100% for all the queries, as there is no `WHERE` clause. However, the starting column index is different for each query. Q_1 retrieves attributes from index 1 to 32. The starting position for Q_2 is 5. The starting position for subsequent queries increases in increments of 4, e.g., 9 for Q_3, 13 for Q_4, and so on. The queries can be divided into three categories based on the source of their data. The first type is Q_1, which can get all the data directly from the database, since all the accessed attributes are loaded. The second type is Q_9, which accesses data exclusively from the raw file. For these two queries, *merge read* works the same as *raw read*, therefore their execution time should be almost identical. The last category contains the remaining 7 queries, Q_2 through Q_8. Along with the increase of the starting column index, the proportion of loaded data decreases, from 87.5% for Q_2, to 12.5% for Q_8. Therefore, the workload of `PARSE` and `TOKENIZE` increases from Q_1 to Q_9. Additionally, we vary the number of worker threads used for data extraction in order to change the processing type from CPU-bound into I/O-bound. We execute the queries using *merge read* and *raw read*, and monitor the performance across the two mechanisms.

The results are shown in Figure 16. Figure 16a depicts the result when there is a single worker thread dedicated for chunk extraction. In this configuration, the extraction stages are sequentially run, hence the running time is the sum of the times spent in each stage. That is why the execution time increases both for *merge read* and *raw read*. We can see that the execution time for these two methods is almost the same both in Q_1 and Q_9, as expected. However, *merge read* is always faster in this case since the cost of reading additional columns from the database is less than the cost of extracting them from the raw file. From the figure, we observe that the larger the number of columns read from the database, the more gains *merge read* provides. The maximum gain corresponds to Q_2, which accesses 87.5% of loaded data.

Figure 16b depicts the results for two worker threads. In this situation, `TOKENIZE` and `PARSE` can be executed in parallel. That is the reason for the reduction in execution time by almost half for *raw read*, when compared to Figure 16a. When the starting column index increases, the workload for `TOKENIZE` and `PARSE` augments as well. Hence, the running time for *raw read* increases smoothly. The behavior of *merge read* is more interesting. For Q_2, the running time is nearly the same for both methods, which means that the time for extracting the additional columns is nearly equal to the time spent for reading the columns from the database. From Q_3 to Q_6, the running time decreases steadily, since these queries are I/O-bound. Thus, there exist spare computational resources to execute more work without affecting the running time. Moreover, the less data are read from the database, the better the performance. However, for Q_7 and Q_8, which are CPU-bound, the running time increases proportionally with the number of additional columns that have to be extracted from the raw file.

Figure 16c and 16d depict the results when the number of worker threads increases to three and four, respectively. For these two configurations – and any number of worker threads larger than 4 – *raw read* is always faster than *merge read*. The reason is that the queries become I/O-bound, thus

