UNIVERSITY OF CALIFORNIA, MERCED

In-situ Data Processing Over Raw File

by

Yu Cheng

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Electronic Engineering and Computer Science

Committee in charge:
Professor Florin Rusu, Chair
Professor Mukesh Singhal
Professor Alberto E. Cerpa
Professor Dong Li

Spring 2016

The dissertation of Yu Cheng is approved:

Florin Rusu, Chair	Date
Mukesh Singhal	Date
Alberto E. Cerpa	Date
Dong Li	Date

University of California, Merced

Acknowledgments

First of all, I would like to thank my advisor Professor Florin Rusu sincerely. Without his constant guidance over the years, this project would not have been possible. I was inspired and enlightened by his profound knowledge, acute insights, the thorough understanding of the philosophy of research, sense of humor and most importantly, meticulous attention to details.

I would like to thank my dissertation committee, Professor Mukesh Singhal, Professor Alberto E. Cerpa and Professor Dong Li from Electronic Engineering and Computer Science for their time, insightful comments and constructive guidance that significantly improved this dissertation.

I also would like to express my appreciation to my lab mates and colleagues at UC Merced. Chengjie Qin and Weijie Zhao offered me valuable advice when I began my research and life at Merced. Also the other lab mates Martin Torres, Xin Zhang, Zhiyi Huang, Abdur Rafay, Abhineet Dubey and Chi Zhang. I also want to thank my current colleagues Dr. Tao Ren, Dr. Ye Zhu, Dr. Zhijiang Ye and Dr. Hongyu Gao for all the help.

For the past few years back in China, I have been so lucky to work with great scholars both at WHUT and HUST. As my bachelor thesis advisor, Professor Wenbi Rao (WHUT) and Master thesis advisor, Professor ChangSheng Xie (HUST), they both inspired me with their sense of responsibility, rigorous scientific spirit; they led me to the journey of scientific research.

I also would like to thank the numerous fellowship awards from UC Merced, financial support, and training from being research and teaching assistant at UC Merced.

Last but not least I want to express my gratitude to my parents, wife, and son. All of them sacrificed a lot to support my research path with a sense of responsibility, a thorough understanding the meaning of hard work and an optimal attitude for appreciation of life. They are always a constant source of love and encouragement for me.

Curriculum Vitae

Education

- Ph.D. in Electronic Engineering and Computer Science. University of California, Merced (UCM). Merced, CA, USA. Aug. 2011-May 2016
- M.S. in Computer Engineering. Huazhong University of Science and Technology (HUST). Wuhan, Hubei, China. Sept. 2006-Mar. 2008
- B.S. in Computer Science. Wuhan Univ of Tech (WHUT). Wuhan, Hubei, China. Sept. 2001-July 2005

Publications

1. Yu Cheng, Weijie Zhao and Florin Rusu. "Bi-Level Online Aggregation on Raw Data", submitted.
2. Yu Cheng and Florin Rusu. "SCANRAW: A Database Meta-Operator for Parallel In-Situ Processing and Loading", ACM Transactions on Database Systems. Vol. 40, No. 3, October 2015.
3. Weijie Zhao, Yu Cheng, and Florin Rusu. "Vertical Partitioning for Query Processing over Raw Data", Proceedings of SSDBM 2015, San Diego, California, June 2015.
4. Yu Cheng and Florin Rusu. "Parallel In-Situ Data Processing with Speculative Loading", Proceedings of ACM SIGMOD 2014, Snowbird, Utah, June 2014, pp. 1287-1298.
5. Yu Cheng and Florin Rusu. "Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID", Distributed and Parallel Databases, May 2014.
6. Yu Cheng and Florin Rusu. "Astronomical Data Processing in EXTASCID", Proceedings of SSDBM 2013 (Demo Track), Baltimore, Maryland, July 2013, pp. 387-390.
7. Yu Cheng, Chengjie Qin, and Florin Rusu. "GLADE: Big Data Analytics Made Easy", Proceedings of ACM SIGMOD 2012 (Demo Track), Scottsdale, Arizona, May 2012. pp. 697-700.

Honors

Graduate Dean's Dissertation Fellowship, UC Merced	2016
SIGMOD student travel award.	2014
UC Mercd Graduate Summer Fellowship Award, UC Merced	2013

Contents

List of Figures

List of Tables

Abstract

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

In this paper, we analyze the shortcomings of the traditional database under different configuration and propose several novel solutions to overcome these problems. We firstly propose *SCANRAW*, an innovate 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 components to identify common functionality. We analyze alternative implementations and discuss possible optimization for each stage. Our primary contribution is a *parallel superscalar pipeline design* 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 in 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 achieve optimal performance. 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. More-over, *SCANRAW* maximizes resource utilization for the entire workload execution, while speculatively loading data, and without interfering with normal query processing.

Besides, incorporate query workload in raw data processing allows us to model raw data processing with partial loading as fully-replicated binary vertical partitioning. We model loading as binary vertical partitioning with full replication. We design a two-stage heuristic that combines the concepts of query coverage and attribute usage frequency. The heuristic comes within close range of the optimal solution in a fraction of the time. We extend the optimization formulation and the heuristic to a restricted type of pipeline raw data processing. The results confirm the superior performance of the proposed heuristic over related vertical partitioning algorithms and the accuracy of the formulation in capturing the execution details of a real operator.

Online aggregation (OLA) is an efficient method for data exploration that identifies uninteresting patterns faster by continuously estimating the result of a computation during the actual processing as long as the estimate is accurate enough to be deemed uninteresting, the system can stop the query immediately. However, building an efficient OLA system has a high upfront cost of randomly shuffling and loading the data. We then propose *OLA-RAW*, a novel system for in-situ processing over raw files that integrates data loading and online aggregation seamlessly while preserving their advantagesgenerating accurate estimates as early as possible and having zero time-to-

query. We design an accuracy-driven bi-level sampling process over raw files and define and analyze corresponding estimators. The samples are extracted and loaded adaptively in random order based on the current system resource utilization. We implement *OLA-RAW* starting from a state-of-the-art in-situ data processing system and evaluate its performance across a variety of datasets and file formats. Our results show that *OLA-RAW* maximizes resource utilization across a query workload and dynamically chooses the optimal sampling and loading plan that minimizes each query's execution time while guaranteeing the required accuracy. The result is a focused data exploration process that avoids unnecessary work and discards uninteresting data.

Chapter 1

Introduction

1.1 Background

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. The volume and variety of "Big Data" pose serious problems to traditional database systems. Data generated by modern scientific instruments is gigantic, besides the data is in multi-type formats. For example, in astronomy scientist usually use the advanced digital telescope to capture series of high-quality images from the sky and save them as files. In PTF project, between 2000 and 4000 science images are acquired nightly [73], it generates nearly $60GB$ of raw data per night on average, with peak volumes approaching $100GB$ on clear winter nights [30]. During this project, a total of $300TB$ of raw and processed image data will be stored over the five years [30]. In biology, a single experimental run of a modern gene sequencing machine will produce on the order of a terabyte of the sequence data. The scientists use SAM or BAM files to store the experiment result. How to handle petabytes of information in the different format is a challenging issue.

The database has to define the relational schema, load the data, only after that it can execute the queries on the dataset. Schema definition imposes a strict structure on the data, which is expected to remain stable. However, this is rarely the case for rapidly evolving datasets represented using key-value and other semi-structured data formats, e.g., JSON. Data loading is a schema-driven process in which data are duplicated in the internal database representation to allow for efficient processing. Even though storage is relatively cheap, generating and storing multiple copies of the same data can easily become a bottleneck for massive datasets. Moreover, it is quite often the case that most queries only access the small portion of the attributes.

Meanwhile, as more and more people move their data to the cloud, they are aware of the reason and importance of the argument "Why not to stop early?". Since cloud provider applies the Pay-as-you-go utility computing billing method, there is a real and observable cost associated with every CPU cycle consumed and bytes stored; the end-user would consider those costs to the workload management. According to reason, the user would be satisfied with the estimated result achieving 99% of the accuracy in 10% of the standard execution time. Online aggregation(OLA) is an efficient approach to reduce the response time of queries and present approximate results

generated with an error bound. If users satisfy the estimation, they can terminate the query immediately, which could dramatically reduce the execution time. As the computation progresses, the error bounds would narrow until bounds are zero-width after finishing the query.

1.1.1 Motivating examples

Genomic data processing To make our point, let us consider a representative example from genomics. SAM/BAM files[1] – SAM [49] are text, BAM [12] 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 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 typical 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 processing 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, however. 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 an SQL query.

Semi-structured data processing The Twitter API[6] provides access to several objects in JSON format through a well-defined interface. The schema of the objects is, however, not well-defined, since it includes "nullable" attributes and nested objects. The state-of-the-art RDBMS solution to process semi-structured JSON data [69] is to first load the objects as tuples in a BLOB column. Essentially, this entails complete data duplication, even though many of the object attributes are never used. The internal representation consists of a map of key-values that is serialized/deserialized into/from persistent storage. The user can directly run SQL queries on the map based on the keys, treated as virtual attributes. As an optimization, set columns - chosen by the user or by the system based on appearance frequency - are promoted to physical status. The decision on which columns to materialize is only a heuristic, quite often sub-optimal.

Astronomical data processing Another interesting example is Palomar Transient Factory[7] (PTF) project [55] that aims to identify and automatically classify transient astrophysical objects such as variable stars and supernovae in real-time. A list of potential transients – or candidates – is extracted from the images taken by the telescope during a night. They are stored as a table

[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/`

[6] https://dev.twitter.com/docs/platform-objects/

[7] `www.astro.caltech.edu/ptf/`

in one or more FITS[8] files. The initial stage in the identification process is to execute a series of aggregate queries over the batch of extracted candidates. This is the equivalent of data exploration. The general SQL form of the queries is:

```
SELECT AGGREGATE(expression) AS agg
FROM candidate
WHERE select-condition
HAVING agg < threshold
```

where *AGGREGATE* is SUM, COUNT or AVERAGE and *threshold* is a verification parameter. These queries check certain statistical properties of the entire batch and are executed in sequence—a query is executed only if all the previous queries are satisfied. If the candidate batch passes the verification criteria, an in-depth analysis is performed for individual candidates. The entire process – verification and in-depth analysis – is executed by querying a PostgreSQL[9] database—only after the candidates are loaded from the original FITS files. This workflow is highly inefficient for two reasons. First, the verification cannot start until data are loaded. Second, if the batch does not pass the verification, both the time spent for loading and the storage used for data replication are wasted.

1.2 Raw File Query Processing

Figure 1.1 depicts the generic process that has to be followed to make querying over raw files possible. The input to the process is a raw file – SAM/BAM in our running example – a schema, and a procedure to extract tuples with the given schema from the raw file. The output is a tuple representation that can be processed by the execution engine. For each stage, we introduce trade-o s involved and possible optimization. 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 1.1.

1.2.1 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 in the file or some external structure the entire file has to be read one line by another and passed to *EXTRACT*, at the first time it is accessed. As an optimization – already implemented by the file system – the system reads multiple lines co-located on the same page together. An additional optimization – also performed by the *file system* – is the caching of pages in memory buffers such that future requests for the same page can be served directly from memory without accessing the disk. Thus, while the disk throughput limits the first access, subsequent accesses can be much faster as long as data is in the cache.

Further reading optimization 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 we sort tuples on a particular

[8] http://heasarc.gsfc.nasa.gov/fitsio/

[9] http://www.postgresql.org/

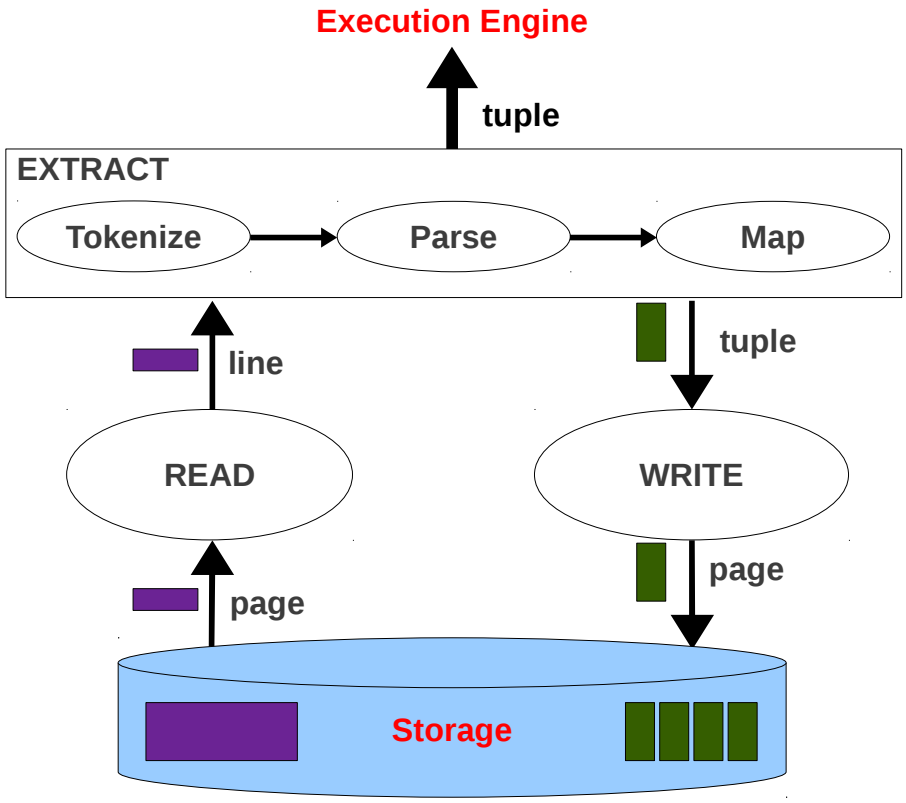


Figure 1.1: Query processing over raw files.

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 index built inside a database can also be applied to flat files by incurring the same or higher construction and maintenance costs. In the case of our genomic example, BAI files [12] are indexes built on top of BAM files. Columnar storage [2] and compression [60] are other strategies to minimize the amount of data read from disk—orthogonal to our discussion.

1.2.2 TOKENIZE

Abstractly, `EXTRACT` transforms a tuple from the 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. The `PARSE` receive the vector along with the text chunk as well. 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. We replace the delimiter 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 to the processing representation, i.e., selective tokenizing and parsing [38]. The idea is to stop the linear scan over the text as soon as identifying all the concerning attribute. 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 [7], from one conversion to another. Mostly, when `PARSE` receive the vector, 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.

1.2.3 PARSE

In `PARSE`, attributes are converted from the 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, whose 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 [38] 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 [7] 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.

1.2.4 MAP

The last stage of extraction is `MAP`. It takes the binary attributes converted in `PARSE` and organizes them into a data structure suitable for processing. In the case of a row-store execution engine, a record consists of many attributes. 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 correctly. Copying data around has to be avoided and replaced with memory mapping whenever possible.

At the end of `EXTRACT`, data are loaded into memory and ready for processing. At this point, there are multiple paths for the following work. In external tables, data are passed to the execution engine for query processing and discarded afterward. In NoDB [7] 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 is in the cache. In standard database loading, the system has to extract and written data to disk, and only after that, the system can receive and execute queries, which requires reading data again—from the database, though. It is important to notice that these stages have to be performed for any processing and any raw data, not only text. In the case of raw binary data though the bulk of processing is very likely to be concentrated in `MAP` instead of `TOKENIZE` and `PARSE`.

1.2.5 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 to minimize the interference. The typical sequential solution is to read a page, convert the tuples from text to binary, write them on a page, and then repeat the entire process for all the pages in the raw input file. This `READ-EXTRACT-WRITE` pattern guarantees non-overlapping access to the 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 optimization implemented in `TOKENIZE` and PARSE raises some complex trade-offs. If the database supports query-driven partial loading, the system 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 [39], which allow for efficient schema expansion by adding new columns. Loading new attributes reduce 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 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.

1.3 Database loading

Although Database Management Systems (DBMS) remain overall the effective data analysis technology, more and more users unwilling to choose it for emerging applications such as scientific analysis and social networks. This is due in large part to the complexity involved; there is a significant initialization cost in loading data and preparing the database system for queries. Take the same examples above; a scientist needs quickly to examine a few Terabytes of new information in search of specific properties. Even though only a few attributes might be relevant for the task, the entire data must first be loaded into the database. For significant amounts of data, this means a few hours of delay, even with parallel loading across multiple machines. Besides being a significant time investment, it is also important to consider the extra computing resources required for a full load and its side-effects concerning energy consumption and economic sustainability.

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 [7, 38]. Instead, the standard practice is to write dedicated applications encapsulating the query logic on top of *generic file access libraries* that provide instant access to data through a well-defined API. While a series of applications for a limited set of parametrized queries are provided with the library, new queries typically require the implementation of an entirely new application, even when the system can reuse significant logic. Relational databases avoid this problem altogether by implementing a declarative querying mechanism based on SQL, which requires data representation independence, though—achieved through loading and storing data in a proprietary format.

1.4 External tables

External tables [74] 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. Modern database engines, e.g., Oracle and MySQL, provide external tables as a feature to directly analyze 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. 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 go through conversion stages — 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 vast number of queries. The more queries are executed, the more likely is that the database outperforms external tables in response time.

Chapter 2

Problems and contributions

Figure2.1 depicts the comparison of existing solutions data loading and the external table. As we can see, although the query execution is fast from a database, the storage and data-to-query cost are high. The reason is that database has to spend a long time to extract the data from a raw file and load them as proper binary format, which doubles the data storage as well. External table could immediately run the query from a raw file – with zero time-to-query cost– by repeatedly extracting tuples from the raw file for every query. Besides, since all the binary tuples have been dropped after query execution, there is no additional storage cost as well. However, since tuples have to be converted from the raw format into the internal database representation for every query, the execution performance is both constant and poor. Therefore, at a high level, we aim to design and implement a solution that gets close to the optimal situation –with both less time-to-query and query time— as much as possible.

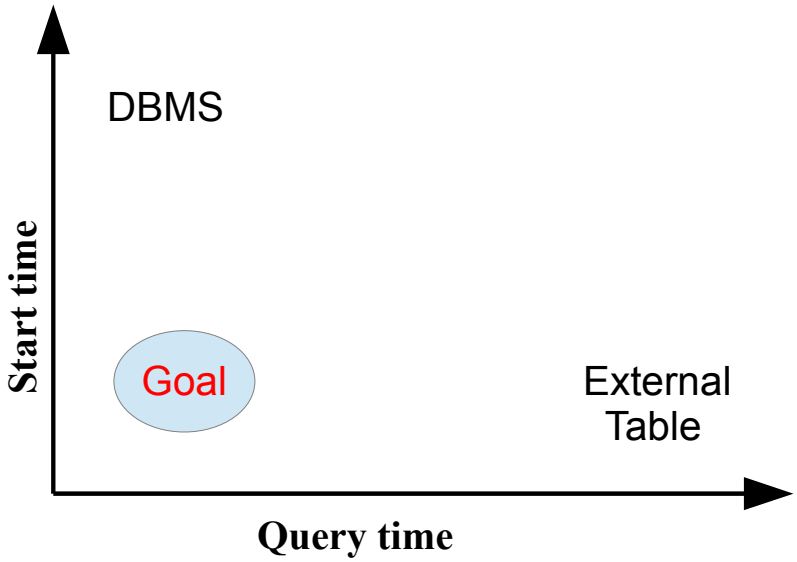


Figure 2.1: Data Loading vs External Table.

2.1 Problems & Approach

In this thesis, we describe the problem by comprehensively investigating, analyzing and summarizing in different aspects including three subproblems. Then, we put forward the possible solutions according to various problems and show the corresponding result.

The first problem we consider is to *execute SQL-like queries in-situ over raw files, e.g., SAM/BAM, with a database engine in a shared memory multi-core environment where I/O operations are overlapped with extraction and several chunks can be processed concurrently while minimizing resource utilization*. In this problem, data is converted to the database processing representation at query time and can be loaded into the database. Our approach is to devise a method that provides instant access to data and also achieves optimal performance when the workload consists of a sequence of queries. We design and implement a scan operator called `SCANRAW` that provides single-query optimal execution over raw files. This implementation can be applied both to external table processing as well as standard data loading. And second, we propose a mechanism for query-driven gradual data loading. After executing sufficient queries, 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. Figure 2.2 shows the main idea about `SCANRAW` that we detect the useful and necessary data and distributes data loading across the query workload.

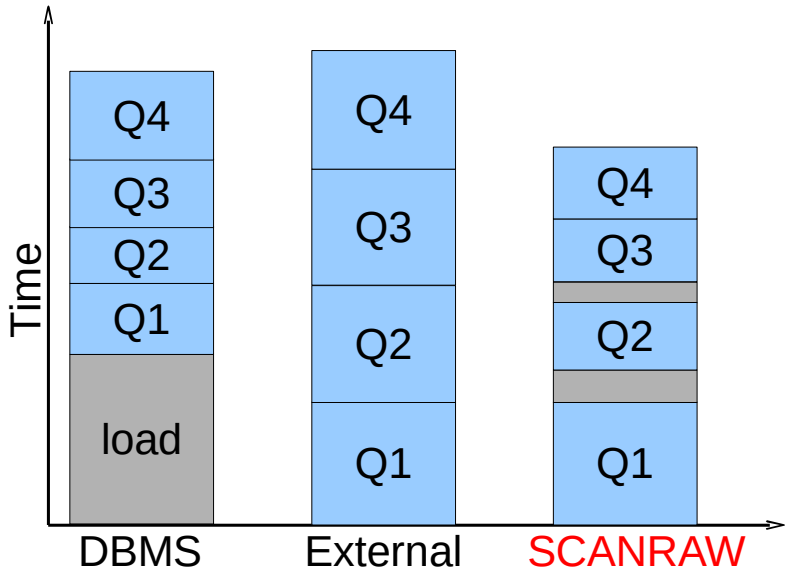


Figure 2.2: Intuition of Scanraw.

The second problem we investigate is *Given a dataset in some raw format, a query workload, and a limited database storage budget, find what data to load in the database such that the overall workload execution time is minimized.* Consider a relational schema $R(A_1, A_2, \ldots, A_n)$ and an instantiation of it that contains $|R|$ tuples. Semi-structured JSON data can be mapped to the relational model by linearizing nested constructs [69]. To execute queries over R, tuples have to be read in

memory and converted from the storage format into the processing representation. Two timing components correspond to this process. T_{RAW} is the time to read data from storage into memory. T_{RAW} can be computed straightforwardly for a given schema and storage bandwidth $band_{IO}$. A constraint specific to raw file processing – and row-store databases, for that matter – is that all the attributes are read in a query—even when not required. T_{CPU} is the second timing component. It corresponds to the conversion time. For every attribute A_j in the scheme, the conversion is characterized by two parameters, defined at tuple level. The *tokenizing time* T_{t_j} is the time to locate the attribute in a tuple in the storage format. The *parsing time* T_{p_j} is the time to convert the attribute from storage format into processing representation. A limited amount of storage B is available for storing data converted into the processing representation. This eliminates the conversion and replaces it with an I/O process that operates at column level—only complete columns can be saved in the processing format. The time to read an attribute A_j in processing representation, T_j^{IO}, can be determined when the type of the attribute and $|R|$ are known.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
Q_1	X	X						
Q_2	X	X	X	X				
Q_3			X	X	X			
Q_4		X		X		X		
Q_5	X		X	X	X		X	
Q_6	X	X	X	X	X	X	X	

Table 2.1: Query access pattern to raw data attributes.