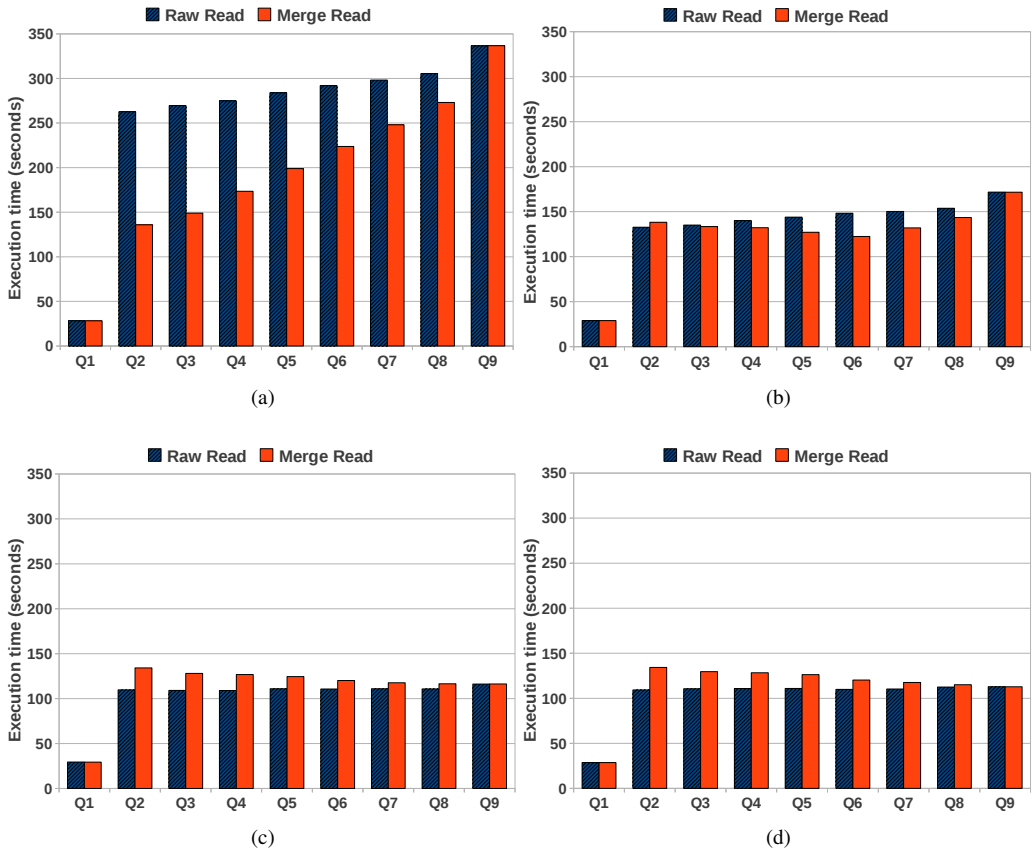


Fig. 16: Comparison between *merge read* and *raw read* as a function of the number of worker threads used for data extraction: 1 (a), 2 (b), 3 (c), and 4 (d).

the running time is determined exclusively by the additional reading from the database. Reading less amount of data is the strategy to decrease the execution time in this situation. Since the amount of data read by *raw read* is the same for queries Q_2 to Q_8, the execution time is nearly constant. The running time increases slightly only for Q_9, which is CPU-bound. The *merge read* execution time drops smoothly because the additional data read from the database decreases as well. When the number of worker threads reaches four, all the queries become I/O-bound, both for *merge read*, as well as for *raw read*. In this case, *raw read* is the better choice.

7.1.9. Speculative loading for a query sequence. Figure 17 and 18 depict the `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

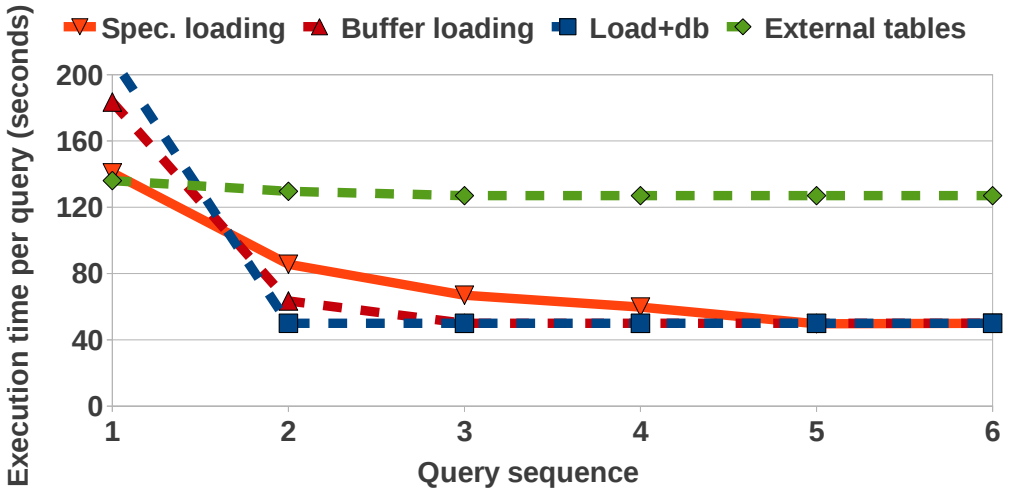


Fig. 17: Execution time for query i.