Consider a workload $W = \{Q_1, Q_2, \ldots, Q_m\}$ of m SQL-like queries executed over the schema R. The workload can be extracted from historical queries, or an expert user can define it. Each query Q_i is characterized by $\{A_{j_1}, A_{j_2}, \ldots, A_{j_{|Q_i|}}\}$, a subset of attributes accessed by the query. Queries are assumed to be distinct, i.e., there are no two queries that access the same set of attributes. A weight w_i characterizing importance, e.g., frequency in the workload, is assigned to every query. Ideally, $\sum_i w_i = 1$, but this is not necessary. The problem we aim to solve is *how to use the storage B optimally such that the overall query workload execution time is minimized?* Essentially, what attributes to save in processing representation to minimize raw file query processing time? We name this problem *raw data processing with partial loading*. We study two versions of the problem—serial and pipeline. In the serial problem, the I/O and the conversion are executed sequentially, while in the pipeline problem, they can overlap. Similar to offline physical database design [17], the conversion of the attributes stored in processing representation is executed before the workload execution. Table 2.1 depicts the access pattern of a workload of 6 queries to the eight attributes in a raw file. X corresponds to the attribute being accessed in the respective query. For example, Q_1 can be represented as $Q_1 = \{A_1, A_2\}$. For simplicity, assume that the weights are identical across queries, i.e., $w_i = 1/6$, $1 \leq i \leq 6$. If the amount of storage B that can be used for loading data into the processing representation allows for at most three attributes to be loaded, i.e., $B = 3$, the problem we address in this paper is what 3 attributes to load such that the workload execution time is minimized? Since A_8 is not referenced in any of the queries, we are certain that A_8 is not one of

the attributes to be loaded. Finding the best three out of the remaining seven is considerably more challenging.

The last problem we aim to solve is to *optimally execute exploration over raw data in a shared memory multi-core environment where I/O operations are overlapped with extraction, and several chunks can be processed concurrently while minimizing resource utilization*. We have to integrate online aggregation seamlessly into raw data processing such that we cumulate their benefits. Figure 2.3 illustrates the intuition behind the proposed OLA-RAW solution concerning standard database processing (DBMS), online aggregation (OLA), and raw data processing with on demand loading (RAW), respectively. Similar to RAW, OLA-RAW distributes data loading across the query workload. The same idea is extended to shuffling. Instead of randomly permuting all the data before performing online aggregation, OLA-RAW distributes shuffling across the queries in the workload. Moreover, loading and shuffling are combined incrementally such that loaded data do not require further shuffling. *OLA-RAW provides a resource-aware parallel mechanism to extract on demand and incrementally maintain samples from raw data*. The end goal is to reduce the high upfront cost of loading (DBMS) and shuffling (OLA) and to minimize the amount of data accessed by RAW, as long as the user accepts estimates.

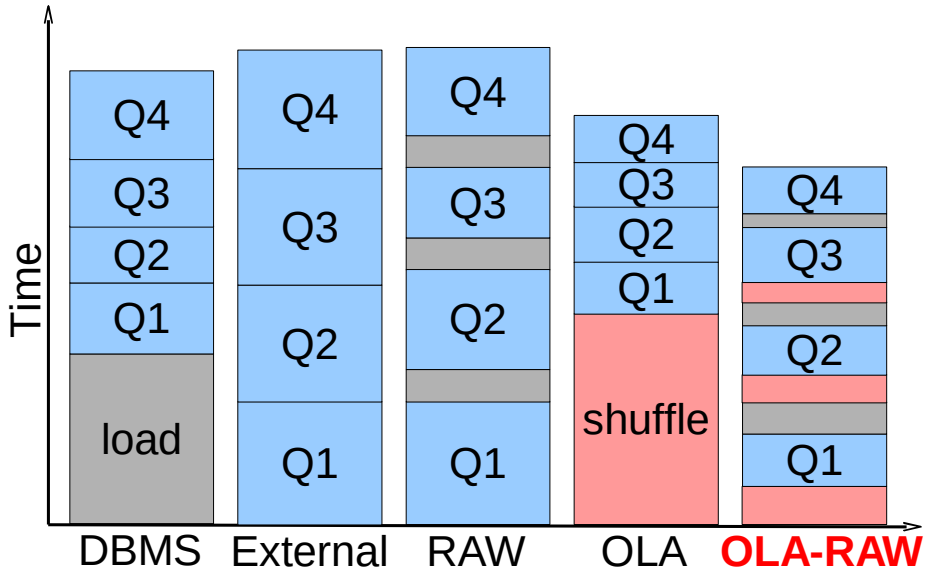


Figure 2.3: Intuition of OLA-RAW.

2.2 Challenges

There are a series of difficult challenges to different subproblems. For the first problem, to get the optimal execution time for the first query over a raw file, the scan operator –*SCANRAW*– choose to combine all possible parallel strategies —such as data partition, multi-threading and pipeline— together to speed up the raw data processing procedure. However, at the same time, we have to

handle all the possible common problems due to the parallel environments, like workload unbalancing, pipeline efficiency, etc. Furthermore, for the query-driven gradual data loading mechanism, it should interfere minimally – if at all – with standard query processing and guarantees that converted data are loaded into the database for every query. The mechanism should find efficient methods to detect and distribute the necessary data loading across the queries workload by maximizing the system utilization all the time.

The second problem is a standard database optimization problem with bounded constraints, similar to vertical partitioning in physical database design [47]. However, while physical design investigates what non-overlapping partitions to build over internal database data, we focus on what data to load, i.e., replicate, in a columnar database with support for multiple storage formats. A viable model should be devised to describe raw data processing with partial loading. The most important point is that a practical solution to this problem has to be designed with extensive experiment. We need to consider two versions of the problem—serial and pipeline. In the following problem, the I/O and the conversion are executed sequentially, while in the pipeline problem, they can overlap. Similar to offline physical database design [17], the transformation of the attributes stored in processing representation is executed before the workload execution. We model loading as binary vertical partitioning with full replication. Based on this equivalence, we provide a linear mixed integer programming optimization formulation that we prove to be NP-hard and inapproximable.

The realization of OLA-RAW poses a series of difficult challenges. First and foremost, an efficient sampling mechanism targeted at raw data has to be devised. The sampling mechanism has to work in-place, over data in the original format. It has to minimize the amount of raw data read and extracted into the processing representation since these are the fundamental limitations of raw data processing. Given our focus on parallel processing, the sampling mechanism has to cope with the so-called "inspection paradox" [57]. Since the estimate correlates with the extraction time, the order to consider chunks has to be the same with the extraction scheduling order. The second challenge is defining and analyzing estimators for the sampling mechanism. To be amenable to online aggregation, the estimators have to be integrated with the sampling method, and they have to support incremental computation over samples of increasing size. A third challenge corresponds to the incremental maintenance of samples. Since extracting samples from raw data is expensive, a mechanism that preserves them for further use in subsequent queries and expands them incrementally is necessary. This has to be realized by using only the spare resources available during processing—the goal is to compute the estimate as fast as possible, not to maintain the sample. From an implementation perspective, the integration of online aggregation into a resource-aware raw data processing system is always challenging because of the complex interactions between I/O, extraction, loading, sampling, and estimation.

2.3 Contibutions

The major contributions we propose in this thesis aims at solving the problems mentioned above, which includes three segments including two scan operators –*SCANRAW* and *OLA-RAW*– and a serial of loading strategies –*speculative loading*, *multistep loading* and *raw data processing with partial loading*. Our specific contributions can be summarized as follows:

- Design SCANRAW, the first parallel super-scalar pipeline operator for in-situ processing over raw data. It is a novel database physical operator for in-situ processing over raw files that integrates data loading and external tables seamlessly while preserving their advantages—optimal performance across a query workload and zero time-to-query. SCANRAW has a *parallel super-scalar pipeline architecture* that overlaps data reading, conversion into the database representation, and query processing. Implement SCANRAW in the DataPath [9] multi-threaded database system.
- Design OLA-RAW—a novel system which supports OLA functionality and in-situ processing over raw files– that execute the query in an efficient way and get optimal performance across a query workload. OLA-RAW supports not only standard raw file process executing complex analytical queries over terabytes of raw data very efficiently, but also it supports *overlap query processing with estimation*. Unlike the traditional OLA system which assumes that data is generated in a "random order", OLA-RAW begins with raw data files and has to design an efficient mechanism to produce tuples in a random order by itself. Another aspect that differentiates OLA-RAW from other online aggregation systems is OLA-RAW is a query-driven operator that could dynamically change its I/O plan into an efficient way to access the raw file. Finally, it aims to speed up the traditional raw data processing not only for the first query but also for a sequence of queries by selectively preserving processed samples in a smart way.
- Propose *speculative loading* as a gradual data loading mechanism and successfully implement it in *SCANRAW* that takes advantage dynamically of the disk idle intervals arising during data conversion and query processing. The main idea of 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.
- Propose *multistep loading* as a supplemental gradual data loading mechanism for speculative loading and successfully implement it in *SCANRAW*. The main idea in *multistep loading* is to find the computation bottleneck and try to postpone unnecessary workload 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.
- Propose a practical solution for raw data processing with partial loading problem. Take query workload into account, and we successfully model raw data processing with partial loading as fully-replicated binary vertical partitioning. We provide a linear mixed integer programming optimization formulation that we prove to be NP-hard and inapproximable. We design a two-stage heuristic that combines the concepts of query coverage and attribute usage frequency. The heuristic comes within close range of the optimal solution in a fraction of the time. We extend the optimization formulation and the heuristic to a restricted type of pipelined raw data processing. In the pipelined scenario, data access and extraction are executed concurrently. We evaluate the performance of the heuristic and the accuracy of the optimization formulation over three real data formats – CSV, FITS, and JSON – processed with a state-of-the-art pipelined operator for raw data processing.
- Design and implement serial comprehensive experiments to evaluate all solutions performance across a variety of synthetic and real-world datasets. Analyze the experiments results,

describe the work behavior and explain the reasons.

2.4 Conclusion

Combining the bunch of proposed solutions described above, we efficiently speed up the raw data processing in a different work environment. The figure 2.4 shows the summary of our contributes. The *SCANRAW*—with super-scaler pipeline architecture—integrates *speculative loading and multistep loading*, preserving the advantages from database loading(DBMS) and external table(External)—optimal performance across a query workload and zero time-to-query(Start time). Since it detects and loading only useful and necessary data, the storage for loading is on demand which could minimize the storage cost. Furthermore, *OLA-RAW* successfully inherit all the advan-

	Start time	Execution	Storage
DBMS	loading	fast	twice
External	instant	slow	one
Scanraw	instant	fast	on demand
OLA	shuffling	faster	twice
OLA-RAW	instant	faster	on demand

Figure 2.4: Summary of contributions.

tages from *SCANRAW* and *OLA* and eliminate their drawbacks. The result shows that *OLA-RAW* find an efficient way to generate samples and produce estimation with error bound, which aims to terminate the query execution as fast as possible. Besides, it maintains the samples to speed up the following queries but only when necessary. The storage cost is low, and query performance is high.

We conclude with a detailed look at related work (Chapter 3). In chapter 4 we describe the implementation of *SCANRAW* in details including architecture, operation and scheduling problems. Later, we propose a serial of loading strategies –speculative loading and multistep loading is presented in charpter 5. Raw data process with partial loading problem is discussed in chapter 6. *OLA-RAW* is depicted in chapter7. We conclude the work finally (Chapter 8) and plans for future work.

Chapter 3

Related Work

Several researchers [6,29,48,66] have recently identified the need to reduce the analysis time for processing tasks operating over massive repositories of raw data. In-situ processing [50] 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 [8,74] and various optimizations that eliminate the requirement for scanning the entire file to answer the query [7,38, 42]. The second category is organized around the MapReduce programming paradigm [25] 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[1], the application is still responsible for a significant part of the conversion, i.e., the Map and Reduce functions contain large amounts of tokenizing and parsing code. The work in this category focuses on eliminating the conversion code by caching the already converted data in memory or storing it in binary format inside a database [3].

SCANRAW [20, 21] The SCANRAW meta-operator is introduced in [20]. 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 opti-

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

mize 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 multi-core processors for improving performance significantly when mapping data into the processing representation is expensive. As far as we know, SCANRAW is the first parallel pipelined solution for external tables in the literature. Moreover, SCANRAW goes well beyond the external tables functionality and supports speculative loading, tokenizing, and parsing.

Adaptive partial loading [38] 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 the highest priority. Data are loaded only if sufficient I/O bandwidth is available.

NoDB [7] 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 eliminates tokenizing. The positional map is built incrementally, from the executed queries. Moreover, only the attributes required for 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 [42] 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 standard query language is the main contribution brought by data vaults.

Invisible loading [3] 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 [38], 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 on 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 [53] 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 loading. Overall, instant loading introduces faster algorithms for tokenizing that we integrate with 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 vast number of text attributes that do not require parsing.

SDS/Q [15] 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 to reduce query response time. Compared to SDS/Q, SCANRAW can not only process queries directly from raw scientific data 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 [47] & VIDa [46] 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 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 to decide whether to load the data into the database to improve the execution of subsequent queries.

Impala [51] Impala is an open source massively parallel processing SQL query engine for data stored in a computer cluster running Hadoop[2]. Impala brings scalable parallel database technology to Hadoop, enabling users to issue low-latency SQL queries to data stored in HDFS and HBase[3] without requiring data movement or transformation. Impala applies task-parallelism to convert raw data into the binary format for execution. When Impala executes a query from a raw file, it has first to retrieve the data from HDFS, the underlying file system. Therefore, the execution time for the first query cannot be entirely controlled by Impala. Compared to Impala, SCANRAW applies super-scalar pipeline parallelism to maximize the hardware throughput.

[2] http://hadoop.apache.org
[3] http://hbase.apache.org/

Chapter 4

SCANRAW – parallel data processing operator

4.1 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 [26]: 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.

4.1.1 Data Parallelism

Data parallelism is a form of parallelization of 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.

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 [26] 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 4.1 depicts the generic structure of an internal chunk containing metadata to support range-based data partitioning. The metadata includes the minimum and maximum values for each attribute and is 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

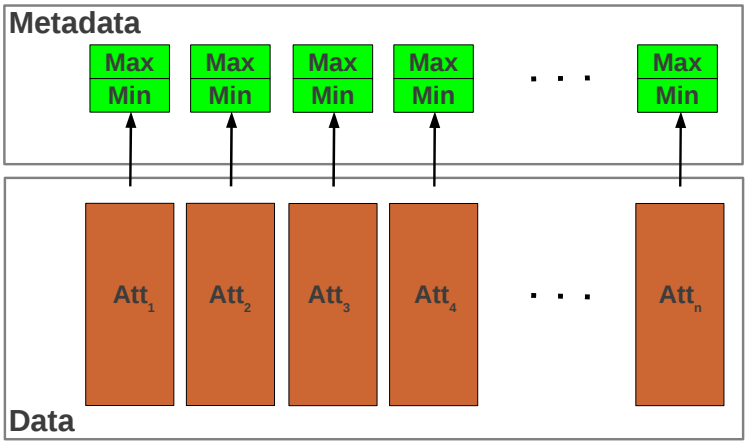


Figure 4.1: Chunk structure for internal processing.

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 sets, 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[1], 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`[2] and `LLVM`[3], detect vectorization opportunities automatically whenever 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.

Vectorization can be used to speed-up the tokenization process, as shown in [53]. The `SSE 4.2` instruction set works on 128-bit registers and contains instructions for the comparison of two 16

[1] `https://software.intel.com/en-us/articles/extending-the-worlds-most-popular-processor-architecture`
[2] `https://gcc.gnu.org/`
[3] `http://www.llvm.org/`

Algorithm 1 Delimiters Find

Require: Source string *input*; Delimiters SC
Ensure: Index of next delimiter in *input*

```
1:  searchIdx = 0;
2:  mode = _SIDD_CMP_EQUAL_ANY;
3:  __m128i data;
4:  __m128i pattern = _mm_set_epi8(SC);
5:  while !IsEnd(input) do
6:      data = _mm_loadu_si128(input);
7:      searchIdx = _mm_cmpistri(pattern, data, mode);
8:      if (searchIdx < 16) then
9:          //processingdelimiter
10:     end if
11:     input = input + 16;
12: end while
```

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.

4.1.2 Task Parallelism

Task parallelism[4] – 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 1.1 – `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.

4.1.3 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

[4] `http://en.wikipedia.org/wiki/Task\_parallelism`

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 $T_{no\text{-}pipeline} = \sum_{i=1}^{n} t_i$ and $1/T_{no\text{-}pipeline}$, 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, 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:

$$\eta = \frac{L \cdot T_{no\text{-}pipeline}}{T_{no\text{-}pipeline} + (L-1) \cdot t_m} \tag{4.1}$$

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 1.1, 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.

In this section, we consider *single query execution* over raw data. Given a set of raw files and an 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 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 1.2 and applying the parallelization techniques discussed in Section 4.1. Our primary contribution is a *parallel super-scalar pipeline architecture* [59] 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 [26]—also implemented in `SCANRAW`. Some solutions follow the principle `READ-EXTRACT-PROCESS`, e.g, external tables and NoDB [7, 38], while others [3] 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.

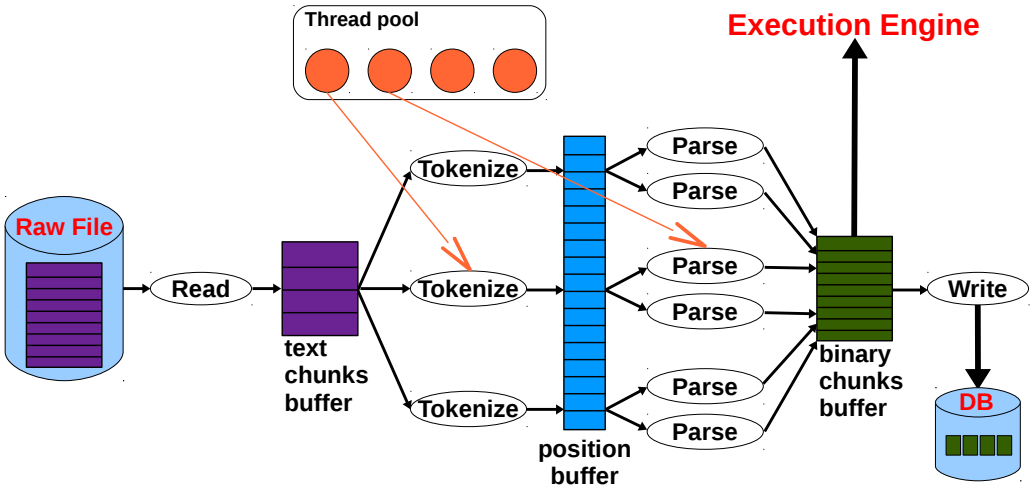


Figure 4.2: SCANRAW architecture.

4.2 Architecture

The super-scalar pipeline architecture of SCANRAW is depicted in Figure 4.2. Although based on the abstract process representation given in Figure 1.1, 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 [26]. MAP is not an independent stage anymore. 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. The scheduler thread also controls READ and WRITE 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 behavior—*speculative loading* (Section 5.1). 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 [10] 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 data is logically split into horizontal portions comprising a sequence of lines, i.e., chunks. Chunks represent the reading and processing unit. The *position buffer* between TOKENIZE and PARSE contains the text chunks read from the file and their corresponding positional map computed in TOKENIZE. Finally, *binary chunks buffer* contains the binary representation of the chunks. This is the processing representation used in the execution engine as well as the format in which data are stored in 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 unique 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 are in the cache, SCANRAW only 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 into 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. 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 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 records the general information for chunk ID but also remembers the critical contend of 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 the database with other columns that reside in raw files.

4.3 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 4.2. The stand-alone threads – depicted in

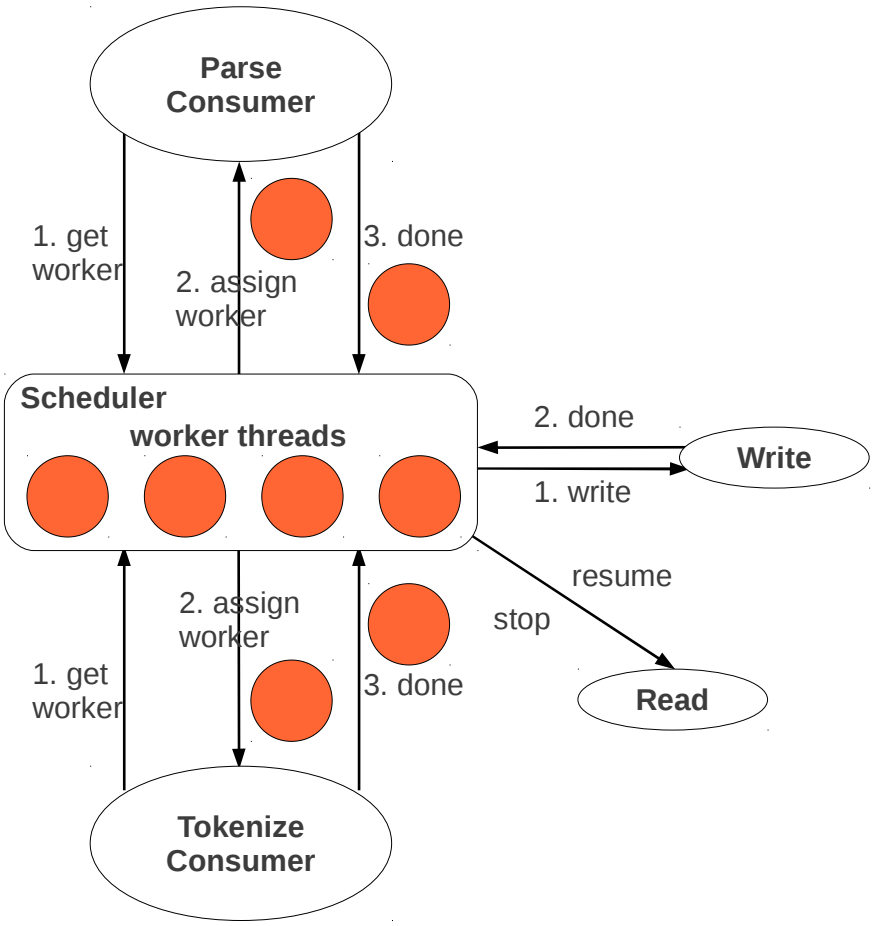


Figure 4.3: SCANRAW threads and control messages.

Figure 4.3 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 4.3 – that are configured dynamically with the task to execute.

4.3.0.1 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 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, a sequential scan is the only alternative. If the data was read before, a series of optimizations could 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. Mainly, 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 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 in 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. These triggers `READ` to produce a new chunk if the buffer is not full. When the processing finishes, 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 enough space in the destination buffer. Otherwise, the chunk cannot move forward. For example, a request from the `PARSE` consumer can be accomplished if there is space 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 4.4, we discuss in detail the strategies used to schedule worker threads in `SCANRAW`.

4.3.0.2 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 4.2, 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 1.2. The scheduler selects the task to assign to each worker from a set of requests made by the corresponding consumer threads.

4.4 Worker Thread Scheduling

`SCANRAW` can be viewed as a parallelism pipeline 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. Workload variation across stages usually causes load imbalance. 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 schedule 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 of particular importance 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 [16, 64], their application to a database operator for raw file processing is new. 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 7.6.

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 jobs are viewed equal to each other, so when two different tasks are ready and ask for resources, the scheduler just 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. How-

ever, 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 preceding 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 the 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 system runtime state, which includes the available resource status and statistics on current and past workloads. The resource status involves the number of available worker threads and the used memory capacity. The running time – the primary 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 most time-consuming stage determines the pipeline performance. We use function $f'(i) = \frac{t_i \cdot n_i}{N}$ to express the expected running time for stage i. The stage with 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:

$$\begin{aligned} &\textit{minimize} && \max_{1 \le i \le k} f'(i) - \min_{1 \le j \le k} f'(j) \\ &\textit{subject to} && \sum_{i=1}^{k} m_i \le M \end{aligned} \tag{4.2}$$

Adaptive scheduling not only guarantees optimal system utilization but can also adapt automatically to workload imbalance. To solve the imbalanced workload problem, the execution time for all the stages has to change dynamically in a short time interval. To detect this situation, when the average execution time is calculated, we assign different weights based on previous stage performances. The degree of responsiveness to the workload can be controlled by the assignment of weight values. If we consider more on recent executions, the faster the adaptation to changes. Op-

posite, if we rely more on past performances, 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:

$$\mathrm{T}_i = \sum_{i=1}^{l} w_i \cdot t_i \tag{4.3}$$

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.

4.5 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. About the process depicted in Figure 1.1, 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 any query, but rather to the raw file which contains data. As such, the state of the internal buffers is preserved across queries 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? After the entire raw file has been loaded in 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, architectural modifications are required. For example, raw binary files such as BAM do not need TOKENIZE and PARSE becomes exclusively MAP, transforming data from the format returned by the file access library to the internal processing format. In this case, the corresponding buffer and the consumer thread monitoring it are removed from the pipeline altogether. It is this generalization to any 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 significant benefit of SCANRAW when compared to file access libraries which require writing an entirely new program for every query.

Query optimization To efficiently 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 specified time intervals. In the case of SCANRAW, statistical data are collected while data are converted to 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 also be 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 before loading, SCANRAW does not address the problem of thoroughly sorting and reorganizing data based on queries, i.e., database cracking [41]. 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 to control the operation better 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 into the resource allocation procedure. The scheduler is in the best position to monitor the use of resources since it manages the distribution 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 to cope with the I/O throughput. For memory utilization, it makes sense to allocate more memory to SCANRAW instances that are used more often since this increases the rate of data reuse across queries.

4.6 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 on 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?
- Where is the time spent in the pipeline? How does the dynamic SCANRAW architecture adapt to data characteristics? Query properties?
- How fast can SCANRAW extract tuples from the raw file?
- How does the scheduling algorithm affect query execution performance?

Implementation SCANRAW is implemented as a C++ prototype. Each stand-alone thread, as well as the workers, are realized 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 underlying architecture. We integrate SCANRAW with a state-of-the-art multi-thread database system [9, 18, 19] shown to be I/O-bound for a large class of queries. This integration 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 three times and report the average value as the result. If the experiment consists of a single query, we always enforce to read data from the 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.

4.6.1 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 [3, 7]. 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.

4.6.1.1 Parallelism and speedup

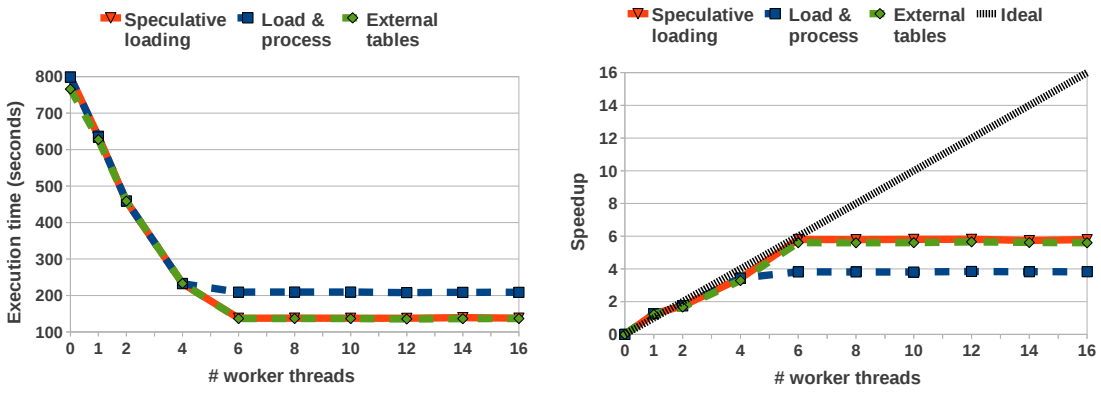
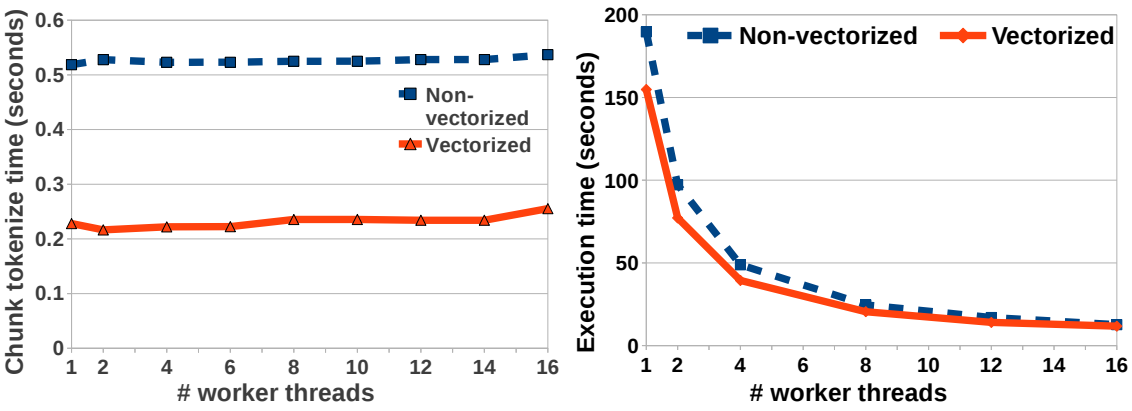


Figure 4.4: Execution time (a) and speedup (b) as a function of the number of worker threads.

Figure 4.4 depicts the effect the number of workers in the thread pool has on the execution of speculative loading, query-driven loading, and execution – load all data into the database only when queried – and external tables. Notice that all these three regimes are directly supported in `SCANRAW` with simple modifications to the scheduler writing policy. Zero worker threads correspond to sequential execution, i.e., the chunks go through the conversion stages one at a time. With one or more worker threads, `READ` and `WRITE` are separated from conversion—`TOKENIZE` and `PARSE`. Moreover, their execution is overlapped. While the general trend is standard – increasing the degree of parallelism results in better performance – many findings require clarification. The execution time (Figure 4.4a) – and the speedup (Figure 4.4b) – level-off beyond 6 workers. The reason for this is that processing becomes I/O-bound. Increasing the number of worker threads does not improve performance anymore. As expected, loading all data during query processing increases the execution time. What is not expected though is that this is not the case when the number of worker threads is 1, 2, and 4. In these cases, full loading, speculative loading, and external tables have identical execution time. The reason for this behavior is that processing is CPU-bound and – due to parallelism – `SCANRAW` manages to overlap conversion to binary and loading into the database completely. Essentially, loading comes for free since the disk is idle. In Figure 4.4a, the curves for external tables and speculative loading are always overlapped for more than one thread. Independent of the number of workers, `SCANRAW` minimizes query execution time. All the unique `SCANRAW` features – super-scalar pipeline, asynchronous threads, dynamic scheduling – combine to make loading and processing as efficient as external tables. We are not aware of any other raw file processing operator capable of achieving this performance.



(a) (b)

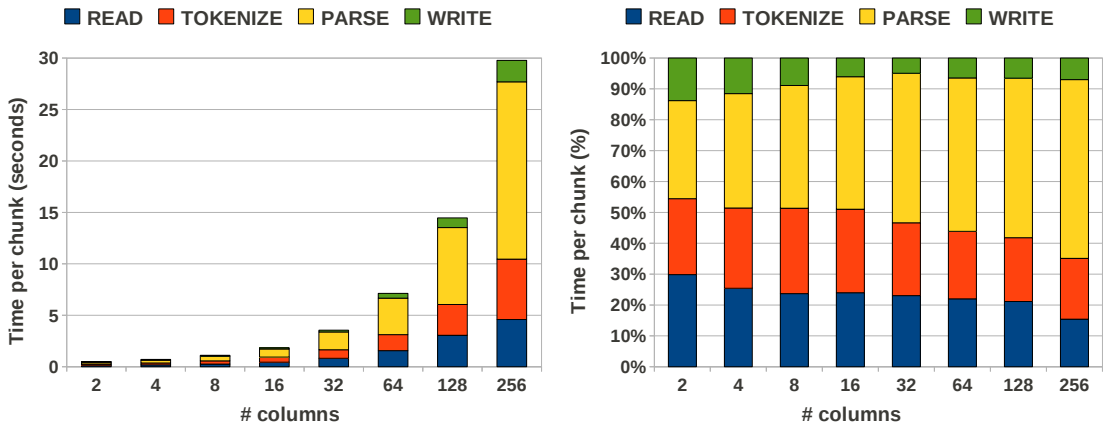
Figure 4.5: The effect vectorization has on tokenization (a) and overall query execution time (b).

4.6.1.2 Vectorization

Figure 4.5a 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 4.5b 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 of the overall query execution time, as proved by the detailed pipeline analysis presented in the following.

4.6.1.3 Pipeline analysis

Figure 4.6 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. 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 4.6a. 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



(a) (b)

Figure 4.6: Pipeline execution time: (a) absolute, (b) relative.

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 4.4a) over raw files, thus making data loading obsolete. Figure 4.6b gives the relative distribution of the data in Figure 4.6a. 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 to make raw file processing efficiently. 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.

4.6.1.4 Position and number of columns

Figure 4.7 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.

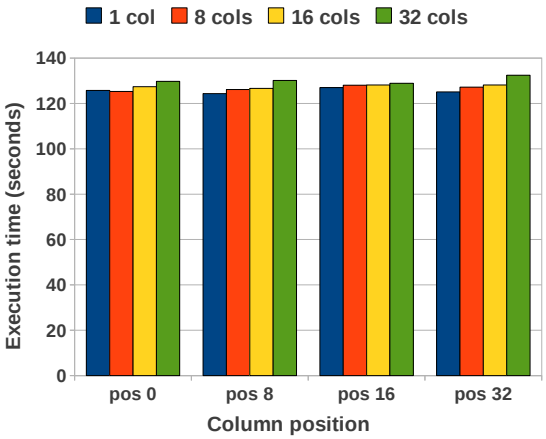


Figure 4.7: Position and number of columns.

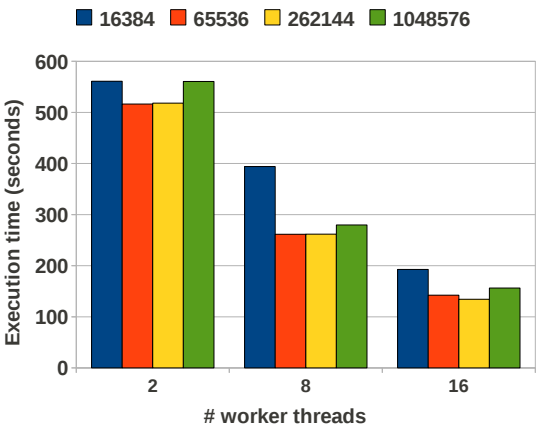


Figure 4.8: Chunk size.

4.6.1.5 Chunk size

The chunk size – the number of lines in the file processed as a unit – has a dramatic impact on the pipeline efficiency—depicted in Figure 4.8. 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 the performance densely. 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 $2^{19} \approx 500K$.

4.6.1.6 Thread scheduling

In this experiment, we compare the performance of the two scheduling algorithms introduced in Section 4.4—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 4.9. In Figure 4.9a, on the left, the vertical axis is the execution time, while in Figure 4.9b, 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 optimizing 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

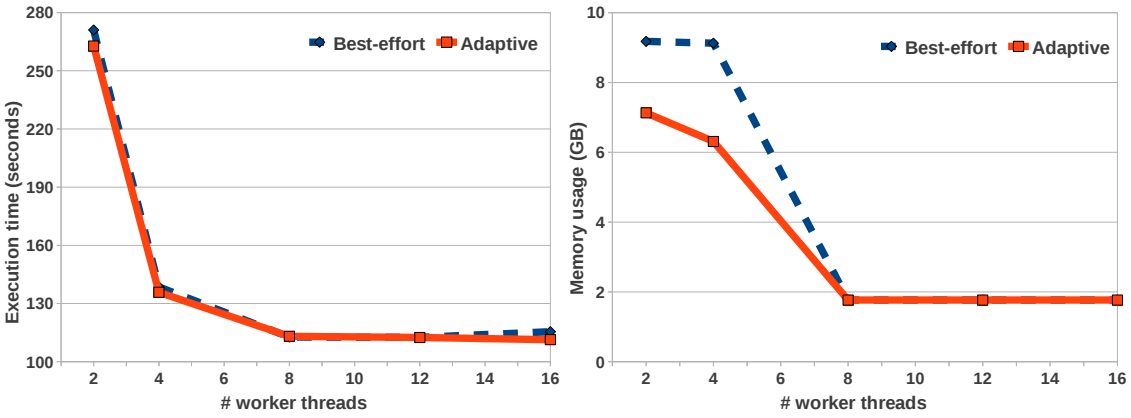


Figure 4.9: Thread scheduling algorithm comparison: (a) execution time, (b) memory usage.

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.

4.7 Conclusion

The SCANRAW meta-operator is a super-scalar pipeline architecture which is designed following a detailed analysis of the conversion process from the raw file representation to the processing format. In this work, we bring two 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.

Chapter 5

Speculative loading for query acceleration

5.1 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.

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 of 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 of detecting free I/O bandwidth and overlap processing with disk operations is required to implement speculative loading. SCANRAW accomplishes these requirements.

There are several solutions in the literature that are related to speculative loading. In invisible loading [3], a fixed amount of data – specified as some 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 [7] achieves optimal execution time for all the queries in a workload only when all the accessed data fit in memory. NoDB only caches the data in the memory without considering any loading. A possible extension to NoDB – explored in [38] – is to flush

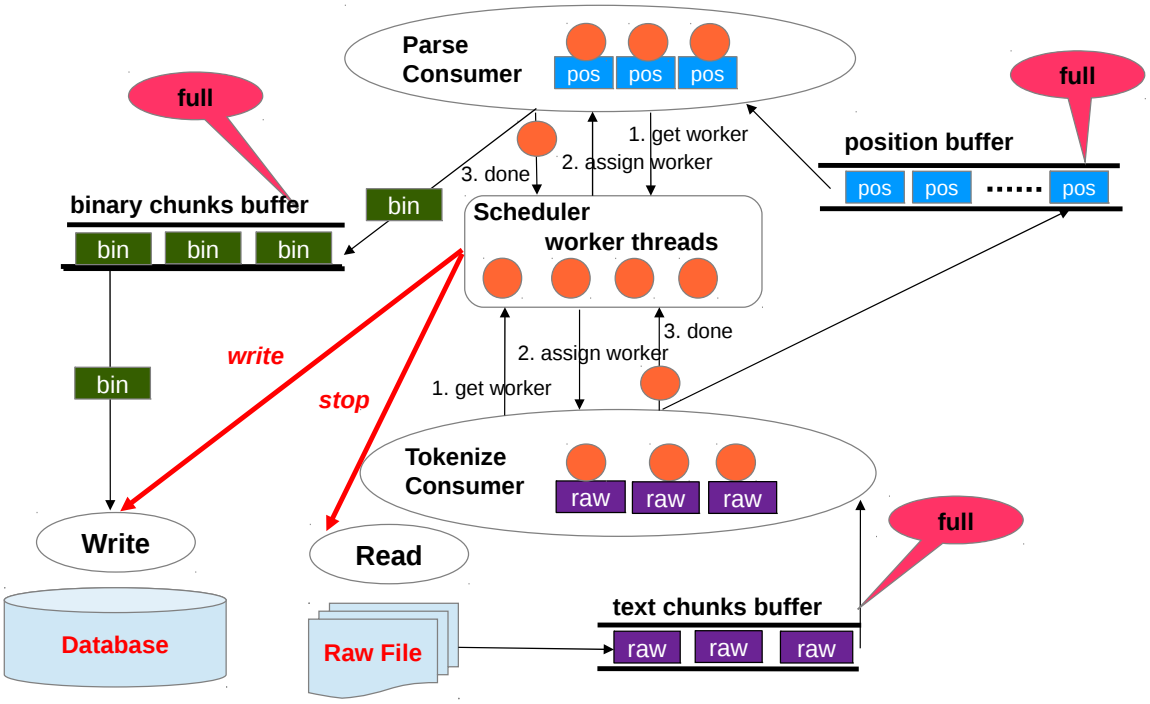


Figure 5.1: Detection mechanism for triggering speculative loading.

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 of 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 5.1). 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. 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 to avoid disk interference. This is realized with the `resume` control message (Figure 4.3) 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 [3] while in speculative loading it is invoked only in rare circumstances. Nothing is 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, which guarantees improved query performance as long as loading new data for every query. The safeguard mechanism enforces chunk loading independent of the system resource utilization. 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 two 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 the 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 into 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 the raw format into the database representation decreases with each query, subsequent queries run faster than the previous ones. Until all concerning data are loaded into the database. Because the system only loads the accessing columns during every query processing, the storage for data loading is adaptive—range from 0 to the storage cost of the database.

5.2 Merge Read Mechanism

An interesting situation arises when only some of the columns required for query processing are loaded either in cache or the database. In this case, `SCANRAW` has to read the remaining columns from the raw file. At the same time, `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., $T_{raw\ read}$, can be expressed as the following equation:

$$T_{raw\ read} = max\left(\frac{S}{r}, \frac{f(n, y+z)}{p}\right) \tag{5.1}$$

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. (5.1) to obtain the execution time for the query under the assumption that the entire raw file has to be read to extract the required attributes. Using the same notation, the execution time for *merge read*, i.e., $T_{merge\ read}$, is calculated in Eq. (5.2). 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.

$$T_{merge\ read} = max\left(\frac{S + R(z)}{r} + \delta t, \frac{f(n, y)}{p}\right) \tag{5.2}$$

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 approaches 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 parses 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.

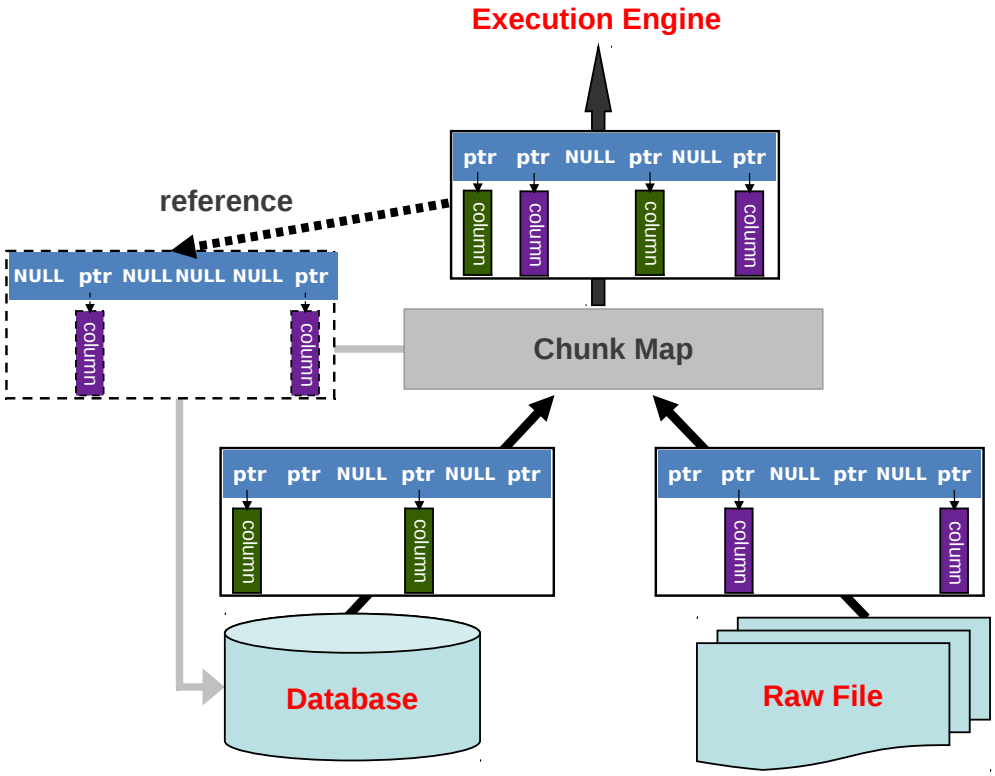


Figure 5.2: Merge read mechanism workflow.

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 5.2 shows the *merge read* workflow in SCANRAW. This workflow is based on the chunk structure and column-based storage schema depicted in Figure 4.1. 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 7.1. 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 5.2, and are sent to the WRITE thread for speculative loading.

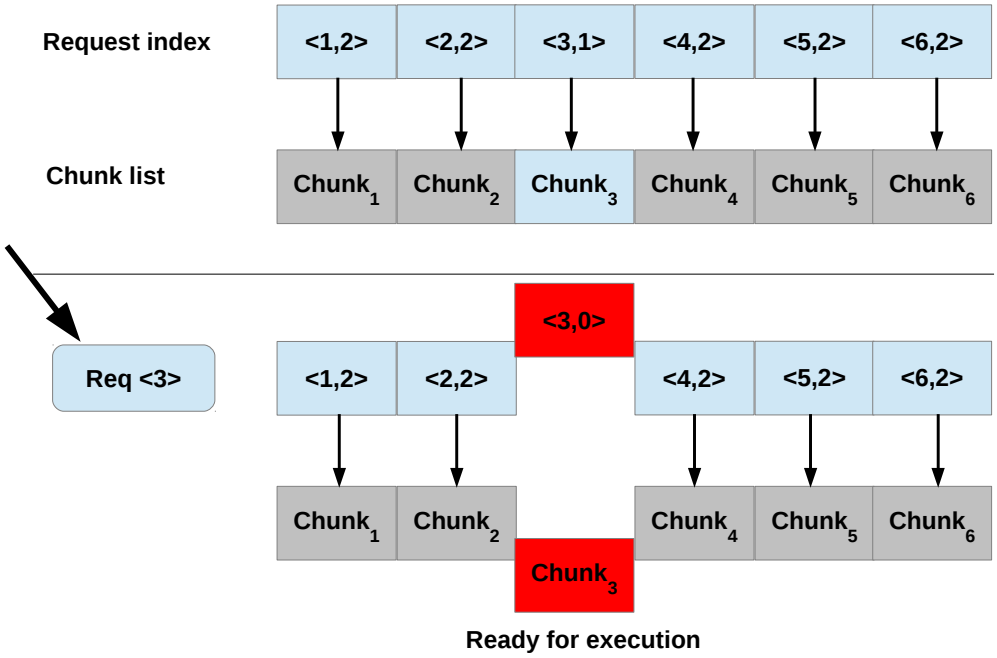


Figure 5.3: Example of a Chunk Map instance in merge read.