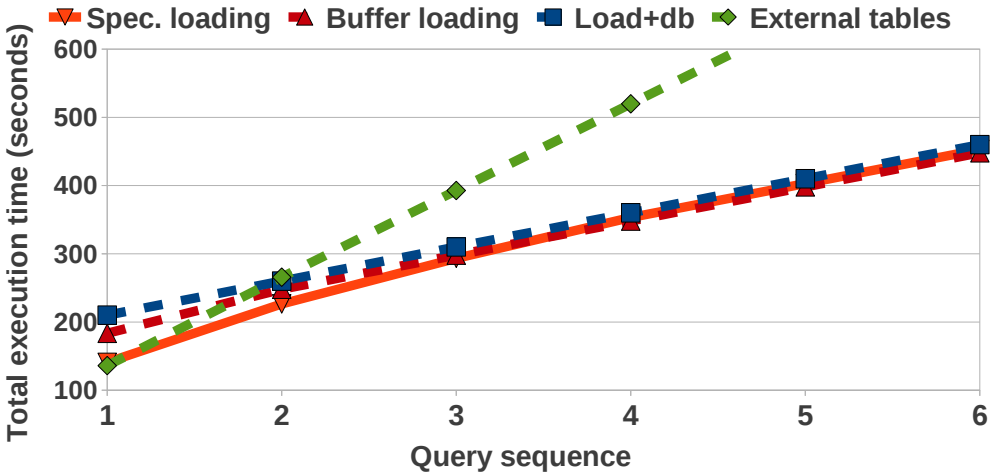


Fig. 18: Overall execution time up to query i.

during query processing is non-deterministic, it is more difficult to observe. Figure 17 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 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 17, 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 18 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.

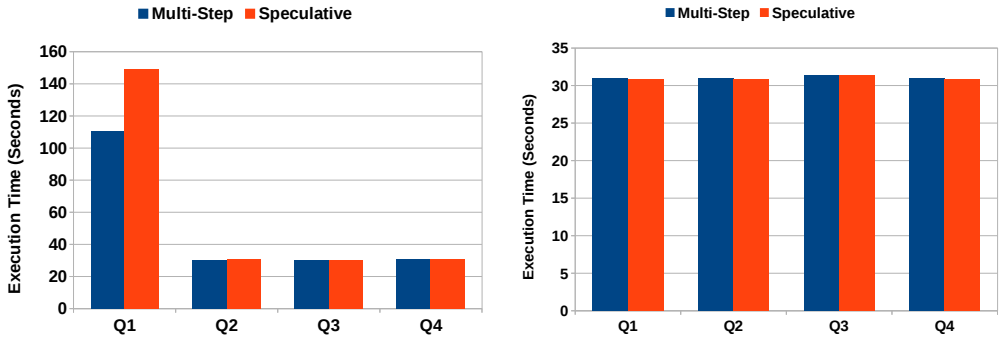


Fig. 19: Multi-step loading for a sequence of identical queries: (a) 2 threads; (b) 16 threads.

7.1.10. Multi-step loading for a query sequence. We compare the performance of MSL against speculative loading using two different query sequences. The first sequence consists of 4 identical queries, i.e., SELECT $\sum_{i=1}^{15} C_i$, C_{16}, C_{17}, ..., C_{31} FROM TABLE WHERE $C_{32} < 10$, executed over a file with 2^{26} tuples and 32 columns, all of which represented as floating point numbers. Only half of the attributes are required to do the computation, i.e., have binary type, while the other half are used for printing. This query allows us to detect and quantify the effect of the input data source on query performance. Figure 19a depicts execution time for the two loading approaches, when SCANRAW is configured with two worker threads. The processing is CPU-bound in this case. Multi-step loading runs faster than speculative loading for the first query because SCANRAW postpones parsing of half of the attributes. Moreover, SCANRAW is also capable to speculatively load all

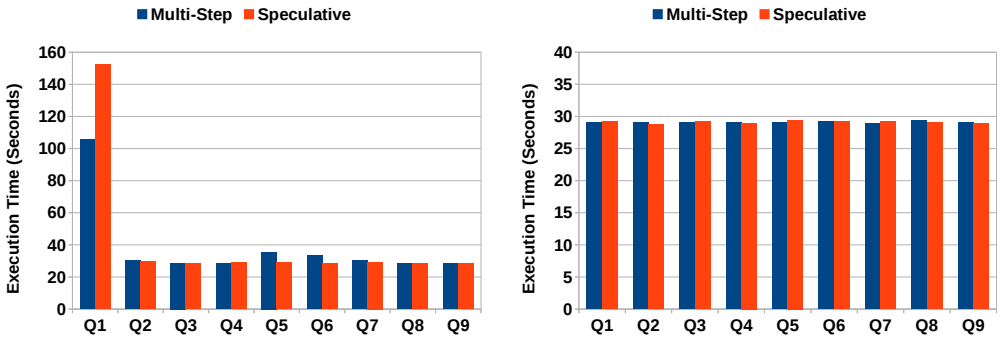


Fig. 20: Multi-step loading for a sequence of different queries: (a) 2 threads; (b) 16 threads.