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 5.3) is designed to synchronize these requests and trace

the status of each chunk. It consists of two entities—the requests index and the chunk list. The request index is used to track 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 requests 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 in 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 5.3, 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 5.3 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.

5.3 Multi-Step Loading (MSL)

Using speculative loading, SCANRAW could instantly access to data and utilize the spare I/O resource to load data into the 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

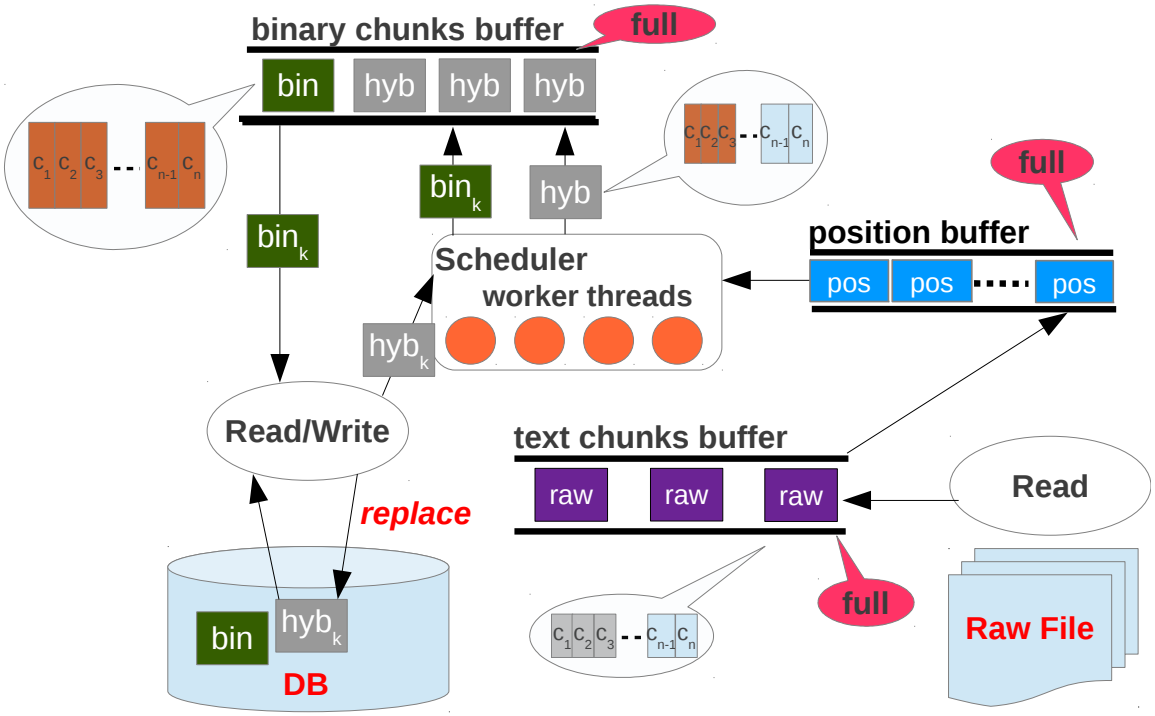


Figure 5.4: Detection mechanism for triggering multi-step loading..

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 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 5.4). 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 will be parsed due to the plenty of CPU resources. MSL works the same as speculative loading. When the `READ` is blocked, the processing becomes CPU-bound. At this time, MSL generates 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 six 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 run. 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 a smaller size. However, these attributes have to be parsed before sending the execution engine can process them. 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.

5.4 Experimental Evaluation

The objective of the experimental evaluation is to investigate the workflow for a serial of loading strategies. Specifically, the experiments we design are targeted to answer the following questions:

- 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? And a sequence of queries?
- What is the work behavior of multisteps loading compared to standard loading methods?
- What resource utilization does SCANRAW achieve combined with loading strategies?

5.4.1 Micro-Benchmarks

In this section, all the system configuration reuse the experiment environment in section 7.6.1.

5.4.1.1 Percentage of loaded data

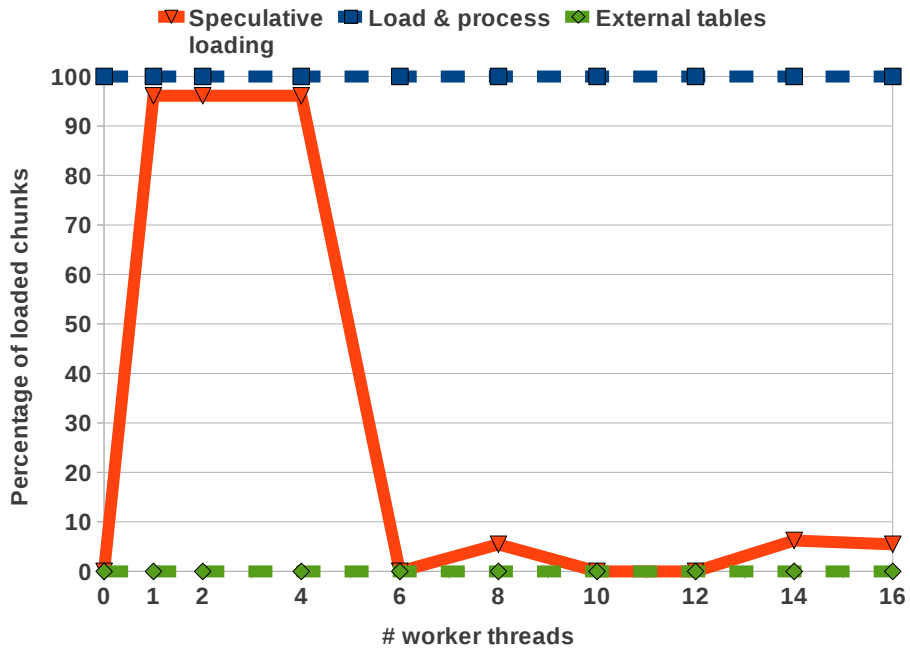


Figure 5.5: Percentage of loaded data as a function of the number of worker threads.

The effect of parallel processing on speculative loading is illustrated in Figure 5.5. 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.

5.4.1.2 Merge read mechanism

The merge read mechanism (Section 5.2) 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 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 of the two mechanisms.

The results are shown in Figure 5.6. Figure 5.6a 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 5.6b 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 5.6a. 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 fewer 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 5.6c and 5.6d 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

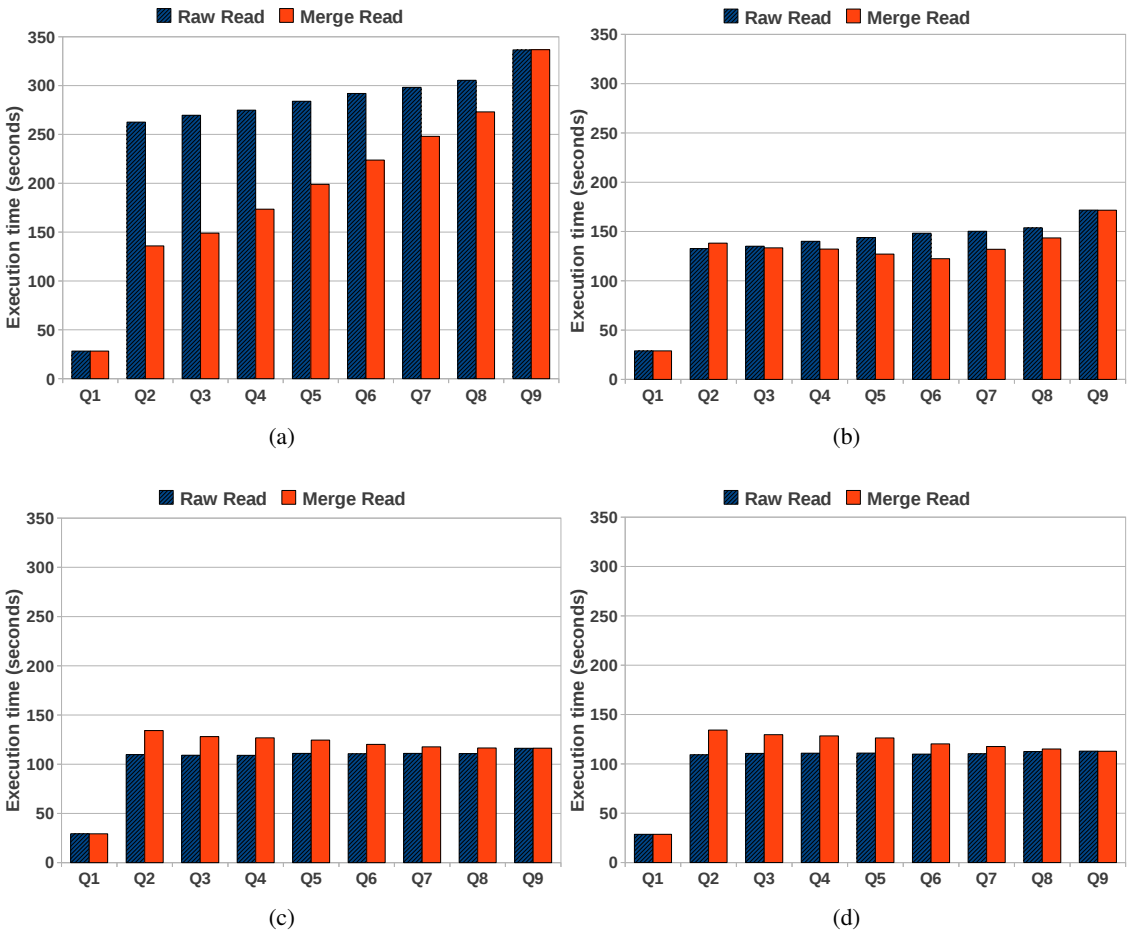


Figure 5.6: Comparison between *merge read* and *raw read* as a function of the number of worker threads used for data extraction: 1 (a), 2 (b), 3 (c), and 4 (d).

4 – *raw read* is always faster than *merge read*. The reason is that the queries become I/O-bound, thus 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.

5.4.1.3 Speculative loading for a query sequence

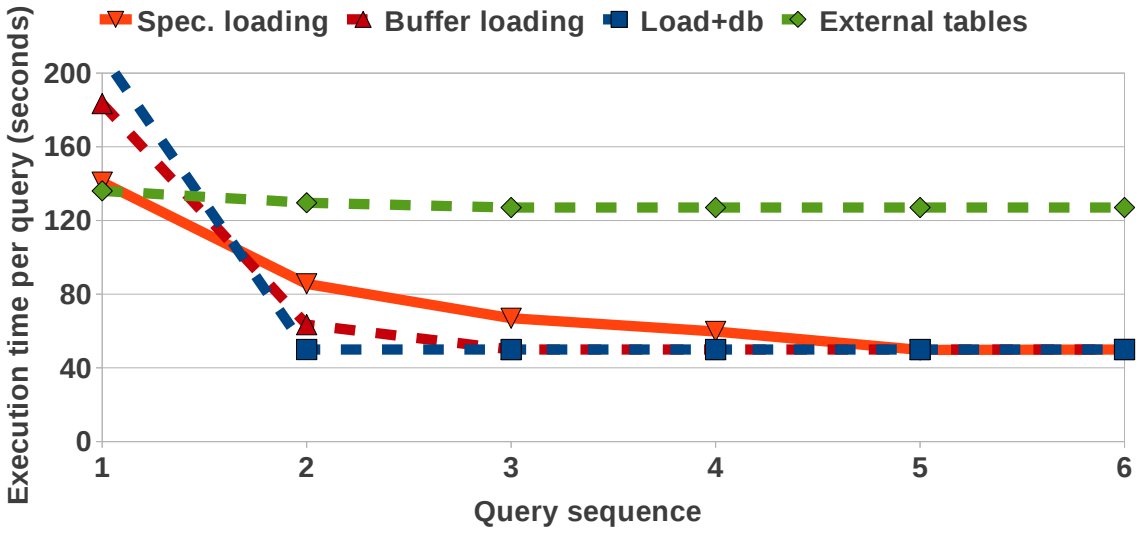


Figure 5.7: Execution time for query i.

Figure 5.7 and 5.8 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 during query processing is non-deterministic, it is more difficult to observe. Figure 5.7 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

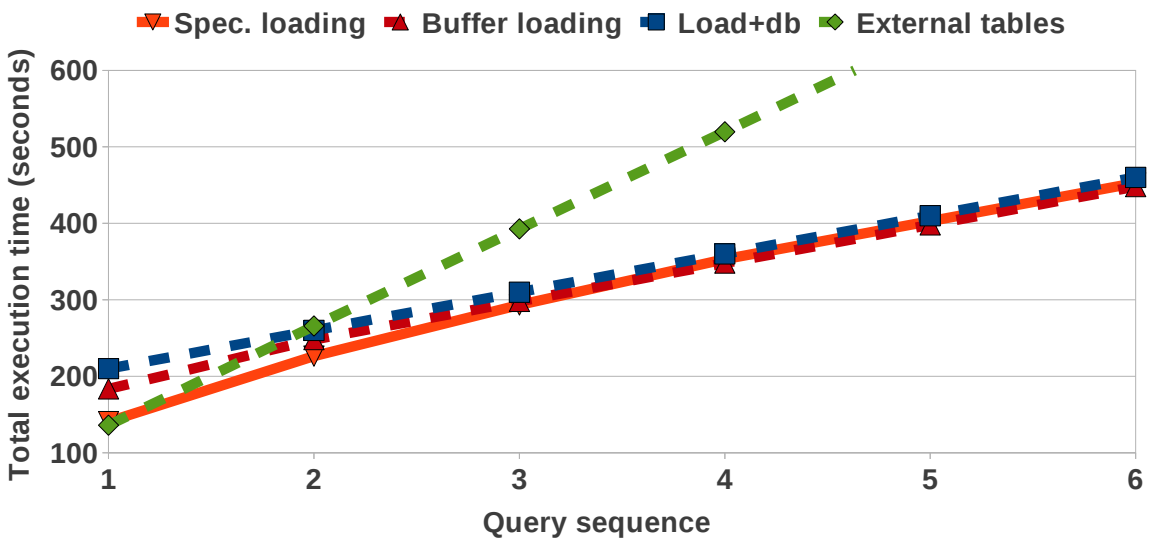


Figure 5.8: Overall execution time up to query *i*.

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 5.7, it takes only 5 queries. This is expected since the size of the cache is $1/4^{th}$ 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 5.8 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.

5.4.1.4 Multi-step loading for a query sequence

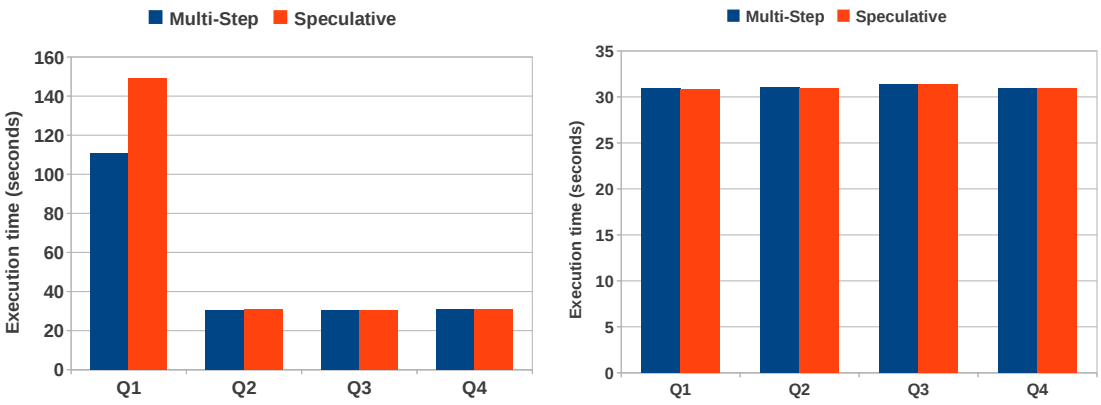


Figure 5.9: Multi-step loading for a sequence of identical queries: (a) 2 threads; (b) 16 threads.

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 the 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 5.9a 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 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 5.9b depicts the results in this case. As expected, the two loading strategies are identical. Their execution time is similar

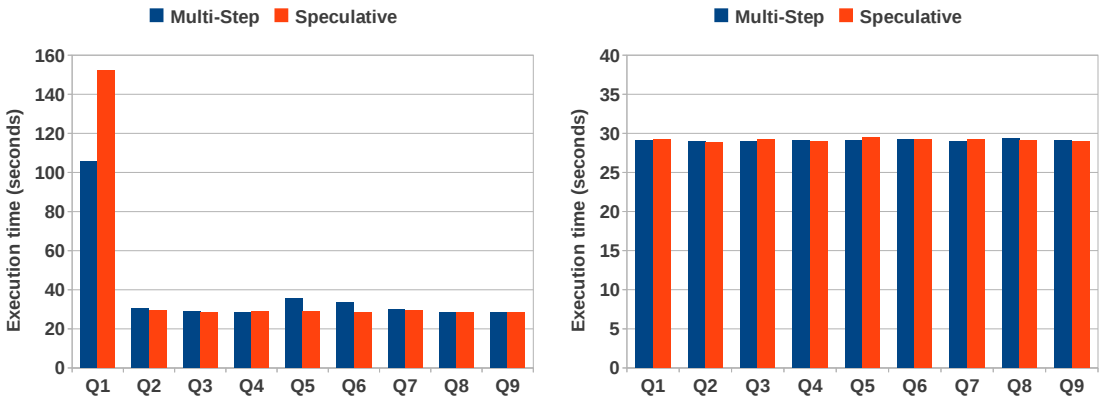


Figure 5.10: Multi-step loading for a sequence of different queries: (a) 2 threads; (b) 16 threads.

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. Every query accesses all the attributes. 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 5.10 depicts the results. We follow the same method to measure execution time when the processing is CPU-bound and I/O-bound, respectively. Figure 5.10a 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 same 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 extra 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 5.10b. They confirm that MSL and speculative loading are identical even for this workload.

5.4.1.5 Resource utilization

In Figure 5.11, 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 eight 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` to utilize resources optimally. Whenever the CPU is fully utilized, and no reading happens, `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

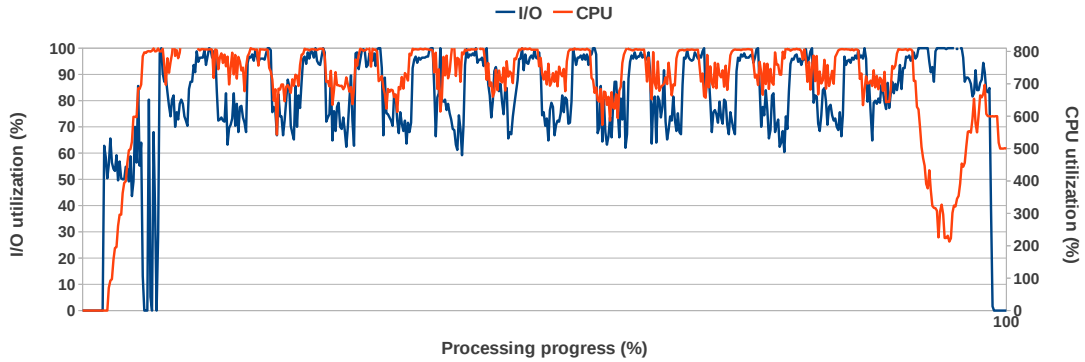


Figure 5.11: Resource utilization in SCANRAW.

soon as worker threads become available, the scheduler resumes reading and disk usage goes back to 100% since a sequence of chunks are typically read at a time.

5.4.2 Real Data

In order to evaluate SCANRAW on a real dataset, we use genomic sequence alignment data from the *1000 Genomes* project[1]. 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 the predicate. Table 5.1 shows the results we obtain for different SCANRAW configurations. In all the queries over SAM files, we use a SCANRAW implementation for processing tab-delimited text files. The tokenizing and parsing are handled inside SCANRAW. For BAM file processing, we use BAMTools to extract the tuples from binary and implement the 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 seven times less to process a file more than five 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 standard binary format which is widely used in astronomy to store, transmit, manipulate, and archive data. For instance, the Sloan Digital Sky Survey (SDSS)[2] data are available in FITS format. Besides the image data, FITS files can also store tables, either in ASCII or binary.

[1] http://www.1000genomes.org/data

[2] http://www.sdss3.org

A widely used tool to handle FITS files is the C library CFITSIO[3], 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. SCANRAW has only to map them to the binary chunk structure to make FITS data available for processing. This is a simple memory mapping operation that incurs no overhead.

Table 5.1 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 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.

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

Table 5.1: SCANRAW execution time (seconds) for real data.

5.4.3 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 CSV text files, without loading. Impala accesses data stored in the HDFS[4] 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. 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.

[3] http://heasarc.gsfc.nasa.gov/fitsio

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

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 5.12 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.

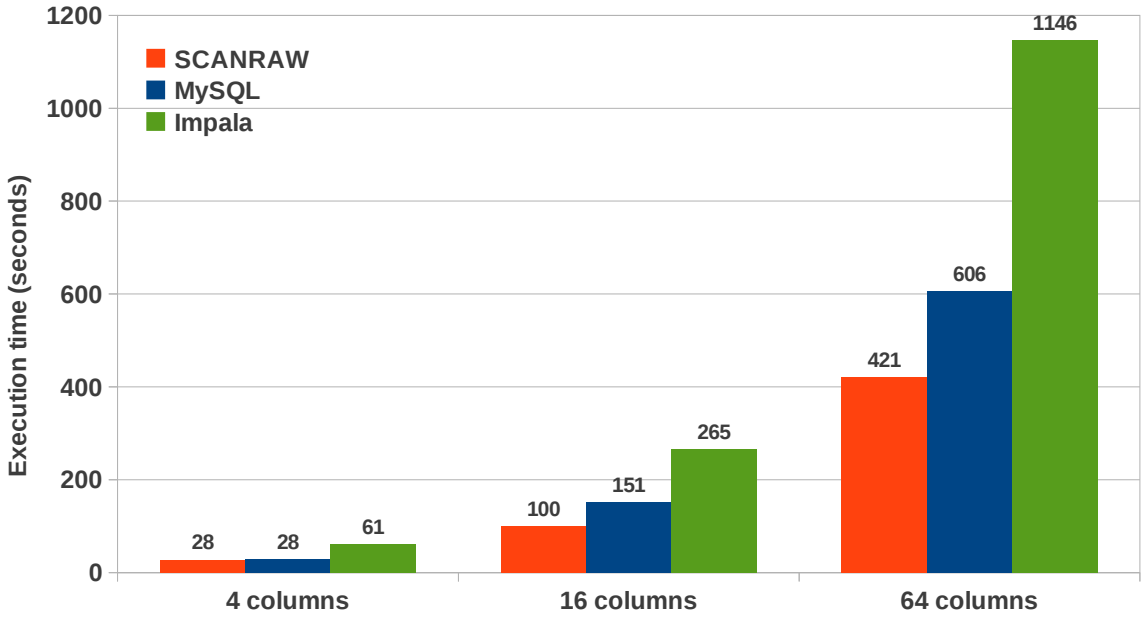


Figure 5.12: Comparison of the external tables mechanism.