the accessed columns during query execution. At the end of Q_1, all the 32 attributes are loaded into the database. However, their format is different. While all the attributes are binary in speculative loading, half of them are text in multi-step loading. In subsequent queries, data are read exclusively from the database. The execution time is similar for the two loading methods since the processing is I/O-bound. Speculative loading becomes standard database processing since there is no extraction work to execute. The only difference in multi-step loading is that half of the attributes are treated as `string`, even though they are `decimal`. As long as they are not involved in arithmetic operations, this is perfectly fine. When the number of worker threads is 32, query processing is I/O-bound even for the first query. Figure 19b depicts the results in this case. As expected, the two loading strategies are identical. Their execution time is similar to the execution time for queries Q_2-Q_4 in the case of two threads since the same amount of data is read from disk.

The second sequence contains 9 queries. All the attributes are accessed by every query. The number of binary attributes is 4 in the first two queries, 8 in the following two, 16 in Q_5 and Q_6, and 32 in the last three queries. Figure 20 depicts the results. We follow the same method to measure execution time when the processing is CPU-bound and I/O-bound, respectively. Figure 20a corresponds to CPU-bound processing, i.e., two worker threads. As before, MSL runs faster than speculative loading for the first query and is similar for Q_2. When executing Q_3 and Q_4, MSL has to parse four additional attributes, loaded as `string` in the database. Nonetheless, the two methods have almost identical execution time. This is because MSL processing is still I/O-bound, even with the additional parsing. When the number of binary attributes increases to 16 (Q_5 and Q_6), though, the additional parsing required in MSL results in a slight increase in execution time. The reason is that parsing turns the process from I/O-bound into CPU-bound. While MSL parses the attributes into binary format, it speculatively replaces the corresponding text content with binary. Therefore, after several queries, the execution converges to speculative loading, which is I/O-bound. The results for I/O-bound execution, i.e., 16 worker threads, are shown in Figure 20b. They confirm that MSL and speculative loading are identical even for this workload.

7.1.11. Resource utilization. In Figure 21, 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,

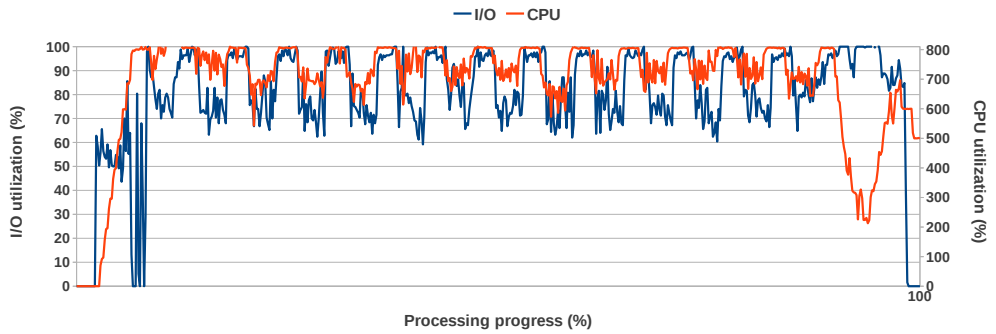


Fig. 21: Resource utilization in SCANRAW.

the scheduler resumes reading and disk utilization goes back to 100% since a sequence of chunks are typically read at a time.

7.2. Real Data

In order to evaluate SCANRAW on a real dataset, we use genomic sequence alignment data from the *1000 Genomes* project[11]. 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 I 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 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.

FITS is a common binary format which is widely used in astronomy to store, transmit, manipulate, and archive data. For instance, the Sloan Digital Sky Survey (SDSS)[12] data are available in FITS format. Besides the image data, FITS files can also store tables, either in ASCII or binary. A widely used tool to handle FITS files is the C library CFITSIO[13], developed by NASA. CFITSIO provides a rich API to manipulate data in FITS files. SCANRAW can execute queries on FITS files containing binary tables directly. We enable SCANRAW to access FITS files by replacing TOKENIZE and PARSE with CFITSIO API function calls. To make FITS data available for processing by the execution engine, SCANRAW has only to map them to the binary chunk structure. This is a simple memory mapping operation that incurs no overhead.

Table I displays the results for processing a FITS binary table with 64 integer attributes and 67 million rows. The total size is 8.1 GB. All the raw data processing methods investigated in this paper

[11] http://www.1000genomes.org/data
[12] http://www.sdss3.org
[13] http://heasarc.gsfc.nasa.gov/fitsio

are compared. As in the case of BAM data, there is minimal difference between external tables and loading. This is because the file access library has major inefficiencies in retrieving data from the raw file. It turns out that not reading the data is the problem, but rather creating the processing data structures. As a result, the spare I/O throughput can be used for loading data into the database. This is exactly what SCANRAW achieves through speculative loading. Based on these results, we conclude that, for binary data, SCANRAW operates in a regime that is closer to data loading. But this is due entirely to the file access library performance.

Table I: SCANRAW execution time (seconds) for real data.

Method	**SAM** (145 GB)	**BAM** (26 GB)	**FITS** (8.1 GB)
External tables	370	2,714	220
Data loading	945	2,722	224
Database processing	122	122	29
Speculative loading	370	2,717	223

7.3. SCANRAW vs. Impala vs. MySQL

In this experiment, we compare the SCANRAW external table functionality against state-of-the-art data processing systems that support raw file execution. We include MySQL (5.1.73) and Impala (2.1.0) in the comparison since these are the only two freely available systems we are aware of. MySQL provides external table functionality through the CSV storage engine, which enables direct querying over text CSV files, without loading. Impala accesses data stored in the HDFS[14] distributed file system. HDFS splits files into chunks that can be retrieved and processed independently. Impala uses task parallelism for processing multiple chunks concurrently, as long as they are read fast enough from HDFS. In order to let Impala read directly from the local file system, HDFS is configured in "short-circuit" mode. The experiments are executed on a dedicated server with an Intel(R) Core(TM) i7-4770 CPU, 32 GB of RAM, 2 TB of disk storage, and using CentOS 6.6. The system is different because Impala requires CPUs with support for vectorized instructions, e.g., SSE4 or above.

We run experiments over three CSV files, containing 4, 16, and 64 integer attributes, respectively. There are 2^{26} rows in each file. Their sizes are 2.5 GB, 10 GB, and 40 GB, respectively. The query computes the average of the sum of all the attributes across all the tuples. Pure external tables access the entire file. Figure 22 depicts the execution time across the three systems. SCANRAW achieves the best performance in all the cases. The difference increases with the number of attributes. MySQL is almost as efficient as SCANRAW for 4 attributes, but the lack of multi-thread parallelism becomes dominant for a larger number of attributes since more computation is required for the conversion. The same trend can be observed for Impala. However, since Impala supports task parallelism, we found the problem to be the inefficient data access through HDFS. The "short-circuit" read mechanism does not seem to have a significant impact in our experimental setting.

7.4. 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. While additional improvements can be obtained through the use of vectorized SIMD instructions, their impact is minimal if they are applied only for tokenizing—this is the case in the literature [Mühlbauer et al. 2013]. 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

[14] https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html

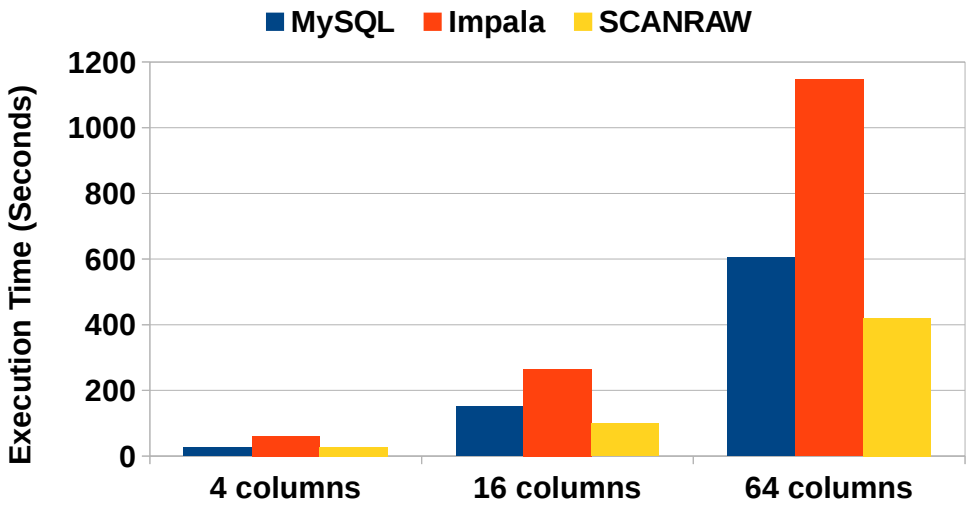


Fig. 22: Comparison of external table mechanisms.