5.5 Conclusion

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 a 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 [53]. SCANRAW with speculative loading achieves optimal performance across a sequence of queries at any point in the execution. It is similar to external tables for the first query and more efficient than database processing in the long run. Moreover, SCANRAW makes full data loading efficient to the point where database processing – with pre-loading – achieves better overall

execution time than external tables even for a two-query sequence. The time distribution is split almost equally among I/O and CPU-intensive pipeline stages. When the number of columns in the file is large, CPU-intensive stages – TOKENIZE and PARSE – account for more than 80% of the time to process a chunk. By overlapping processing across multiple chunks and between stages, SCANRAW makes even this type of execution I/O-bound. This method 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 accessing the raw file. Merging data from the database and the raw file proves efficient only when the number of threads allocated to data extraction is one, at most two.

Chapter 6

Vertical Partitioning for Query Processing over Raw Data

6.1 Background Description

Motivated by the flexibility of NoSQL systems to access schema-less data and by the Hadoop functionality to directly process data in any format, we have recently witnessed a sustained effort to bring these capabilities inside relational database management systems (RDBMS). Starting with version 9.3, PostgreSQL[1] includes support for JSON data type and corresponding functions. Vertica Flex Zone[2] and Sinew [69] implement flex table and column reservoir, respectively, for storing key-value data serialized as maps in a BLOB column. In both systems, certain keys can be promoted to individual columns, in storage as well as in a dynamically evolving schema. With regards to directly processing raw data, several query-driven extensions have been proposed to the loading and external table [54, 74] mechanisms. Instead of loading all the columns before querying, in adaptive partial loading [38] data are loaded only at query time, and only the attributes required by the query. This idea is further extended in invisible loading [3], where only a fragment of the queried columns are loaded, and in NoDB [7], data vaults [42], SDS/Q [14], and RAW [47], where columns are loaded only in memory, but not into the database. SCANRAW [20] is a super-scalar pipeline operator that loads data speculatively, only when spare I/O resources are available. While these techniques enhance the RDBMS' flexibility to process schema-less raw data, they have several shortcomings, as the following examples show.

Example 1: Twitter data. The Twitter API[3] provides access to several objects in JSON format through a well-defined interface. The schema of the objects is, however, not well-defined, since it includes "nullable" attributes and nested objects. The state-of-the-art RDBMS solution to process semi-structured JSON data [69] is first to load the objects as tuples in a BLOB column. Essentially, this entails complete data duplication, even though many of the object attributes are never used. The internal representation consists of a map of key-values that is serialized/deserialized

[1] http://www.postgresql.org/

[2] http://www.vertica.com/tag/flexzone/

[3] https://dev.twitter.com/docs/platform-objects/

into/from persistent storage. The map can be directly queried from SQL based on the keys, treated as virtual attributes. As an optimization, certain columns – chosen by the user or by the system based on appearance frequency – are promoted to physical status. The decision on which columns to materialize is only a heuristic, quite often sub-optimal.

Example 2: Sloan Digital Sky Survey (SDSS) data. SDSS[4] is a decade-long astronomy project having the goal to build a catalog of all the astrophysical objects in the observable Universe. Images of the sky are taken by a high-resolution telescope, typically in binary FITS format. The catalog data summarize quantities measured from the images for every detected object. The catalog is stored as binary FITS tables. Additionally, the catalog data are loaded into an RDBMS and made available through standard SQL queries. The loading process replicates multi-terabyte data three times – in ASCII CSV and internal database representation – and it can take several days—if not weeks [67]. In order to evaluate the effectiveness of the loading, we extract a workload of 1 million SQL queries executed over the SDSS catalog[5] in 2014. The most frequent table in the workload is `photoPrimary`, which appears in more than 70% of the queries. `photoPrimary` has 509 attributes, out of which only 74 are referenced in queries. This means that 435 attributes are replicated three times without ever being used—a significantly sub-optimal storage utilization.

Inspired by the above examples, we study the raw data processing with partial loading problem. *Given a dataset in some raw format, a query workload, and a limited database storage budget, find what data to load in the database such that the overall workload execution time is minimized.* This is a standard database optimization problem with bounded constraints, similar to vertical partitioning in physical database design [47]. However, while physical design investigates what non-overlapping partitions to build over internal database data, we focus on what data to load, i.e., replicate, in a columnar database with support for multiple storage formats.

Existing solutions for loading and raw data processing are not adequate for our problem. Complete loading not only requires a significant amount of storage and takes a prohibitively long time, but is also unnecessary for many workloads. Pure raw data processing solutions [7, 14, 42, 47] are not adequate either, because parsing semi-structured JSON data repeatedly is time-consuming. Moreover, accessing data from the database is clearly optimal in the case of workloads with tens of queries. The drawback of query-driven, adaptive loading methods [3, 20, 38] is that they are greedy, workload-agnostic. Loading is decided based upon each query individually. It is easy to imagine a query order in which the first queries access non-frequent attributes that fill the storage budget, but have limited impact on the overall workload execution time.

6.1.1 Problem Statement

Consider a relational schema $R(A_1, A_2, \ldots, A_n)$ and an instantiation of it that contains $|R|$ tuples. Semi-structured JSON data can be mapped to the relational model by linearizing nested constructs [69]. In order to execute queries over R, tuples have to be read in memory and converted from the storage format into the processing representation. Two timing components correspond to this process. T_{RAW} is the time to read data from storage into memory. T_{RAW} can be computed straightforwardly for a given schema and storage bandwidth $band_{IO}$. A constraint specific to raw

[4] www.sdss.org/dr12/

[5] http://skyserver.sdss.org/CasJobs/

file processing – and row-store databases, for that matter – is that all the attributes are read in a query—even when not required. T_{CPU} is the second timing component. It corresponds to the conversion time. For every attribute A_j in the schema, the conversion is characterized by two parameters, defined at tuple level. The *tokenizing time* T_{t_j} is the time to locate the attribute in a tuple in storage format. The *parsing time* T_{p_j} is the time to convert the attribute from storage format into processing representation. A limited amount of storage B is available for storing data converted into the processing representation. This eliminates the conversion and replaces it with an I/O process that operates at column level—only complete columns can be saved in the processing format. The time to read an attribute A_j in processing representation, T_j^{IO}, can be determined when the type of the attribute and $|R|$ are known.

	A_1	A_2	A_3	A_4	A_5	A_6	A_7	A_8
Q_1	X	X						
Q_2	X	X	X	X				
Q_3			X	X	X			
Q_4		X		X		X		
Q_5	X		X	X	X		X	
Q_6	X	X	X	X	X	X	X	

Table 6.1: Query access pattern to raw data attributes.

Consider a workload $W = \{Q_1, Q_2, \ldots, Q_m\}$ of m SQL-like queries executed over the schema R. The workload can be extracted from historical queries or it can be defined by an expert user. Each query Q_i is characterized by $\{A_{j_1}, A_{j_2}, \ldots, A_{j_{|Q_i|}}\}$, a subset of attributes accessed by the query. Queries are assumed to be distinct, i.e., there are no two queries that access exactly the same set of attributes. A weight w_i characterizing importance, e.g., frequency in the workload, is assigned to every query. Ideally, $\sum_i w_i = 1$, but this is not necessary.

The problem we investigate in this section is *how to optimally use the storage B such that the overall query workload execution time is minimized?* Essentially, what attributes to save in processing representation in order to minimize raw file query processing time? We name this problem *raw data processing with partial loading*. We study two versions of the problem—serial and pipeline. In the serial problem, the I/O and the conversion are executed sequentially, while in the pipeline problem, they can overlap. Similar to offline physical database design [17], the conversion of the attributes stored in processing representation is executed prior to the workload execution. We let the online problem [4], in which conversion and storage are intertwined, for future work.

6.1.2 Illustrative Example

Table 6.1 depicts the access pattern of a workload of 6 queries to the 8 attributes in a raw file. X corresponds to the attribute being accessed in the respective query. For example, Q_1 can be represented as $Q_1 = \{A_1, A_2\}$. For simplicity, assume that the weights are identical across queries, i.e., $w_i = 1/6$, $1 \leq i \leq 6$. If the amount of storage B that can be used for loading data into the processing representation allows for at most 3 attributes to be loaded, i.e., $B = 3$, the problem we

Variable	Description
raw_i; $i = \overline{0,m}$	read raw file at query i
t_{ij}; $i = \overline{0,m}$, $j = \overline{1,n}$	tokenize attribute j at query i
p_{ij}; $i = \overline{0,m}$, $j = \overline{1,n}$	parse attribute j at query i
$read_{ij}$; $i = \overline{1,m}$, $j = \overline{1,n}$	read attribute j at query i from processing format
$save_j$; $j = \overline{1,n}$	load attribute j in processing format

Table 6.2: Variables in MIP optimization.

address in this paper is what 3 attributes to load such that the workload execution time is minimized? Since A_8 is not referenced in any of the queries, we are certain that A_8 is not one of the attributes to be loaded. Finding the best 3 out of the remaining 7 is considerably more difficult.

Parameter	Description
$\|R\|$	number of tuples in relation R
S_{RAW}	size of raw file
SPF_j, $j = \overline{1,n}$	size of attribute j in processing format
B	size of storage in processing format
$band_{IO}$	storage bandwidth
T_{t_j}, $j = \overline{1,n}$	time to tokenize an instance of attribute j
T_{p_j}, $j = \overline{1,n}$	time to parse an instance of attribute j
w_i, $i = \overline{1,m}$	weight for query i

Table 6.3: Parameters in MIP optimization.

6.1.3 Related Work

Two lines of research are most relevant to the work presented in this paper—raw data processing and vertical partitioning as a physical database design technique. Our contribution is to integrate workload information in raw data processing and model the problem as vertical partitioning optimization. To the best of our knowledge, this is the first paper to consider the problem of optimal vertical partitioning for raw data processing with partial loading.

Raw data processing. Several methods have been proposed for processing raw data within a database engine. The vast majority of them bring enhancements to the external table functionality, already supported by several major database servers [54, 74]. A common factor across many of these methods is that they do not consider loading converted data inside the database. At most, data are cached in memory on a query-by-query basis. This is the approach taken in NoDB [7], Data Vaults [42], SDS/Q [14], RAW [47], and Impala [51]. Even when loading is an option, for

example in adaptive partial loading [38], invisible loading [3], and SCANRAW [20], the workload is not taken into account and the storage budget is unlimited. The decision on what to load is local to every query, thus, prone to be acutely sub-optimal over the entire workload.

The heuristic developed in this paper requires workload knowledge and aims to identify the optimal data to load such that the execution time of the entire workload is minimized. As in standard database processing, loading is executed offline, before query execution. However, the decision on what data to load is intelligent and the time spent on loading is limited by the allocated storage budget. Notice that the heuristic is applicable both to secondary storage-based loading as well as to one-time in-memory caching without subsequent replacement.

Vertical partitioning. Vertical partitioning has a long-standing history as a physical database design strategy, dating back to the 1970's. Many types of solutions have been proposed over the years, ranging from integer programming formulations to top-down and bottom-up heuristics that operate at the granularity of a query or of an attribute. A comparative analysis of several vertical partitioning algorithms is presented in [44]. The serial MIP formulation for raw data processing is inspired from the formulations for vertical partitioning given in [24, 37]. While both are non-linear, none of these formulations considers pipeline processing. We prove that even the linear MIP formulation is NP-hard. The scale of the previous results for solving MIP optimizations have to be taken with a grain of salt, given the extensive enhancements to integer programming solvers over the past two decades. As explained in Section 6.3.5, the proposed heuristic combines ideas from several classes of vertical partitioning algorithms, adapting their optimal behavior to raw data processing with partial loading. The top-down transaction-level algorithm given in [22] is the closest to the query coverage stage. While query coverage is a greedy algorithm, [22] employs exhaustive search to find the solution. As the experimental results show, this is time-consuming. Other top-down heuristics [5, 56] consider the interaction between attributes across the queries in the workload. The partitioning is guided by a quantitative parameter that measures the strength of the interaction. In [56], only the interaction between pairs of attributes is considered. The attribute usage frequency phase of the proposed heuristic treats each attribute individually, but only after query coverage is executed. The objective in [5] is to find a set of vertical partitions that are subsequently evaluated for index creation. Since we select a single partitioning scheme, the process is less time-consuming. Finally, the difference between the proposed heuristic and bottom-up algorithms [31, 33, 34, 45, 58] is that the latter cannot guarantee that only two partitions are generated at the end. This is a requirement for raw data processing with partial loading. All these algorithms are offline. They are executed only once, before query processing, over a known workload. Online vertical partitioning algorithms form a separate class. In [4], the entire workload sequence is known in advance. However, the vertical partitioning evolves with each query. Another series of algorithms [43, 52, 68] operate over an unknown workload, given one query at a time. Their goal is to gather evidence from the past workload in order to determine the optimal vertical partitioning for future queries.

6.2 Mixed Integer Programming

In order to reason on the complexity of the problem and discuss our solution in a formal framework, we model raw file query processing as mixed integer programming (MIP) [13] optimization with 0/1 variables. Table 6.2 and 6.3 contain the variables and parameters used in the

optimization formulation, respectively. Query index 0 corresponds to saving in the processing representation, i.e., loading, executed before processing the query workload. Parameters include characteristics of the data and the system. The majority of them can be easily determined. The time to tokenize T_{t_j} and parse T_{p_j} an attribute are the most problematic since they depend both on the data and the system, respectively. Their value can be configured from previous workload executions or, alternatively, by profiling the execution of the extraction process on a small sample of the raw file.

The MIP optimization problem for serial raw data processing is formalized as follows (we discuss the pipeline formulation in Section 6.4):

$$\begin{aligned}
&\text{minimize } \; T_{load} + \sum_{i=1}^{m} w_i \cdot T_i \; \text{ subject to constraints:} \\
&C_1: \; \sum_{j=1}^{n} save_j \cdot SPF_j \cdot |R| \leq B \\
&C_2: \; read_{ij} \leq save_j; \; i = \overline{1,m}, \; j = \overline{1,n} \\
&C_3: \; save_j \leq p_{0j} \leq t_{0j} \leq raw_0; \; j = \overline{1,n} \\
&C_4: \; p_{ij} \leq t_{ij} \leq raw_i; \; i = \overline{1,m}, \; j = \overline{1,n} \\
&C_5: \; t_{ij} \leq t_{ik}; \; i = \overline{0,m}, \; j > k = \overline{1,n-1} \\
&C_6: \; read_{ij} + p_{ij} = 1; \; i = \overline{1,m}, \; j = \overline{1,n}, \; A_j \in Q_i
\end{aligned} \tag{6.1}$$

6.2.1 Objective Function

The linear objective function consists of two terms. The time to load columns in processing representation T_{load} is defined as:

$$\begin{aligned}
T_{load} = & raw_0 \cdot \frac{S_{RAW}}{band_{IO}} + \\
& |R| \cdot \sum_{j=1}^{n} \left(t_{0j} \cdot T_{t_j} + p_{0j} \cdot T_{p_j} + save_j \cdot \frac{SPF_j}{band_{IO}} \right)
\end{aligned} \tag{6.2}$$

while the execution time corresponding to a query T_i is a slight modification:

$$\begin{aligned}
T_i = & raw_i \cdot \frac{S_{RAW}}{band_{IO}} + \\
& |R| \cdot \sum_{j=1}^{n} \left(t_{ij} \cdot T_{t_j} + p_{ij} \cdot T_{p_j} + read_{ij} \cdot \frac{SPF_j}{band_{IO}} \right)
\end{aligned} \tag{6.3}$$

In both cases, the term outside the summation corresponds to reading the raw file. The first term under the sum is for tokenizing, while the second is for parsing. The difference between loading and query execution is only in the third term. In the case of loading, variable $save_j$ indicates if attribute j is saved in processing representation, while in query execution, variable $read_{ij}$ indicates if attribute j is read from the storage corresponding to the processing format at query i. We make the

reasonable assumption that the read and write I/O bandwidth are identical across storage formats. They are given by $band_{IO}$.

6.2.2 Constraints

There are six types of linear constraints in our problem. Constraint C_1 bounds the amount of storage that can be used for loading data in the processing representation. While C_1 is a capacity constraint, the remaining constraints are functional, i.e., they dictate the execution of the raw file query processing mechanism. C_2 enforces that any column read from processing format has to be loaded first. There are $O(m \cdot n)$ such constraints—one for every attribute in every query. Constraint C_3 models loading. In order to save a column in processing format, the raw file has to be read and the column has to be tokenized and parsed, respectively. While written as a single constraint, C_3 decomposes into three separate constraints – one corresponding to each "≤" operator – for a total of $O(3 \cdot n)$ constraints. C_4 is a reduced form of C_3, applicable to query processing. The largest number of constraints, i.e., $O(m \cdot n^2)$, in the MIP formulation are of type C_5. They enforce that it is not possible to tokenize an attribute in a tuple without tokenizing all the preceding schema attributes in the same tuple. C_5 applies strictly to raw files without direct access to individual attributes. Constraint C_6 guarantees that every attribute accessed in a query is either extracted from the raw file or read from the processing representation.

6.2.3 Computational Complexity

There are $O(m \cdot n)$ binary 0/1 variables in the linear MIP formulation, where m is the number of queries in the workload and n is the number of attributes in the schema. Solving the MIP directly is, thus, impractical for workloads with tens of queries over schemas with hundreds of attributes, unless the number of variables in the search space can be reduced. We prove that this is not possible by providing a reduction from a well-known NP-hard problem to a restricted instance of the MIP formulation. Moreover, we also show that no approximation exists.

Definition 1 (k-element cover) *Given a set of n elements $R = \{A_1, \ldots, A_n\}$, m subsets $W = \{Q_1, \ldots, Q_m\}$ of R, such that $\bigcup_{i=1}^{m} Q_i = R$, and a value k, the objective in the k-element cover problem is to find a size k subset R' of R that covers the largest number of subsets Q_i, i.e., $Q_i \subseteq R'$, $1 \leq i \leq m$.*

For the example in Table 6.1, $\{A_1, A_2\}$ is the single 2-element cover solution (covering Q_1). While many 3-element cover solutions exist, they all cover only one query.

The k-element cover problem is a restricted instance of the MIP formulation, in which parameters T_{t_j}, T_{p_j}, and the loading and reading time to/from database are set to zero, i.e., $\frac{SPF_j \cdot |R|}{band_{IO}} \rightarrow 0$, while the raw data reading time is set to one, i.e., $\frac{S_{RAW}}{band_{IO}} \rightarrow 1$. The objective function is reduced to counting how many times raw data have to be accessed. The bounding constraint limits the number of attributes that can be loaded, i.e., $save_j = 1$, while the functional constraints determine the value of the other variables. The optimal solution is given by the configuration that minimizes the number of queries accessing raw data. A query does not access raw data when the $read_{ij}$ variables corresponding to its attributes are all set to one. When the entire workload is considered, this equates to finding those attributes that cover the largest number of queries, i.e., finding the k-attribute cover of

the workload. Given this reduction, it suffices to prove that k-element cover is NP-hard for the MIP formulation to have only exponential-time solutions. We provide a reduction to the well-known minimum k-set coverage problem [71] that proves k-element cover is NP-hard.

Definition 2 (minimum k-set coverage) *Given a set of n elements $R = \{A_1, \ldots, A_n\}$, m subsets $W = \{Q_1, \ldots, Q_m\}$ of R, such that $\bigcup_{i=1}^{m} Q_i = R$, and a value k, the objective in the minimum k-set coverage problem is to choose k sets $\{Q_{i_1}, \ldots Q_{i_k}\}$ from W whose union has the smallest cardinality, i.e., $\left|\bigcup_{j=1}^{k} Q_{i_j}\right|$.*

Algorithm 2 Reduce k-element cover to minimum k'-set coverage

Require: Set $R = \{A_1, \ldots, A_n\}$ and m subsets $W = \{Q_1, \ldots, Q_m\}$ of R; number k' of sets Q_i to choose in minimum set coverage
Ensure: Minimum number k of elements from R covered by choosing k' subsets from W
1: **for** $i = 1$ to n **do**
2: $res =$ ***k-element cover***(W, i)
3: **if** $res \geq k'$ **then return** i
4: **end for**

Algorithm 2 gives a reduction from k-element cover to minimum k-set coverage. The solution to minimum k-set coverage is obtained by invoking k-element cover for any number of elements in R and returning the smallest such number for which the solution to k-element cover contains at least k' subsets of W. Since we know that minimum k-set coverage is NP-hard [71] and the solution is obtained by solving k-element cover, it implies that k-element cover cannot be any simpler, i.e., k-element cover is also NP-hard. The following theorem formalizes this argument.

Theorem 1 *The reduction from k-element cover to minimum k-set coverage given in Algorithm 2 is correct and complete.*

The proof of this theorem is available in the extended version of the paper [75].

Corollary 1 *The MIP formulation is NP-hard and cannot be approximated unless NP-complete problems can be solved in randomized sub-exponential time.*

The NP-hardness is a direct consequence of the reduction to the k-element cover problem and Theorem 1. In addition, [65] proves that minimum k-set coverage cannot be approximated within an absolute error of $\frac{1}{2}m^{1-2\epsilon} + O(m^{1-3\epsilon})$, for any $0 < \epsilon < \frac{1}{3}$, unless P = NP. Consequently, the MIP formulation cannot be approximated.

6.3 Heuristic Algorithm

In this section, we propose a novel heuristic algorithm for raw data processing with partial loading that has as a starting point a greedy solution for the k-element cover problem. The algorithm also includes elements from vertical partitioning—a connection we establish in the paper. The central idea is to combine *query coverage* with *attribute usage frequency* in order to determine the best attributes to load. At a high level, query coverage aims at reducing the number of queries that require access to the raw data, while usage frequency aims at eliminating the repetitive extraction of the heavily-used attributes. Our algorithm reconciles between these two conflicting criteria by optimally dividing the available loading budget across them, based on the format of the raw data and the query workload. The solution found by the algorithm is guaranteed to be as good as the solution corresponding to each criterion, considered separately.

In the following, we make the connection with vertical partitioning clear. Then, we present separate algorithms based on query coverage and attribute usage frequency. These algorithms are combined into the proposed heuristic algorithm for raw data processing with partial loading. We conclude the section with a detailed comparison between the proposed heuristic and algorithms designed specifically for vertical partitioning.

6.3.1 Vertical Partitioning

Vertical partitioning [56] of a relational schema $R(A_1, \ldots, A_n)$ splits the schema into multiple schemas – possibly overlapping – each containing a subset of the columns in R. For example, $\{R_1(A_1); R_2(A_2); \ldots R_n(A_n)\}$ is the atomic non-overlapping vertical partitioning of R in which each column is associated with a separate partition. Tuple integrity can be maintained either by sorting all the partitions in the same order, i.e., positional equivalence, or by pre-pending a tuple identifier (*tid*) column to every partition. Vertical partitioning reduces the amount of data that have to be accessed by queries that operate on a small subset of columns since only the required columns have to be scanned—when they form a partition. However, tuple reconstruction [40] can become problematic when integrity is enforced through *tid* values because of joins between partitions. This interplay between having partitions that contain only the required columns and access confined to a minimum number of partitions, i.e., a minimum number of joins, is the objective function to minimize in vertical partitioning. The process is always workload-driven.

Raw data processing with partial loading can be mapped to *fully-replicated binary vertical partitioning* as follows. The complete raw data containing all the attributes in schema R represent the *raw partition*. The second partition – *loaded partition* – is given by the attributes loaded in processing representation. These are a subset of the attributes in R. The storage allocated to the loaded partition is bounded. The asymmetric nature of the two partitions differentiates raw data processing from standard vertical partitioning. The raw partition provides access to all the attributes, at the cost of tokenizing and parsing. The loaded partition provides faster access to a reduced set of attributes. In vertical partitioning, all the partitions are equivalent. While having only two partitions may be regarded as a simplification, all the top-down algorithms we are aware of [5, 22, 56] apply binary splits recursively in order to find the optimal partitions. The structure of raw data processing with partial loading limits the number of splits to one.

6.3.2 Query Coverage

A query that can be processed without accessing the raw data is said to be *covered*. In other words, all the attributes accessed by the query are loaded in processing representation. These are the queries whose attributes are contained in the solution to the k-element cover problem. Intuitively, increasing the number of covered queries results in a reduction to the objective function, i.e., total query workload execution time, since only the required attributes are accessed. Moreover, access to the raw data and conversion are completely eliminated. However, given a limited storage budget, it is computationally infeasible to find the optimal set of attributes to load—the k-element cover problem is NP-hard and cannot be approximated (Corollary 1). Thus, heuristic algorithms are required.

Algorithm 3 Query coverage

Require: Workload $W = \{Q_1, \ldots, Q_m\}$; storage budget B
Ensure: Set of attributes $\{A_{j_1}, \ldots, A_{j_k}\}$ to be loaded in processing representation
1: $attsL = \emptyset$; $coveredQ = \emptyset$
2: **while** $\sum_{j \in attsL} SPF_j < B$ **do**
3: $idx = \text{argmax}_{i \notin coveredQ} \left\{ \frac{cost(attsL) - cost(attsL \cup Q_i)}{\sum_{j \in \{attsL \cup Q_i \setminus attsL\}} SPF_j} \right\}$
4: **if** $cost(attsL) - cost(attsL \cup Q_{idx}) \leq 0$ **then break**
5: $coveredQ = coveredQ \cup idx$
6: $attsL = attsL \cup Q_{idx}$
7: **end while**
8: **return** $attsL$

We design a standard greedy algorithm for the k-element cover problem that maximizes the number of covered queries within a limited storage budget. The pseudo-code is given in Algorithm 3. The solution *attsL* and the covered queries *coveredQ* are initialized with the empty set in line 1. As long as the storage budget is not exhausted (line 2) and the value of the objective function *cost* (Eq. (6.2) and Eq. (6.3)) decreases (line 4), a query to be covered is selected at each step of the algorithm (line 3). The criterion we use for selection is the reduction in the cost function normalized by the storage budget, i.e., we select the query that provides the largest reduction in cost, while using the smallest storage. This criterion gives preference to queries that access a smaller number of attributes and is consistent with our idea of maximizing the number of covered queries. An alternative selection criterion is to drop the cost function and select the query that requires the least number of attributes to be added to the solution. The algorithm is guaranteed to stop when no storage budget is available or all the queries are covered.

Example. We illustrate how the ***Query coverage*** algorithm works on the workload in Table 6.1. Without loss of generality, assume that all the attributes have the same size and the time to access raw data is considerably larger than the extraction time and the time to read data from processing representation, respectively. These is a common situation in practice, specific to delimited text file formats, e.g., CSV. Let the storage budget be large enough to load three attributes, i.e., $B = 3$. In the first step, only queries Q_1, Q_3, and Q_4 are considered for coverage in line 3, due to the storage constraint. While the same objective function value is obtained for each query, Q_1 is selected for loading because it provides the largest normalized reduction, i.e., $\frac{T_{RAW}}{2}$. The other two

queries have a normalized reduction of $\frac{T_{RAW}}{3}$, where T_{RAW} is the time to read the raw data. In the second step of the algorithm, $attsL = \{A_1, A_2\}$. This also turns to be the last step since no other query can be covered in the given storage budget. Notice that, although Q_3 and Q_4 make better use of the budget, the overall objective function value is hardly different, as long as reading raw data is the dominating cost component.

6.3.3 Attribute Usage Frequency

The query coverage strategy operates at query granularity. An attribute is always considered as part of the subset of attributes accessed by the query. It is never considered individually. This is problematic for at least two reasons. First, the storage budget can be under-utilized, since a situation where storage is available but no query can be covered, can appear during execution. Second, a frequently-used attribute or an attribute with a time-consuming extraction may not get loaded if, for example, is part of only long queries. The assumption that accessing raw data is the dominant cost factor does not hold in this case. We address these deficiencies of the query coverage strategy by introducing a simple greedy algorithm that handles attributes individually. As the name implies, the intuition behind the attribute usage frequency algorithm is to load those attributes that appear frequently in queries. The rationale is to eliminate the extraction stages that incur the largest cost in the objective function.

Algorithm 4 Attribute usage frequency

Require: Workload $W = \{Q_1, \ldots, Q_m\}$ of R; storage budget B; set of loaded attributes $saved = \{A_{s_1}, \ldots, A_{s_k}\}$

Ensure: Set of attributes $\{A_{s_{k+1}}, \ldots, A_{s_{k+t}}\}$ to be loaded in processing representation

1: $attsL = saved$
2: **while** $\sum_{j \in attsL} SPF_j < B$ **do**
3: $\quad idx = \text{argmax}_{j \notin attsL} \left\{ cost\left(attsL\right) - cost\left(attsL \cup A_j\right) \right\}$
4: $\quad attsL = attsL \cup idx$
5: **end while**
6: **return** $attsL$

The pseudo-code for the attribute usage frequency strategy is given in Algorithm 4. In addition to the workload and the storage budget, a set of attributes already loaded in the processing representation is passed as argument. At each step (line 3), the attribute that generates the largest decrease in the objective function is loaded. In this case, the algorithm stops only when the entire storage budget is exhausted (line 2).

Example. We illustrate how the ***Attribute usage frequency*** algorithm works by continuing the example started in the query coverage section. Recall that only two attributes $saved = \{A_1, A_2\}$ out of a total of three are loaded. A_4 is chosen as the remaining attribute to be loaded since it appears in five queries, the largest number between unloaded attributes. Given that all the attributes have the same size and there is no cost for tuple reconstruction, $\{A_1, A_2, A_4\}$ is the optimal loading configuration for the example in Table 6.1.

6.3.4 Putting It All Together

The heuristic algorithm for raw data processing with partial loading unifies the query coverage and attribute usage frequency algorithms. The pseudo-code is depicted in Algorithm 5. Given a storage budget B, ***Query coverage*** is invoked first (line 3). ***Attribute usage frequency*** (line 4) takes as input the result produced by ***Query coverage*** and the unused budget Δ_q. Instead of invoking these algorithms only once, with the given storage budget B, we consider a series of allocations. B is divided in δ increments (line 2). Each algorithm is assigned anywhere from 0 to B storage, in δ increments. A solution is computed for each of these configurations. The heuristic algorithm returns the solution with the minimum objective. The increment δ controls the complexity of the algorithm. Specifically, the smaller δ is, the larger the number of invocations to the component algorithms. Notice, though, that as long as $\frac{B}{\delta}$ remains constant with respect to m and n, the complexity of the heuristic remains $O(m + n)$.

The rationale for using several budget allocations between query coverage and attribute usage frequency lies in the limited view they take for solving the optimization formulation. Query coverage assumes that the access to the raw data is the most expensive cost component, i.e., processing is I/O-bound, while attribute usage frequency focuses exclusively on the extraction, i.e., processing is CPU-bound. However, the actual processing is heavily-dependent on the format of the data and the characteristics of the system. For example, binary formats, e.g., FITS, do not require extraction, while hierarchical text formats, e.g., JSON, require complex parsing. Moreover, the extraction complexity varies largely across data types. The proposed heuristic algorithm recognizes these impediments and solves many instances of the optimization formulation in order to identify the optimal solution.

Algorithm 5 Heuristic algorithm

Require: Workload $W = \{Q_1, \ldots, Q_m\}$; storage budget B
Ensure: Set of attributes $\{A_{j_1}, \ldots, A_{j_k}\}$ to be loaded in processing representation

```
1:  obj_min = ∞
2:  for i = 0; i = i + δ; i ≤ B do
3:      attsL_q = Query coverage(W, i)
4:      attsL_f = Attribute usage frequency(W, Δ_q, attsL_q)
5:      attsL = attsL_q ∪ attsL_f
6:      obj = cost(attsL)
7:      if obj < obj_min then
8:          obj_min = obj
9:          attsL_min = attsL
10:     end if
11: end for
12: return attsL_min
```

6.3.5 Comparison with Heuristics for Vertical Partitioning

A comprehensive comparison of vertical partitioning methods is given in [44]. With few exceptions [56, 58], vertical partitioning algorithms consider only the non-replicated case. When replication is considered, it is only partial replication. The bounded scenario – limited storage budget for replicated attributes – is discussed only in [58]. At a high level, vertical partitioning algorithms can be classified along several axes [44]. We discuss the two most relevant axes for the proposed heuristic. Based on the direction in which partitions are built, we have top-down and bottom-up algorithms. A top-down algorithm [5,22,56] starts with the complete schema and, at each step, splits it into two partitioned schemas. The process is repeated recursively for each resulting schema. A bottom-up algorithm [31, 33, 34, 45, 58] starts with a series of schemas, e.g., one for each attribute or one for each subset of attributes accessed in a query, and, at each step, merges a pair of schemas into a new single schema. In both cases, the process stops when the objective function cannot be improved further. A second classification axis is given by the granularity at which the algorithm works. An attribute-level algorithm [5, 31, 33, 34, 45, 56, 58] considers the attributes independent of the queries in which they appear. The interaction between attributes across queries still plays a significant role, though. A query or transaction-level algorithm [22] works at query granularity. A partition contains either all or none of the attributes accessed in a query.

Based on the classification of vertical partitioning algorithms, the proposed heuristic qualifies primarily as a top-down query-level attribute-level algorithm. However, the recursion is only one-level deep, with the loaded partition at the bottom. The partitioning process consists of multiple steps, though. At each step, a new partition extracted from the raw data is merged into the loaded partition—similar to a bottom-up algorithm. The query coverage algorithm gives the query granularity characteristic to the proposed heuristic, while attribute usage frequency provides the attribute-level property. Overall, the proposed heuristic combines ideas from several classes of vertical partitioning algorithms, adapting their optimal behavior to raw data processing with partial loading. An experimental comparison with specific algorithms is presented in the experiments (Section 7.6) and a discussion on their differences in the related work (Section 3).

6.4 Pipeline Processing

In this section, we discuss on the feasibility of MIP optimization in the case of pipelined raw data processing with partial loading. We consider a super-scalar pipeline architecture in which raw data access and the extraction stages – tokenize and parse – can be executed concurrently by overlapping disk I/O and CPU processing. This architecture is introduced in [20], where it is shown that, with a sufficiently large number of threads, raw data processing is an I/O-bound task. Loading and accessing data from the processing representation are not considered as part of the pipeline since they cannot be overlapped with raw data access due to I/O interference. We show that, in general, pipelined raw data processing with partial loading cannot be modeled as a linear MIP. However, we provide a linear formulation for a scenario that is common in practice, e.g., binary FITS and JSON format. In these cases, tokenization is atomic. It is executed for all or none of the attributes. This lets parsing as the single variable in the extraction stage. The MIP formulation cannot be solved efficiently, due to the large number of variables and constraints—much larger than in the sequential

formulation. We handle this problem by applying a simple modification to the heuristic introduced in Section 6.3 that makes the algorithm feasible for pipelined processing.

6.4.1 MIP Formulation

Since raw data access and extraction are executed concurrently, the objective function corresponding to pipelined query processing has to include only the maximum of the two:

$$T_i^{pipe} = |R| \cdot \sum_{j=1}^{n} read_{ij} \cdot \frac{SPF_j}{band_{IO}} + \max\left\{ raw_i \cdot \frac{S_{RAW}}{band_{IO}}, |R| \cdot \sum_{j=1}^{n} \left(t_{ij} \cdot T_{t_j} + p_{ij} \cdot T_{p_j} \right) \right\} \tag{6.4}$$

This is the only modification to the MIP formulation for sequential processing given in Section 6.2. Since the max function is non-linear, solving the modified formulation becomes impossible with standard MIP solvers, e.g., CPLEX[6], which work only for linear problems. The only alternative is to eliminate the max function and linearize the objective. However, this cannot be achieved in the general case. It can be achieved, though, for specific types of raw data—binary formats that do not require tokenization, e.g., FITS, and text formats that require complete tuple tokenization, e.g., JSON. As discussed in the introduction, these formats are used extensively in practice.

Queries over raw data can be classified into two categories based on the pipelined objective function in Eq. (6.4). In I/O-bound queries, the time to access raw data is the dominant factor, i.e., max returns the first argument. In CPU-bound queries, the extraction time dominates, i.e., max returns the second argument. If the category of the query is known, max can be immediately replaced with the correct argument and the MIP formulation becomes linear. Our approach is to incorporate the category of the query in the optimization as 0/1 variables. For each query i, there is a variable for CPU-bound (cpu_i) and one for IO-bound (io_i). Only one of them can take value 1. Moreover, these variables have to be paired with the variables for raw data access and extraction, respectively. Variables of the form $cpu.raw_i$, $cpu.t_{ij}$, and $cpu.p_{ij}$ correspond to the variables in Table 6.2, in the case of a CPU-bound query. Variables $io.raw_i$, $io.t_{ij}$, and $io.p_{ij}$ are for the IO-bound case, respectively.

With these variables, we can define the functional **constraints** for pipelined raw data processing:

$$\begin{aligned} C_7 &: cpu_i + io_i = 1;\ i = \overline{1,m} \\ C_{8-10} &: cpu.x + io.x = x;\ x \in \{raw_i, t_{ij}, p_{ij}\} \\ C_{11-13} &: cpu.x \leq cpu_i;\ i = \overline{1,m} \\ C_{14-16} &: io.x \leq io_i;\ i = \overline{1,m} \end{aligned} \tag{6.5}$$

Constraint C_7 forces a query to be either CPU-bound or IO-bound. Constraints C_{8-10} tie the new

[6]http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

family of CPU/IO variables to their originals in the serial formulation. For example, the raw data is accessed in a CPU/IO query only if it is accessed in the stand-alone query. The same holds for tokenizing/parsing a column j in query i. Constraints C_{11-13} and C_{14-16}, respectively, tie the value of the CPU/IO variables to the value of the corresponding query variable. For example, only when a query i is CPU-bound, it makes sense for $cpu.t_{ij}$ and $cpu.p_{ij}$ to be allowed to take value 1. If the query is IO-bound, $io.t_{ij}$ and $io.p_{ij}$ can be set, but not $cpu.t_{ij}$ and $cpu.p_{ij}$.

At this point, we have still not defined when a query is CPU-bound and when is IO-bound. This depends on the relationship between the time to access the raw data and the time to extract the referenced attributes. While the parsing time is completely determined by the attributes accessed in the query, the tokenizing time is problematic since it depends not only on the attributes, but also on their position in the schema. For example, in the SDSS `photoPrimary` table containing 509 attributes, the time to tokenize the 5^{th} attribute is considerably smaller than the time to tokenize the 205^{th} attribute. Moreover, there is no linear relationship between the position in the schema and the tokenize time. For this reason, we cannot distinguish between CPU- and IO-bound queries in the general case. However, if there is no tokenization – the case for binary formats such as FITS – or the tokenization involves all the attributes in the schema – the case for hierarchical JSON format – we can define a threshold $PT = \left\lfloor \frac{\frac{S_{RAW}}{band_{IO}} - |R| \cdot \sum_{j=1}^{n} T_{t_j}}{\frac{|R| \cdot \sum_{j=1}^{n} T_{p_j}}{n}} \right\rfloor$ that allows us to classify queries. PT is given by the ratio between the time to access raw data less the constant tokenize time and the average time to parse an attribute. Intuitively, PT gives the number of attributes that can be parsed in the time required to access the raw data. If a query has to parse more than PT attributes, it is CPU-bound. Otherwise, it is IO-bound. The threshold **constraints** C_{17} and C_{18} make these definitions formal:

$$\begin{aligned} C_{17} &: \sum_{j=1}^{n} p_{ij} - PT < cpu_i \cdot n;\ i = \overline{1,m} \\ C_{18} &: PT - \sum_{j=1}^{n} p_{ij} \leq io_i \cdot n;\ i = \overline{1,m} \end{aligned} \tag{6.6}$$

For the atomic tokenization to hold, constraint C_5 in the serial formulation has to be replaced with $t_{ij} = t_{ik};\ i = \overline{1,m},\ j,k = \overline{1,n-1}$.

The complete pipelined MIP includes the constraints in the serial formulation (Eq. (6.1)) and the constraints C_{7-18}. The linear **objective function** corresponding to query processing is re-written using the newly introduced variables as follows:

$$\begin{aligned} T_i = & io.raw_i \cdot \frac{S_{RAW}}{band_{IO}} + |R| \cdot \sum_{j=1}^{n} read_{ij} \cdot \frac{SPF_j}{band_{IO}} + \\ & |R| \cdot \sum_{j=1}^{n} \left(cpu.t_{ij} \cdot T_{t_j} + cpu.p_{ij} \cdot T_{p_j} \right) \end{aligned} \tag{6.7}$$

6.4.2 Heuristic Algorithm

Since the number of variables and constraints increases with respect to the serial MIP formulation, the task of a direct linear solver becomes even harder. It is also important to notice that the problem remains NP-hard and cannot be approximated since the reduction to the k-element cover still applies. In these conditions, heuristic algorithms are the only solution. We design a simple modification to the heuristic introduced in Section 6.3, specifically targeted at pipelined raw data processing.

Given a configuration of attributes loaded in processing representation, the category of a query can be determined by evaluating the objective function. What is more important, though, is that the evolution of the query can be traced precisely as attributes get loaded. An I/O-bound query remains I/O-bound as long as not all of its corresponding attributes are loaded. At that point, it is not considered by the heuristic anymore. A CPU-bound query has the potential to become I/O-bound if the attributes that dominate the extraction get loaded. Once I/O-bound, a query cannot reverse to the CPU-bound state. Thus, the only transitions a query can make are from CPU-bound to I/O-bound, and to loaded from there. If an IO-bound query is not covered in the ***Query coverage*** section of the heuristic, its contribution to the objective function cannot be improved since it cannot be completely covered by ***Attribute usage frequency***. Based on this observation, the only strategy to reduce the cost is to select attributes that appear in CPU-bound queries. We enforce this by limiting the selection of the attributes considered in line 3 of ***Attribute usage frequency*** to those attributes that appear in at least one CPU-bound query.

6.5 Experimental Evaluation

The objective of the experimental evaluation is to investigate the accuracy and performance of the proposed heuristic across a variety of datasets and workloads executed sequentially and pipelined. To this end, we explore the accuracy of predicting the execution time for complex workloads over three raw data formats—CSV, FITS, and JSON. Additionally, the sensitivity of the heuristic is quantified with respect to the various configuration parameters. Specifically, the experiments we design are targeted to answer the following questions:

- What is the impact of each stage in the overall behavior of the heuristic?
- How accurate is the heuristic with respect to the optimal solution? With respect to vertical partitioning algorithms?
- How much faster is the heuristic compared to directly solving the MIP formulation? Compared to other vertical partitioning algorithms?
- Can the heuristic exploit pipeline processing in partitioning?
- Do the MIP model and the heuristic reflect reality across a variety of raw data formats?

Implementation. We implement the heuristic and all the other algorithms referenced in the paper in `C++`. We follow the description and the parameter settings given in the original paper as closely as possible. The loading and query execution plans returned by the optimization routine are executed with the SCANRAW [20] operator for raw data processing. SCANRAW supports serial and pipelined execution. The real results returned by SCANRAW are used as reference. We use IBM CPLEX 12.6.1 to implement and solve the MIP formulations. CPLEX supports parallel

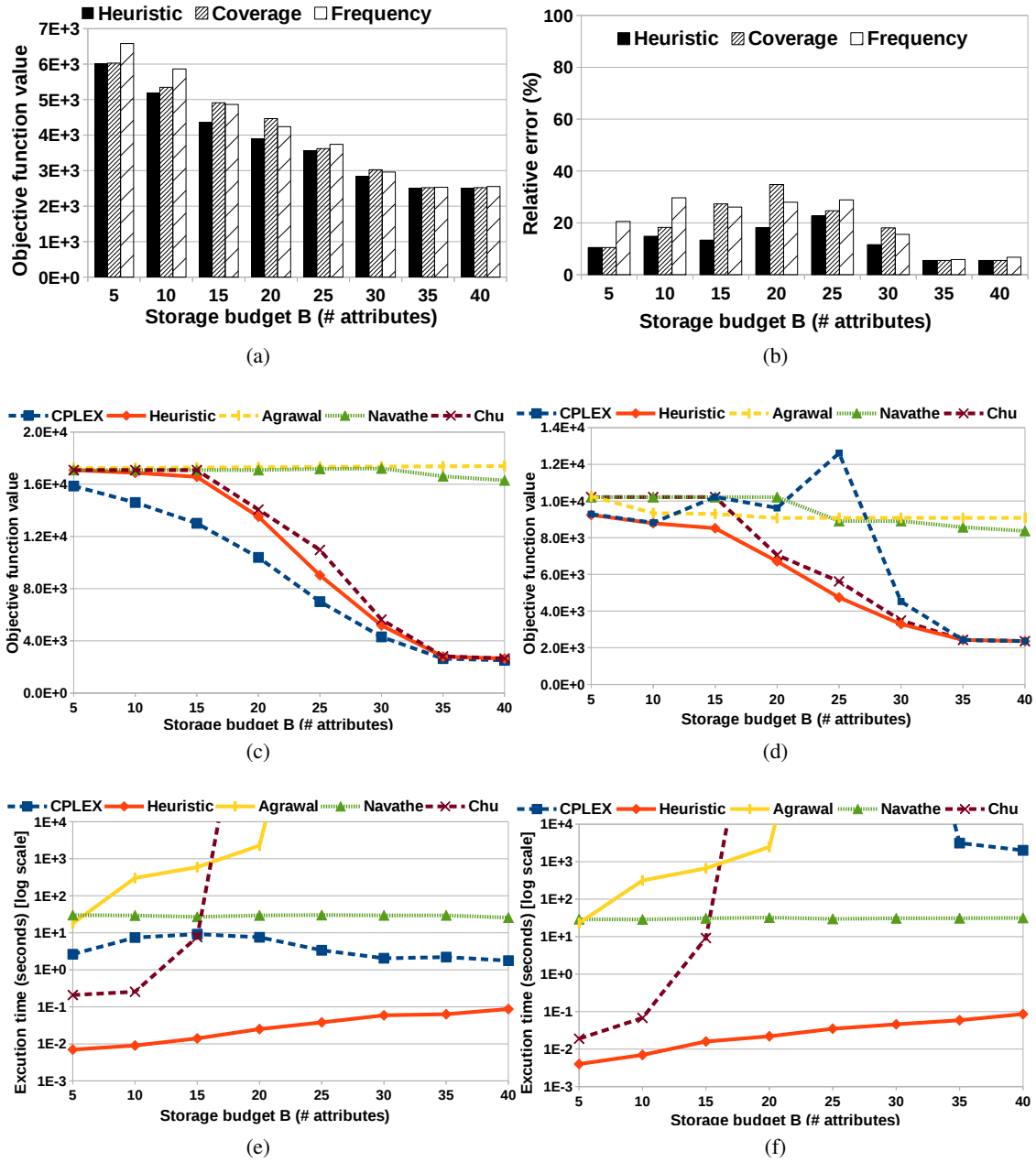


Figure 6.1: Experimental results for the heuristic algorithm. Comparison between the stages: (a) objective function value; (d) relative error with respect to the optimal solution. Comparison with CPLEX and vertical partitioning algorithms in objective function value (b,c) and execution time (e,f) for serial (b,e) and pipelined (c,f) raw data processing.

processing. The number of threads used in the optimization is determined dynamically at runtime.