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, facilitated by an adaptive scheduling algorithm that provides a significant reduction in memory usage when compared to the best-effort alternative. Due to parallel conversion from text to binary, `SCANRAW` outperforms BAMTools by a factor of 7, while processing a file 5 times larger. Data extraction for all the accessed attributes is the optimal strategy whenever the raw file has to be read. Merging data from the database and the raw file proves effective only when the number of threads allocated to data extraction is one, at most two.

8. RELATED WORK

Several researchers [Gray et al. 2005; Ailamaki et al. 2010; Stonebraker et al. 2009; Kersten et al. 2011] have recently identified the need to reduce the analysis time for processing tasks operating over massive repositories of raw data. In-situ processing [Lorincz et al. 2003] 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 [Witkowski et al. 2011; Alur et al. 2008] and various optimizations that eliminate the requirement for scanning the entire file to answer the query [Idreos et al. 2011; Alagiannis et al. 2012; Ivanova et al. 2012]. The second category is organized around the MapReduce programming paradigm [Dean and Ghemawat 2008] 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[15], 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

[15]https://github.com/julianhyde/optiq

the conversion code by caching the already converted data in memory or storing it in binary format inside a database [Abouzied et al. 2013].

SCANRAW [Cheng and Rusu 2014b]. The SCANRAW meta-operator is introduced in [Cheng and Rusu 2014b]. The super-scalar pipeline architecture is designed following a detailed analysis of the conversion process from the raw file representation to the processing format. Speculative loading is proposed as an adaptive mechanism to load data into the database whenever there is spare disk I/O throughput—and without interfering with query processing. In this work, we bring three novel contributions that enhance the functionality of the SCANRAW meta-operator significantly and provide a deeper understanding of how to process raw files efficiently on modern computer architectures. First, in addition to data partitioning parallelism and pipelining, we also integrate vectorized SIMD instructions, as a new form of parallelism supported by the instruction sets of modern CPUs, in SCANRAW. After carefully considering all the stages of the extraction process, we identify TOKENIZE as the only stage where vectorization provides a significant performance boost. Second, we design two scheduling strategies for assigning worker threads to tasks. Best-effort scheduling satisfies the requests in the order in which they are received by the scheduler—without considering additional data. Adaptive scheduling takes into consideration the state of the entire system when assigning worker threads. The goal is to optimize resource utilization in the system and minimize query execution time, while maximizing the amount of data loaded into the database. And finally, we consider alternative strategies for processing queries when the same data are stored both in the raw file, as well as inside the database. We design the merge read strategy which combines reading data from two sources optimally by grouping multiple requests corresponding to the same source and scheduling them together. In addition to formalizing the concepts introduced by each of the proposed contributions, we also present extensive experimental results that quantify their relevance across the overall SCANRAW architecture.

External tables. Modern database engines, e.g., Oracle and MySQL, 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 multicore 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 [Idreos et al. 2011]. 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 [Alagiannis et al. 2012]. 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 [Ivanova et al. 2012]. 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 [Abouzied et al. 2013]. 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 [Idreos et al. 2011], 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.

Instant loading [Mühlbauer et al. 2013]. 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. The extraction stages are still executed sequentially though, for every data partition—there is no pipeline parallelism. Moreover, instant loading does not support query processing over raw files. 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 that we integrate in SCANRAW. While the benefits of vectorization are evident when TOKENIZE is considered in isolation, the impact on the overall query execution is limited to the percentage tokenization represents from the total. As our detailed stage analysis shows, tokenization represents only a small fraction since parsing dominates the extraction time. Consequently, the impact vectorization has on in-situ raw file data processing is considerable only in specific scenarios, e.g., the file consists of a large number of text attributes that do not require parsing.

SDS/Q [Blanas et al. 2014]. SDS/Q executes queries directly over data stored in HDF5 files. Similar to NoDB, it never loads data into the database. Instead, it always reads data from the raw

file. However, since HDF5 is a binary storage format, in most cases, the parse and tokenize stages can be omitted. Moreover, SDS/Q builds external bitmap indexes to eliminate the requirement for scanning the entire file in order to reduce query response time. Compared to SDS/Q, SCANRAW can not only process queries directly from scientific raw files in binary format – HDF5 is only one such example – but can also execute queries from other file formats, including text files. Furthermore, due to the parallel pipeline for data conversion, SCANRAW can load data into the databases whenever necessary—and without paying any cost for it. This is not supported in SDS/Q, which is a distributed shared-memory data processing system without secondary storage functionality.