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. The cached and buffered read rates are 3 GB/second and 565 MB/second, respectively. Ubuntu 14.04.2 SMP 64-bit with Linux kernel 3.13.0-43 is the operating system.

Methodology. We perform all experiments at least 3 times and report the average value as the result. We enforce data to be read from disk by cleaning the file system buffers before the execution of every query in the workload. This is necessary in order to maintain the validity of the modeling parameters.

Data. We use three types of real data formats in our experiments—CSV, FITS, and JSON. The CSV and FITS data are downloaded from the SDSS project using the CAS tool. They correspond to the complete schema of the `photoPrimary` table, which contains 509 attributes. The CSV and FITS data are identical. Only their representation is different. CSV is delimited text, while FITS is in binary format. There are 5 million rows in each of these files. CSV is 22 GB in size, while FITS is only 19 GB. JSON is a lightweight semi-structured key-value data format. The Twitter API provides access to user tweets in this format. Tweets have a hierarchical structure that can be flattened into a relational schema. We acquire 5,420,000 tweets by making requests to the Twitter API. There are at most 155 attributes in a tweet. The size of the data is 19 GB.

Workloads. We extract a real workload of 1 million SQL queries executed over the SDSS catalog in 2014. Out of these, we select the most popular 100 queries over table `photoPrimary` and their corresponding frequency. These represent approximately 70% of the 1 million queries. We use these 100 queries as our workload in the experiments over CSV and FITS data. The weight of a query is given by its relative frequency. Furthermore, we extract a subset of the 32 most popular queries and generate a second workload. The maximum number of attributes referenced in both workloads is 74. We create the workload for the tweets data synthetically since we cannot find a real workload that accesses more than a dozen of attributes. The number of attributes in a query is sampled from a normal distribution centered at 20 and having a standard deviation of 20. The attributes in a query are randomly selected out of all the attributes in the schema or, alternatively, out of a subset of the attributes. The smaller the subset, the more attributes are not accessed in any query. The same weight is assigned to all the queries in the workload.

6.5.1 Micro-Benchmarks

In this set of experiments, we evaluate the sensitivity of the proposed heuristic with respect to the parameters of the problem, specifically, the number of queries in the workload and the storage budget. We study the impact each stage in the heuristic has on the overall accuracy. We measure the error incurred by the heuristic with respect to the optimal solution computed by CPLEX and the decrease in execution time. We also compare against several top-down vertical partitioning algorithms. We use the SDSS data and workload in our evaluation.

We consider the following vertical partitioning algorithms in our comparison: Agrawal [5],

Navathe [56], and Chu [22]. The same objective function (Eq. (6.1)) is used across all of them. A detailed description of our implementation can be found in the extended version of the paper [75].

The *Agrawal algorithm* [5] is a pruning-based algorithm in which all the possible column groups are generated based on the attribute co-occurrence in the query workload. For each column group, an interestingness measure is computed. Since there is an exponential number of such column groups, only the "interesting" ones are considered as possible partitions. A column group is interesting if the interestingness measure, i.e., CG-Cost, is larger than a specified threshold. The interesting column groups are further ranked based on another measure, i.e., VP-Confidence, which quantifies the frequency with which the entire column group is referenced in queries. The attributes to load are determined by selecting column groups in the order given by VP-Confidence, as long as the storage budget is not filled. While many strategies can be envisioned, our implementation is greedy. It chooses those attributes in a column group that are not already loaded and that minimize the objective function, one-at-a-time.

The *Navathe algorithm* [56] starts with an affinity matrix that quantifies the frequency with which any pair of two attributes appear together in a query. The main step of the algorithm consists in finding a permutation of the rows and columns that groups attributes that co-occur together in queries. While finding the optimal permutation is exponential in the number of attributes, a quadratic greedy algorithm that starts with two random attributes and then chooses the best attribute to add and the best position, one-at-a-time, is given. These are computed based on a benefit function that is independent of the objective. The final step of the algorithm consists in finding a split point along the attribute axis that generates two partitions with minimum objective function value across the query workload. An additional condition that we have to consider in our implementation is the storage budget—we find the optimal partition that also fits in the available storage space.

The *Chu algorithm* [22] considers only those partitions supported by at least one query in the workload, i.e., a column group can be a partition only if it is accessed entirely by a query. Moreover, a column group supported by a query is never split into smaller sub-parts. The algorithm enumerates all the column groups supported by any number of queries in the workload – from a single query to all the queries – and chooses the partition that minimizes the objective function. The remaining attributes – not supported by the query – form the second partition. This algorithm is exponential in the number of queries in the workload $O(2^m)$. The solution proposed in [22] is to limit the number number of query combinations to a relatively small constant, e.g., 5. In our implementation, we let the algorithm run for a limited amount of time, e.g., one hour, and report the best result at that time—if the algorithm has not finished by that time.

Heuristic stage analysis. Figure 6.1a and 6.1b depict the impact each stage in the heuristic – query **coverage** and attribute usage **frequency** – has on the accuracy, when taken separately and together, i.e., **heuristic**. We measure both the absolute value (Figure 6.1a) and the relative error with respect to the optimal value (Figure 6.1b). We depict these values as a function of the storage budget, given as the number of attributes that can be loaded. We use the 32 queries workload. As expected, when the budget increases, the objective decreases. In terms of relative error, though, the heuristic is more accurate at the extremes—small budget or large budget. When the budget is medium, the error is the highest. The reason for this behavior is that, at the extremes, the number of choices for loading is considerably smaller and the heuristic finds a good enough solution. When the storage budget is medium, there are many loading choices and the heuristic makes only local

optimal decisions that do not necessarily add-up to a good global solution. The two-stage heuristic has better accuracy than each stage taken separately. This is more clear in the case of the difficult problems with medium budget. Between the two separate stages, none of them is dominating the other in all the cases. This proves that our integrated heuristic is the right choice since it always improves upon the best stage.

Serial heuristic accuracy. Figure 6.1c depicts the accuracy as a function of the storage budget for several algorithms in the case of serial raw data processing. The workload composed of 100 queries is used. Out of the heuristic algorithms, the proposed heuristic is the most accurate. As already mentioned, the largest error is incurred when the budget is medium. Between the vertical partitioning algorithms, the query-level granularity algorithm [22] is the most accurate. The other two algorithms [5, 56] do not improve as the storage budget increases. This is because they are attribute-level algorithms that are not optimized for covering queries.

Serial heuristic execution time. Figure 6.1e depicts the execution time for the same scenario as in Figure 6.1c. It is clear that the proposed heuristic is always the fastest, even by three orders of magnitude in the best case. Surprisingly, calculating the exact solution using CPLEX is faster than all the vertical partitioning algorithms almost in all the cases. If an algorithm does not finish after one hour, we stop it and take the best solution at that moment. This is the case for Chu [22] and Agrawal [5]. However, the solution returned by Chu is accurate—a known fact from the original paper.

Pipelined heuristic accuracy. The objective function value for pipelined processing over FITS data is depicted in Figure 6.1d. The same 100 query workload is used. The only difference compared to the serial case is that CPLEX cannot find the optimal solution in less than one hour. However, it manages to find a good-enough solution in most cases. The proposed heuristic achieves the best accuracy for all the storage budgets.

Pipelined heuristic execution time. The proposed heuristic is the only solution that achieves sub-second execution time for all the storage budgets (Figure 6.1f). CPLEX finishes execution in the alloted time only when the budget is large. The number of variables and constraints in the pipeline MIP formulation increase the search space beyond what the CPLEX algorithms can handle.

6.5.2 Case Study: CSV Format

We provide a series of case studies over different data formats in order to validate that the raw data processing architecture depicted in Figure 7.1 is general and the MIP models corresponding to this architecture fit reality. We use the implementation of the architecture in the SCANRAW operator [20] as a baseline. For a given workload and loading plan, we measure the cumulative execution time after each query and compare the result with the estimation computed by the MIP formulation. If the two match, this is a good indication that the MIP formulation models reality accurately.

The CSV format maps directly to the raw data processing architecture. In order to apply the MIP formulation, the value of the parameters has to be calibrated for a given system and a given input file. The time to tokenize T_{t_j} and parse T_{p_j} an attribute are the only parameters that require discussion. This can be done by executing the two stages on a sample of the data and measuring the average value of the parameter for each attribute. As long as accurate estimates are obtained, the

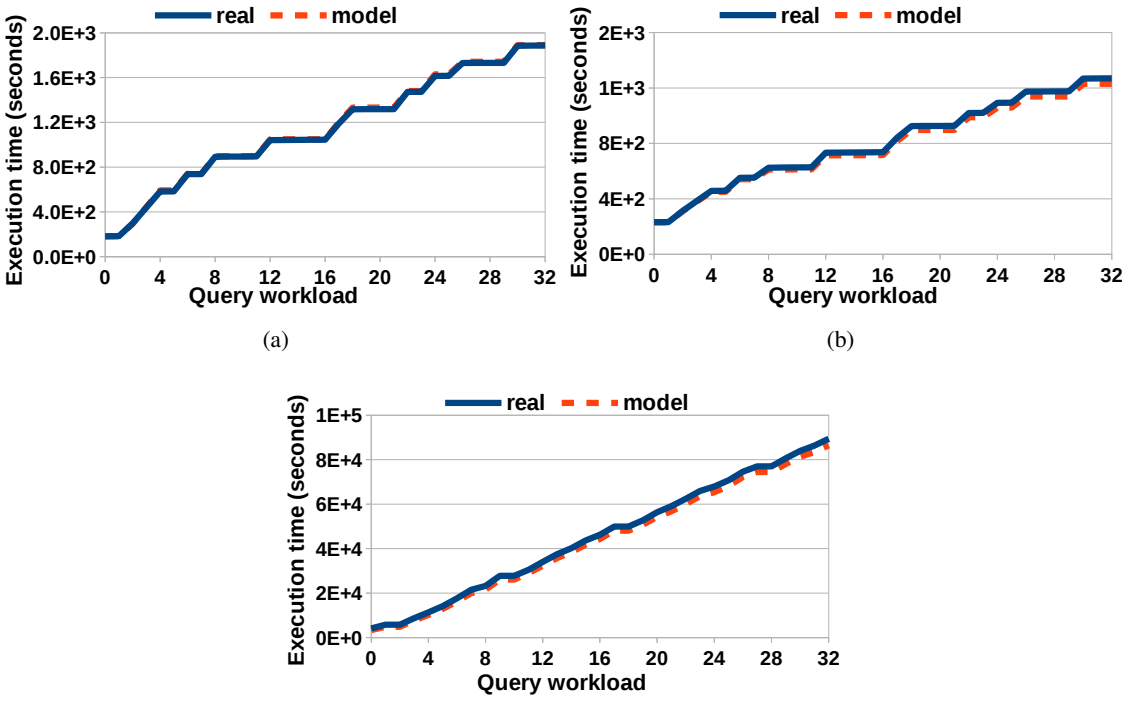


Figure 6.2: Model validation: (a) serial CSV, (b) serial FITS, (c) pipeline JSON.

model will be accurate. Figure 6.2a confirms this on the SDSS workload of 32 queries. In this case, there is a perfect match between the model and the SCANRAW execution.

6.5.3 Case Study: FITS Format

Since FITS is a binary format, there is no extraction phase, i.e., tokenizing and parsing, in the architecture. Moreover, data can be read directly in the processing representation, as long as the file access library provides such a functionality. CFITSIO[7] – the library we use in our implementation – can read a range of values of an attribute in a pre-allocated memory buffer. However, we observed experimentally that, in order to access any attribute, there is a high startup time. Essentially, the entire data are read in order to extract the attribute. The additional time is linear in the number of attributes. Based on these observations – that may be specific to CFITSIO – the following parameters have to be calibrated: the time to read the raw data corresponds to the startup time; an extraction time proportional with the number of attributes in the query is the equivalent of T_{p_j}. T_{t_j} is set to zero. Although pipelining is an option for FITS data, due to the specifics of the CFITSIO library, it is impossible to apply it. The result for the SDSS data confirms that the model is a good fit for FITS data since there is almost complete overlap in Figure 6.2b.

6.5.4 Case Study: JSON Format

At first sight, it seems impossible to map JSON data on the raw data processing architecture and the MIP model. Looking deeper, we observe that JSON data processing is even simpler than CSV processing. This is because every object is fully-tokenized and parsed in an internal map data structure, independent of the requested attributes. At least this is how the JSONCPP[8] library works. Once the map is built, it can be queried for any key in the schema. For schemas with a reduced number of hierarchical levels – the case for tweets – there is no difference in query time across levels. Essentially, the query time is proportional only with the number of requested keys, independent of their existence or not. Based on these observations, we set the model parameters as follows. T_{t_j} is set to the average time to build the map divided by the maximum number of attributes in the schema. T_{p_j} is set to the map data structure query time. Since T_{t_j} is a constant, the pipelined MIP formulation applies to the JSON format. The results in Figure 6.2c confirm the accuracy of the model over a workload of 32 queries executed in SCANRAW.

6.5.5 Discussion

The experimental evaluation provides answers to each of the questions raised at the beginning of the section. The two-stage heuristic improves over each of the component parts. It is not clear which of the query coverage and attribute usage frequency is more accurate. Using them together guarantees the best results. The proposed heuristic comes close to the optimal solution whenever the storage budget is either small or large. When many choices are available – the case for a medium budget – the accuracy decreases, but remains superior to the accuracy of the other vertical partitioning methods. In terms of execution time, the proposed heuristic is the clear winner—by as much as

[7] http://heasarc.gsfc.nasa.gov/fitsio/fitsio.html

[8] http://sourceforge.net/projects/jsoncpp/

three orders of magnitude. Surprisingly, CPLEX outperforms the other heuristics in the serial case. This is not necessarily unexpected, given that these algorithms have been introduced more than two decades ago. The case studies confirm the applicability of the MIP formulation model to several raw data formats. The MIP model fits the reality almost perfectly both for serial and pipelined raw data processing.

6.6 Conclusion

In this chapter, we study the problem of workload-driven raw data processing with partial loading. We model loading as binary vertical partitioning with full replication. Based on this equivalence, we provide a linear mixed integer programming optimization formulation that we prove to be NP-hard and inapproximable. We design a two-stage heuristic that combines the concepts of query coverage and attribute usage frequency. The heuristic comes within close range of the optimal solution in a fraction of the time. We extend the optimization formulation and the heuristic to a restricted type of pipelined raw data processing. In the pipelined scenario, data access and extraction are executed concurrently. We evaluate the performance of the heuristic and the accuracy of the optimization formulation over three real data formats – CSV, FITS, and JSON – processed with a state-of-the-art pipelined operator for raw data processing. The results confirm the superior performance of the proposed heuristic over related vertical partitioning algorithms and the accuracy of the formulation in capturing the execution details of a real operator.

Chapter 7

OLA-RAW – online processing over raw file

7.1 Introduction

Data exploration is the initial step in extracting knowledge from these data. Aggregate statistics are computed to assess the quality of the raw data before a thorough investigation on transformed data is performed. The primary goal of data exploration is to determine if the time-consuming data transformation and in-depth analysis are necessary. Thus, data exploration does not have to be exact. As long as *estimates* — guiding the decision process — is accurate enough, its goal is achieved, which allows for an extensive set of optimization strategies that reduce I/O and CPU utilization to be employed. However, if the detailed analysis is triggered, the work performed during exploration should allow for *incremental* extensions. To illustrate these concepts, we provide an example from a real application in astronomy.

Motivating example. The Palomar Transient Factory[1] (PTF) project [55] aims to identify and automatically classify transient astrophysical objects such as variable stars and supernovae in real-time. A list of potential transients – or candidates – is extracted from the images taken by the telescope during night. They are stored as a table in one or more FITS[2] files. The initial stage in the identification process is to execute a series of aggregate queries over the batch of extracted candidates, which is the equivalent of data exploration. The general SQL form of the queries is:

```
SELECT AGGREGATE(expression) AS agg
FROM candidate
WHERE select-condition
HAVING agg < threshold
```

where *AGGREGATE* is SUM, COUNT or AVERAGE and *threshold* is a verification parameter. These queries check certain statistical properties of the entire batch and are executed in sequence—

[1] www.astro.caltech.edu/ptf/

[2] http://heasarc.gsfc.nasa.gov/fitsio/

a query is only executed if all the previous queries are satisfied. If the candidate batch passes the verification criteria, an in-depth analysis is performed for individual candidates. The entire process – verification and in-depth analysis – is executed by querying a PostgreSQL[3] database—only after loading all candidates from the original FITS files. This workflow is highly inefficient for two reasons. First, before the data loading, the verification cannot start. Second, if the batch does not pass the verification, both the time spent for loading and the storage used for data replication are wasted.

Raw data processing. Several raw data processing systems have been recently introduced [3, 7, 20, 38, 42, 53] to reduce the high upfront database loading cost. They are extensions of the external table mechanism supported by standard database servers [54, 74]. These systems execute SQL queries directly over raw data while optimizing the conversion process into the format required by the query engine, which eliminates loading and provides instant access to data, i.e., verification can start immediately in our example. Querying directly on raw files, i.e., without loading, – is considered as a standard solution. Since it avoids the data-to-query cost and inherits the declarative query execution mechanism provided by SQL—data can be queried in the original format using SQL. Besides, several methods have been proposed for optimizing the raw file processing. They design two mainly methods ways to reduce the loading cost and speed up the following queries execution. One is eliminating unnecessary data and only processing concerning attributes. The other way is caching only useful data in memory [7, 42, 53] or utilize idle I/O resource to flush them into the database during query execution [3, 20]. After all necessary data being loaded into the system, running queries becomes as fast as executing them directly from the database.

Problem statement. We consider the problem of *efficiently producing estimation result during queries execution in-situ over raw files*. Only extend the classic OLA work on top of raw file process system is not suffice. Since to maximize the I/O utilization, raw file processing system usually sequentially scan the file to produce data, which cannot be directly used by classic OLA model. Our objective is to design a novel system –combining advantages from traditional OLA and in-situ data processing system– that provides instant access to data and also produce estimation result during query execution, besides the system should achieve optimal performance when the workload consists of a sequence of queries. There are two aspects to this problem. First, methods that provide sample-based query execution over raw files have to be developed. And second, a mechanism for query-driven continuous sample maintenance has to be devised. This mechanism interferes minimally – if at all – with standard query processing and guarantees to speed up the following queries.

Contributions. The major contributions we propose in this section is OLA-RAW—a novel system which supports OLA functionality and in-situ processing over raw files– that execute a query in an efficient way and get optimal performance across a query workload. OLA-RAW supports not only standard raw file process running complex analytical queries over terabytes of raw data very efficiently, but also it supports *overlap query processing with estimation*. Unlike the traditional OLA system which assumes to generate data in a "random order", OLA-RAW begins with raw data files and has to design an efficient mechanism to produce tuples in a random order by itself. Another aspect that differentiates OLA-RAW from other online aggregation systems is OLA-RAW is a query-driven operator that could dynamically change its I/O plan into an efficient way to access

[3] http://www.postgresql.org/

the raw file. Finally, it aims to speed up the traditional raw data processing not only for the first query but also for a sequence of queries by selectively preserving processed samples in a smart way.

Our specific contributions can be summarized as follows:

- We design `OLA-RAW`, a resource aware parallel super-scalar pipeline system integrates OLA with in-situ processing over raw data.
- We propose a proper sampling method to efficient generate "random" order from raw files.
- We implement `OLA-RAW` in the Scanraw [20] multi-threaded database system and evaluate its performance across a variety of synthetic and real-world datasets. Our results show that `OLA-RAW` achieves better performance than normal adaptive loading for a query sequence at any point in the processing.

7.2 PRELIMINARIES

In this section, we introduce query processing over raw data and online aggregation, respectively. We discuss their characteristics and summarize their optimization goals. We define the online aggregation over raw data problem and identify the challenges that have to be addressed by a unified solution that integrates the two.

7.2.1 Query Processing over Raw Data

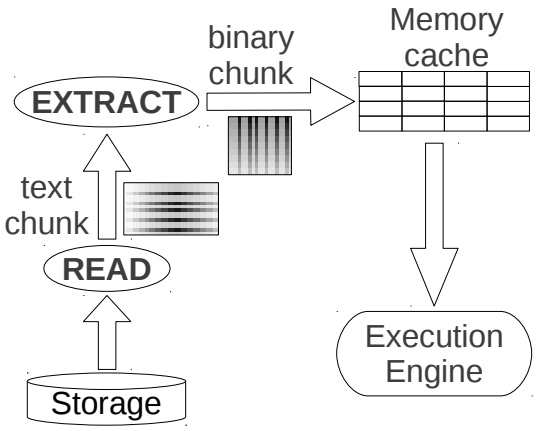


Figure 7.1: Query processing over raw data.

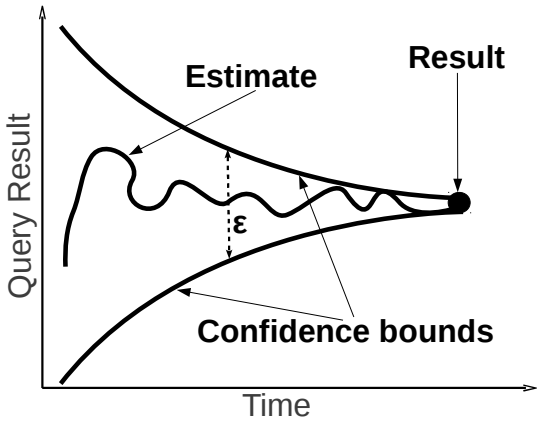


Figure 7.2: Online aggregation.