RAW [Karpathiotakis et al. 2014] & VIDa [Karpathiotakis et al. 2015]. The RAW system and its VIDa extension aim to query heterogeneous data sources transparently, without loading data into a database. RAW generates access paths just-in-time to adapt to the underlying data files and to the incoming queries. SCANRAW integrates external I/O libraries to access different data formats, e.g., BAM and FITS. Furthermore, SCANRAW measures the performance of the library dynamically in order to decide whether to load the data into database to improve the execution of subsequent queries.

Impala [M. Kornacker et al. 2015]. Impala is an open source massively parallel processing SQL query engine for data stored in a computer cluster running Hadoop[16]. Impala brings scalable parallel database technology to Hadoop, enabling users to issue low-latency SQL queries to data stored in HDFS and HBase[17] without requiring data movement or transformation. Impala applies task-parallelism to convert raw data into binary format for execution. When Impala executes a query from a raw file, it has to first retrieve the data from HDFS, the underlying file system. Therefore, the execution time for the first query cannot be fully controlled by Impala. Compared to Impala, SCANRAW applies super-scalar pipeline parallelism in order to maximize the hardware throughput.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we propose SCANRAW—a novel database meta-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 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. 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. Two scenarios will be considered. First, the query workload is known in advance. The question that has to be answered in this case is how to group the queries and in what order to execute them in order to achieve optimal processing time over the entire workload. Also, how can we take the workload into consideration when deciding what columns to load inside the database? In the second scenario, the workload is not know apriori. Queries are admitted dynamically at runtime. The objective remains minimizing the execution time over the entire workload. The existing SCANRAW operator represents a solid foundation in pursuing this type of work.

ACKNOWLEDGMENTS

The work in this article is supported by a U.S. Department of Energy Early Career Award (DOE Career). The authors would like to thank the anonymous reviewers for their valuable comments that improved the quality of this article.

[16] http://hadoop.apache.org
[17] http://hbase.apache.org/

REFERENCES

D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases* 5, 3 (2013), 197–280. DOI: http://dx.doi.org/10.1561/1900000024

A. Abouzied, D. Abadi, and A. Silberschatz. 2013. Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems. In *Proceedings of 2013 EDBT/ICDT Extended Database Technology Conference*. 1–10. DOI: http://dx.doi.org/10.1145/2452376.2452377

A. Ailamaki, V. Kantere, and D. Dash. 2010. Managing Scientific Data. *Commun. ACM* 53, 6 (2010), 68–78. DOI: http://dx.doi.org/10.1145/1743546.1743568

I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. 2012. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of 2012 ACM SIGMOD International Conference on Management of Data*. 241–252. DOI: http://dx.doi.org/10.1145/2213836.2213864

N. Alur, C. Takahashi, S. Toratani, and D. Vasconcelos. 2008. *IBM InfoSphere DataStage Data Flow and Job Design*. IBM Redbooks.

S. Arumugam, A. Dobra, C. Jermaine, N. Pansare, and L. Perez. 2010. The DataPath System: A Data-centric Analytic Processing Engine for Large Data Warehouses. In *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data*. 519–530. DOI: http://dx.doi.org/10.1145/1807167.1807224

R. Avnur and J. Hellerstein. 2000. Eddies: Continuously Adaptive Query Processing. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data*. 261–272. DOI: http://dx.doi.org/10.1145/342009.335420

D. Barnett, E. Garrison, A. Quinlan, M. Stromberg, and G. Marth. 2011. BamTools: a C++ API and Toolkit for Analyzing and Managing BAM Files. *Bioinformatics* 27, 12 (2011), 1691–1692. DOI: http://dx.doi.org/10.1093/bioinformatics/btr174

S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. 2014. Parallel Data Analysis Directly on Scientific File Formats. In *Proceedings of 2014 ACM SIGMOD International Conference on Management of Data*. 385–396. DOI: http://dx.doi.org/10.1145/2588555.2612185

R.D. Blumofe and C.E. Leiserson. 1999. Scheduling Multithreaded Computations by Work Stealing. 46, 5 (1999), 720–748. DOI: http://dx.doi.org/10.1145/324133.324234

Y. Cheng, C. Qin, and F. Rusu. 2012. GLADE: Big Data Analytics Made Easy. In *Proceedings of 2012 ACM SIGMOD International Conference on Management of Data*. 697–700. DOI: http://dx.doi.org/10.1145/2213836.2213936

Y. Cheng and F. Rusu. 2014a. Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID. *Distributed and Parallel Databases* (2014). DOI: http://dx.doi.org/10.1007/s10619-014-7149-7

Y. Cheng and F. Rusu. 2014b. Parallel In-situ Data Processing with Speculative Loading. In *Proceedings of 2014 ACM SIGMOD International Conference on Management of Data*. 1287–1298. DOI: http://dx.doi.org/10.1145/2588555.2593673

J. Dean and S. Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008), 107–113. DOI: http://dx.doi.org/10.1145/1327452.1327492

D. J. DeWitt and J. Gray. 1991. Parallel Database Systems: The Future of Database Processing or a Passing Fad? *SIGMOD Rec.* 19, 4 (1991), 104–112. DOI: http://dx.doi.org/10.1145/122058.122071

J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. 2005. Scientific Data Management in the Coming Decade. *SIGMOD Rec.* 34, 4 (2005), 34–41. DOI: http://dx.doi.org/10.1145/1107499.1107503

S. Idreos, I. Alagiannis, R. Johnson, and A. Ailamaki. 2011. Here are my Data Files. Here are my Queries. Where are my Results?. In *Proceedings of 2011 CIDR Conference on Innovative Database Research*. 57–68.

S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. 2012. MonetDB: Two Decades of Research in Column-Oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45.

S. Idreos, S. Manegold, H. Kuno, and G. Graefe. 2011. Merging What's Cracked, Cracking What's Merged: Adaptive Indexing in Main-Memory Column-Stores. *PVLDB* 4, 9 (2011), 586–597.

M. Ivanova, M. L. Kersten, and S. Manegold. 2012. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *Proceedings of 2012 SSDBM International Conference on Scientific and Statistical Database Management*. 485–494. DOI: http://dx.doi.org/10.1007/978-3-642-31235-9_32

M. Karpathiotakis, I. Alagiannis, T. Heinis, M. Branco, and A. Ailamaki. 2015. Just-In-Time Data Virtualization: Lightweight Data Management with ViDa. In *Proceedings of 2015 CIDR Conference on Innovative Database Research*.

M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. 2014. Adaptive Query Processing on RAW Data. *PVLDB* 7, 12 (2014), 1119–1130.

M. Kersten, S. Idreos, S. Manegold, and E. Liarou. 2011. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *PVLDB* 4, 12 (2011), 1474–1477.

H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. 2009. The Sequence Alignment/Map Format and SAMtools. *Bioinformatics* 25, 16 (2009), 2078–2079. DOI: http://dx.doi.org/10.1093/bioinformatics/btp352

K. Lorincz, K. Redwine, and J. Tov. 2003. Grep versus FlatSQL versus MySQL: Queries using UNIX Tools vs. a DBMS. (2003). Retrieved July 2014 from http://www.eecs.harvard.edu/~konrad/projects/flatsqlmysql/final_paper.pdf

M. Kornacker et al. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Proceedings of 2015 CIDR Conference on Innovative Database Research*.

T. Mühlbauer, W. Rödiger, R. Seilbeck, A. Reiser, A. Kemper, and T. Neumann. 2013. Instant Loading for Main Memory Databases. *PVLDB* 6, 14 (2013), 1702–1713.

D.A. Patterson, J.L. Hennessy, and D. Goldberg. 1996. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann.

V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. 2008. Constant-Time Query Processing. In *Proceedings of 2008 IEEE ICDE International Conference on Data Engineering*. 60–69. DOI: http://dx.doi.org/10.1109/ICDE.2008.4497414

D. Sanchez, D. Lo, R.M. Yoo, J. Sugerman, and C. Kozyrakis. 2011. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *Proceedings of 2011 PACT Conference on Parallel Architectures and Compilation Techniques*. 22–32. DOI: http://dx.doi.org/10.1109/PACT.2011.9

M. Stonebraker, J. Becla, D. J. DeWitt, K. T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik. 2009. Requirements for Science Data Bases and SciDB. In *Proceedings of 2009 CIDR Conference on Innovative Database Research*.

A. Witkowski, M. Colgan, A. Brumm, T. Cruanes, and H. Baer. 2011. Performant and Scalable Data Loading with Oracle Database 11g. (2011). Retrieved July 2014 from http://www.oracle.com/technetwork/testcontent/twpdwbestpractices-for-loading-11g-404400.pdf

Received ; revised ; accepted