Query processing over raw data is depicted in Figure 7.1. The input to the process is a raw file from a non-volatile storage device, e.g., disk or SSD, a schema that can include optional attributes, 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 query engine and, possibly, is materialized (i.e., loaded) on the same storage device. In the `READ` stage, data are read from the original raw file, page-by-page, using the file system's functionality. 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. `EXTRACT` transforms tuples from the raw format into the

processing representation based on the schema provided and using the extraction procedure given as input to the process. There are two main tasks in `EXTRACT`. The first is to identify the schema attributes and output a vector containing the starting position for every attribute in the tuple—or a subset if the query does not access all the attributes. Second, attributes are converted from the raw format to their corresponding binary type and mapped to the processing representation of the tuple—the record in a row-store, or the array in column-stores, respectively. Multiple tuples or column arrays are grouped into a chunk—the unit of processing. At the end of `EXTRACT`, data are loaded into memory and ready for query processing. Multiple paths can be taken at this point. In standard database loading, data are first written to the database and only then query processing starts. In external tables [54, 74], data are passed to the query engine and discarded afterwards. In NoDB [7,38] and in-memory databases [42,53], data are kept in memory for subsequent processing. SCANRAW [20] overlaps the I/O operations with `EXTRACT` over multiple chunks in a super-scalar pipeline architecture. Moreover, the interaction between `READ` and `WRITE` is carefully scheduled to avoid interference.

The existing solutions have severe limitations when applied to data exploration. Standard databases require data loading even for a single query over a raw file—the case in many data exploration tasks. Although external tables avoid loading, the system has to access the entire data for every query, which causes poor performance over a sequence of interesting queries. NoDB, invisible loading [3], and SCANRAW are query-driven in-situ processing systems that improve their performance gradually and converge to the database execution time by loading all the data. However, they are *data-agnostic* and cannot identify uninteresting patterns early in the processing which results in wasted CPU and storage resources.

7.2.2 Online Aggregation

The main idea in online aggregation (OLA) is to compute only an estimate of the query result (Figure 7.2) based on a sample of the data [35]. To provide any useful information, though, the estimation is required to be accurate and statistically significant. Different from one-time estimation [23, 28] that might produce very inaccurate estimates for arbitrary queries, OLA is an iterative process which produces a series of estimators with improving accuracy. This iterative process is accomplished by including more data in estimation, i.e., increasing the sample size, from one iteration to another. As the sample size increases, the accuracy of the estimator improves accordingly. For this to be true, though, data are required to be handled in a statistically meaningful order, i.e., random order, to allow for the definition and analysis of the estimator. The user can decide to stop the query or to run a subsequent iteration based on the accuracy ϵ, i.e., confidence bounds width, of the estimator. As long as the bounds shrink fast enough, the time to execute the entire process is expected to be shorter than computing the exact result over the whole dataset.

The existing OLA solutions for data exploration have strict requirements imposed by the sampling procedure. The samples can be generated offline or online. In the case of offline sampling – the process similar to loading – it is impossible to determine the appropriate sample size to satisfy the accuracy requirement of all possible queries. BlinkDB [62] proposes a solution in which several samples with progressively increasing sizes – the largest one being the entire dataset – are taken. Moreover, since offline sampling is *query-agnostic*, all the columns in the dataset are included in the

sample. Tuple-level online sampling is extremely inefficient because of the many random accesses to data it incurs. According to to [32], if the sample ratio is larger than 4% and there are at least 100 tuples per page, all the pages have to be accessed. Two solutions address this issue. One is data shuffling [1,61,63] generates a permutation of the data as a preprocessing step such that a sequential scan at runtime results in random samples of increasing size. The first solution is similar to creating a second copy of the data and incurs significant processing time—even more than loading. Shuffling is also query-agnostic. The other is page-level (or block-level) online sampling samples over the page (block) space instead of the tuple space. While this reduces the number of pages that have to be accessed, all the tuples inside a page have to be included in the estimator. Moreover, it is known that – for the same sampling ratio – page-level sampling has worse accuracy than tuple-level sampling [32].

7.2.3 Problem Statement

Given a parallel in-situ processing system and an aggregate query with accuracy level ϵ, our goal is to minimize the amount of data we process to compute the query result within the required accuracy. A second objective is to balance system resource utilization dynamically and adaptively. We consider parallel in-situ data processing in a shared memory multi-core architecture where I/O operations overlap with extraction and several chunks can be processed concurrently, e.g., SCANRAW. In such an environment, we consider two scenarios. When processing is I/O-bound, the problem reduces to minimizing the amount of data that have to be read from disk and identifying useful computation for which to use the spare CPU cycles. When processing is CPU-bound, the amount of extracted data has to be minimized and utilize available I/O bandwidth in a meaningful way. Given these, the end goal is to *minimize the amount of processed data while maximizing resource utilization during the processing time*.

Challenges. We consider sampling-based online aggregation in the context of parallel in-situ data processing. This technical problem poses a series of challenges both at the system level as well as regarding estimation. The first challenge is to devise an efficient parallel sampling strategy over raw data. The plan cannot increase the read time tremendously compared to external table processing. It also has to work correctly when several portions of data are extracted concurrently to avoid the so-called "inspection paradox" [57]—a common problem across all the holistic, parallel sampling methods. We have to be able to define a sound estimator over the extracted sample and derive confidence bounds that allow for the correct characterization of accuracy. We have to design suitable algorithms for the efficient computation of the estimator and their confidence bounds. The second challenge is how to store and maintain the sample so that we can reuse it for subsequent queries—by using only the spare resources available during processing. The third challenge is to monitor resource utilization at runtime and identify OLA-specific tasks for which to use the idle resources. The solution has to be integrated with the sampling procedure while preserving the correctness of the estimators.

7.2.4 Related Work

Two lines of research are most relevant to the work presented in this paper –raw data processing and online aggregation. Our contribution is producing online aggregation estimation in raw data

processing. To the best of our knowledge, this is the first paper to consider utilizing online aggregation to speed up raw data processing. Several researchers [6, 29, 36, 48, 66] have recently identified the need to reduce the analysis time for processing tasks operating over massive repositories of raw data. In-situ processing [50] has been confirmed as one promising approach. At a high level, in-situ data processing optimizations aim to enhance the existing traditional data processing systems that allows raw file processing inside the execution engine.

Various optimizations that eliminate the requirement for scanning the entire file to answer the query. **External table** [8, 54, 74], is supported by many modern database systems as a feature to directly analyze flat files using SQL without paying the upfront cost of loading the data into the system. **Adaptive partial loading** [38] avoids 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. **NoDB** [7] never loads data into the database. It always reads data from the raw file thus incurring the inherent overhead associated with tokenizing and parsing. A series of efficient techniques address these sources of overhead. Caching is used extensively to store data converted in the database representation in memory. **Data vaults** [42] applies 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. **Invisible loading** [3] extends adaptive partial loading and caching to MapReduce applications which operate natively over raw files in a distributed file system. The database stores the content of the raw file in binary format. It eliminates the inherent cost of tokenizing and parsing data for each query. **Instant loading** [53] proposes scalable bulk loading methods taking full advantage of modern super-scalar multi-core CPUs.

Online aggregate was first proposed in [35], for improving the response time of queries over raw files by presenting approximate results. It periodically produces approximate answers and allows users to get a quick sense of query result. Approaches that process queries on sampled data sets to provide fast query response times have to consider the trade-off between results accuracy and query performance. In such a setting, meeting user-defined error bounds is necessary to ensure reliable estimation results. A Bayesian framework based approach is used to implement **OLA over MapReduce** [57]. The approach considers the correlation between the aggregate value of each block and the processing time, takes into account the scheduling time and processing time of each block as observed data during the estimation processing. All of the blocks are logically ordered in a statistically random fashion and kept in a single queue. Sequentially taking out the data block from this queue equals randomly processing the data blocks. This strategy could speed up I/O operation associated with random tuple-level sampling. But the downside of data-block sampling is also evident. The accuracy of statistics built over a block-level sample depends on the layout of the data inside data block, i.e., the way tuples are generated. If there is a high dependency between the value of an attribute and the data block on which the corresponding row is located, block-level sampling would have lower precision on tuple-level sampling.

7.3 High Level Approach

In this section, we introduce our proposal and provide a high-level overview. Then we point out some major problems that we met, and also discuss existing solutions.

7.3.1 Our Proposal

Based on the description in section 7.2, we propose a new approach which aims to marry the merits of OLA and in-situ process, and design a novel OLA-based raw file processing system. The system should support in-situ access to the raw data without any advanced cost. Besides, its query execution performance should competitive with other OLA-based data processing system.

In the rest of this section, we identify the difficulties and query processing costs for the combination of in situ systems with OLA. The figure 7.3 depicts the target behavior. It illustrates an important property of our new system; the data-to-query time decreases because there is no need to shuffle the data in random order and loading it in advance. Also, performance is improved gradually as a function of the number of queries processed and finally converges to the OLA-based database system.

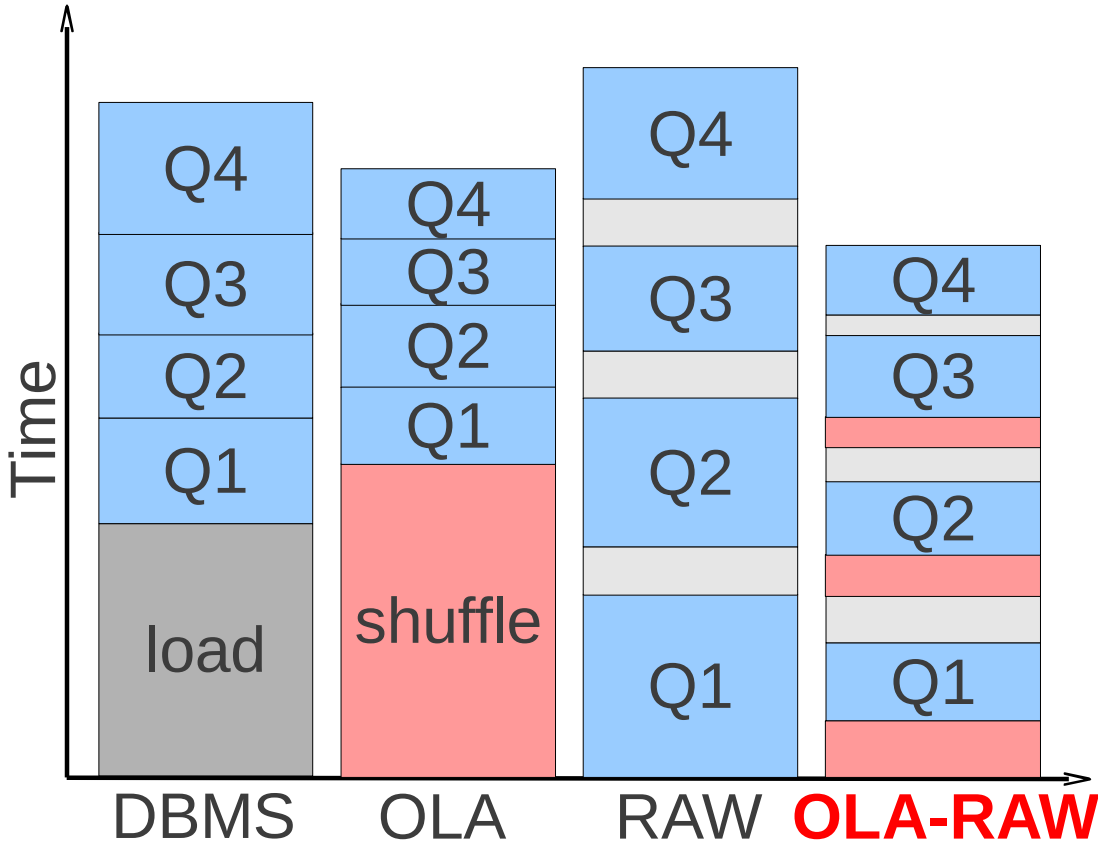


Figure 7.3: Intuition of OLA-RAW.

7.3.2 Key problems

Although modern database system successfully implements OLA inside, there is not a systematic and comprehensive OLA system directly on a raw file. The design and implementation of a

concrete instance have to solve the following problems.

- First, we need to find an efficiently way to generate random samples directly from a raw file. Since most OLA algorithm require that tuples in a relation be processed using a "random" ordering, where "random" has a very stringent mathematical definition. The cost of traditional method –shuffling tuples– on raw data is very high and even impossible in some cases. Since the system does not have any metadata about tuples in the raw file, it is impractical to generate a random tuple by random read or in memory random shuffle procedure.

- Second, we need to design an appropriate system architecture to integrate OLA calculation seamlessly with raw file process engine. The issue mainly about how to implement samples generation and corresponding estimator, and correctly incorporate into raw data processing, depicted in figure 7.1.

- Last, we need to find a feasible way to maintain extracted samples from raw files. The goal we aim at is that query performance could increase after finishing more and more queries. In addition, the samples should be continuously produced when the accuracy of result increase.

7.4 Sampling And Estimator

From the description in section 7.2.2, we know that sampling-based process is an effective solution for managing the massive data that arises in modern information management scenarios. The merits of sampling include reduction in the cost of retrieving and processing the data. In this section, we aim to find proper methods to generate random samples from raw files.

In this paper, we focus primarily on the problem of sampling from a single table T to obtain a quick approximate answer to an aggregation query of the form

$$\begin{aligned} &\text{SELECT } Op(expression) \text{ FROM } T \\ &\text{WHERE } predicator \end{aligned} \tag{7.1}$$

where Op is SUM and COUNT queries, while other aggregation such as AVG, VARIANCE and STD_DEV can be implemented through some extension, and expression is a numeric expression.

7.4.1 Standard Sampling

The first problem is how to design an efficient and feasible method for generating samples. In the following paragraphs, we introduce and compare several standard sampling methods, review their strengths and weaknesses respectively, briefly discuss some implementation issues, and finally choose a proper sampling schema that suits for in-situ data process.

7.4.1.1 Simple Random Sampling

Simple random sample (SRS) is the basic type of sampling, which is often used as a sampling technique itself or as a building block for more complex sampling methods.

Nearly all SRS implementation are tuple-level sampling method, which take each tuple as a sample unit. Denote the item set by S which contains N items: $s_1, s_2, ..., s_N$. Given a sample size $n \lg N$, let S_n be the set of subsets of S. According to [70], the sample mean $\bar{y} = \sum x_i$ is an unbiased estimator for population average μ. The variance s^2 is an unbiased estimator and defined as $s^2 = \frac{1}{n-1} \sum_{i=1}^{n} (y_i - \bar{y})$. An unbiased estimator of μ variance is $\widehat{Var}(\bar{y}) = (\frac{N-n}{N}) \frac{s^2}{n}$ and the square root of the variance of the estimator is its standard error. The unbiased estimator of population total τ is $N \times \bar{y}$, with variance $\hat{V}ar(\hat{\tau}) = N \times (N-n) \frac{s^2}{n}$.

Many researchers later improved this algorithm. They propose algorithms include select-rejection algorithm [27] and Reservoir [72]. Both are implemented based on sequentially scanning the whole data, which is the most efficient data access method from raw files. Despite the improvement, both algorithms have to scan the entire dataset again and again whenever they need to generate more samples during query process depicted in OLA process, which make performance poor and unacceptable.

7.4.1.2 Cluster Sampling

Cluster sampling [70](CS) is a probability sample in which takes each chunk as the sampling unit. For each selected chunk, all the tuples in the chunk are returned in the sample. The simple random sampling is applied on chunk-level. Given such a sampling process, most of the estimators of tuple-level SRS can be applied with only a slight modification. This is done as follows. Assume there are L chunks in the table, and each chunk contains M tuples. Let x_{ij} denote the value of the variable of interest of the jth tuple in ith chunk. The sum of the x values in the ith chunk is denoted simply as $x_i = \sum_{j=1}^{M} x_{ij}$. If we generate a sample with l chunks, then an unbiased estimator of the total value is $\frac{1}{l} \sum_{i=1}^{l} x_i$. The variance of $\bar{x}$ is $\widehat{\bar{x}} = \frac{L-l}{L} \frac{s_u^2}{l}$, where $s_u^2 = \frac{1}{l-1} \sum_{i=1}^{l} (x_i - \bar{x})^2$.

Compared to SRS, cluster sampling does not need to read the whole dataset, so it is faster than SRS. But cluster sample is no longer an SRS sample of the table. The accuracy of statistics built over a chunk level sample depends on the layout of the data inside the chunk, i.e., the way tuples are generated. If the layout is random, i.e., if there is no statistical dependence between the value of a tuple and the chunk in which it resides, a chunk-level sample may be just as good as a uniform random sample. However, if there is a high dependence on the value of an attribute and the chunk location, then chunk-level sampling has lower precision than tuple-level sampling. In other words, with the same sample size, block-based sampling is much faster, compared to row-based sampling method, but yields results that are often much less precise.

7.4.1.3 Stratified Sampling

In stratified sampling [70](SS), the population is partitioned into non-overlapping groups, called strata, and each stratum selects some samples respectively. Usually, samples are independently chosen from each stratum and finally gather together to produce the samples. The particular case where from each stratum a simple random sample is drawn is called a stratified random sample.

In our system, chunks are viewed as stratas. We denote n_h as the number of samples taken from stratum h. The total is from each stratum added up where $\hat{\tau_{sh}} = \sum_{h=1}^{L} \hat{\tau_h}$ is an unbiased estimator. Since selections in different stratum are independent, the variance is $\hat{V}ar(\hat{\tau_{st}}) = \sum_{h=1}^{L} M_h(M_h -$

$m_h)\frac{s_h^2}{n_h}$. And the estimator of average is $\hat{\mu_{sh}} = \frac{\hat{\tau_{sh}}}{N}$ with variance $\hat{V}ar(\hat{\mu_{st}}) = \frac{1}{N^2}\hat{V}ar(\hat{\tau_{st}})$.

Compared to previous two sampling methods, stratification may produce a smaller error of estimation than would be generated by a simple random sample of the same size. This result applies in particular if measurements within strata are very homogeneous. Besides, SS only needs to extract and analyze a portion of tuples for each chunk, so for a single chunk, SS could be much faster than CS especially when the tuple extraction is CPU-bounded.

However, due to the characteristic of the simple random sample, no matter how many tuples are selected from each chunk, the system has to read the whole chunk from the raw file. If the process is I/O bound, there is no cost difference for a single chunk between two different sampling methods. Besides, stratified sampling would generate samples for every data chunk, which means it has to scan the whole raw file and the total process time could be slower than cluster sampling especially when tuple extraction is I/O bound.

7.4.1.4 Discussion

For a given size of samples, cluster sampling is faster than global SRS and stratified sampling but often produces more variable estimates, and it is suitable when the layout of data is random. If the data are very homogeneous inside each chunk, SC could be more accurate than other two methods. According to OLA, when the system needs to generate more samples, cluster sampling only needs to read and process more new chunks until the final result is good enough. Since CS utilizes all data read from raw files, it is the most efficient method. Unlike SRS always scan the entire data set, stratified sampling could generate more samples from different chunks according to its variance value in various chunks. The sample size of each chunk is directly proportional to chunk size and its variance, i.e., choose more samples from the chunk having more tuples or more variace.

7.4.2 Bi-level uniform sampling

From the above discussion, it is clear that selecting a proper sampling method is a challenging task, which needs comprehensively to consider the layout of data and system utilization together. To provide better control over the tradeoff between speed, precision, and system resource utilization, we aim to combine the advantages from both cluster sampling and stratified sampling together to generate samples, which contains two stages. In the first stage, chunks are randomly sampled and denoted by SU1. Inside each sampled chunk, we randomly select tuples as second sampling units and denoted them as SU2. This method is called bi-level uniform sampling scheme [70].

Before analyzing the Bi-level uniform sampling into details, we denote assumption as following: Let N represents the number of chunks in the data set and M_i the number of tuples in the $Chunk_i$. Let y_{ij} denote the value of the variable of interest for the jth tuple in the $Chunk_i$. The total of the y-values in the $Chunk_i$ is $y_i = \sum_{j=1}^{M_i} y_{ij}$. The average value in $Chunk_i$ is $\mu_i = y_i/M_i$. The sum of the whole data set is $\tau = \sum_{i=1}^{N}\sum_{j=1}^{M_i} y_{ij}$. The average value per chunk is $\mu_1 = \tau/N$, while the average value per tuple is $\mu = \tau/M$, where $M = \sum_{i=1}^{N} M_i$ is the total number of tuples in the data set.

In Bi-level uniform sampling, the expected values and the variances of an estimator in the sampling world are calculated taking into consideration the two stages. We denote the E_1 and

E_2 to the expected value of the estimator among all possible two stages samples to be selected from the data. V_1 and V_2 refer to the sampling variance of the estimator. θ is an estimator of the whole dataset, the expected value of the estimator is: $E[\hat{\theta}] = E_1[E_2[\hat{\theta}]]$ and its sample variance is $V[\hat{\theta}] = V_1[E_2[\hat{\theta}]] + E_1[V_2[\hat{\theta}]]$. The first term relates to the sampling variance of the estimator between the chunks (SU1) and the second term relates to the sampling variance between the tuples (SU2) within the chunks (SU1).

7.4.2.1 Approximate Query Results

Let α be the expected aggregate value of query 7.1 when Op is average, sum or count, Y be the estimated value. Assume n random chunks have been selected, and for ith selected chunk, m_i tuples have been randomly sampled, where $1 \leq i \leq n$. Since simple random sampling is used to sample tuples inside each chunk. An unbiased estimator of the total y-value for the $Chunk_i$ in the sample is $\hat{y}_i = \frac{M_i}{m_i} \sum_{j=1}^{m_i} y_{ij}$. Besides, chunks are also selected through simple random sampling, an unbiased estimator of SUM over the dataset is $\hat{\tau} = \frac{N}{n} \sum_{i=1}^{n} \hat{y}_i$.

To estimate average, $\hat{\mu_1} = \hat{\tau}/N$ is an unbiased estimator of the *average sum* per chunk, and $\hat{\mu} = \hat{\tau}/M$ is unbiased estimator of average value among tuples in the dataset.

7.4.2.2 Precision of the Approximate Results

When using sampling to obtain quick approximate answers to aggregation queries, it is crucial to offer the precision of these answers. A common measure of precision for an unbiased sampling based estimate $\hat{\theta}$ of an unknown quantity θ is the standard error of $\hat{\theta}$, which is defined as the square root of the variance of $\hat{\theta}$.

$$
\begin{aligned}
&Var(\hat{\tau}) = N(N-n)\frac{s_u^2}{n} + \frac{N}{n}\sum_{i=1}^{n} M_i(M_i - m_i)\frac{s_i^2}{m_i} \\
&s_u^2 = \frac{1}{n-1}\sum_{i=1}^{n}(\hat{y}_i - \hat{\mu_1})^2 \ , \ s_i^2 = (\frac{1}{m_i - 1})\sum_{j=1}^{m_i}(y_{ij} - \bar{y_i})^2 \\
&\text{where } i = 1, ..., n, \ and \ \hat{\mu_1} = (1/n)\sum_{i=1}^{n}\hat{y}_i.
\end{aligned}
\tag{7.2}
$$

Equation 7.2 is an unbiased estimator of the variance of tuple sum, as mentioned above, it contains two parts. The first term describe the difference between selected chunks, while the second term contains the variance due to estimating the values y_i of each chunk from tuples sampled from it. The quantity s_u^2 is the sum variance among chunks, while s_i^2 is the sum variance within the $Chunk_i$. The variance of tuple average $\hat{\mu} = \hat{\tau}/M$ is the variance equation 7.2 divided by M^2.

7.5 Online Aggregation On Raw File

In this section, we discuss the design of our prototype, called OLA-RAW (**O**n**L**ine **A**ggregation for **Raw** file process) which brings online aggregation closer to raw data process to speed up the

query execution. As far as we know, *OLA-RAW* is the first system that supports online aggregation directly from the raw file. Estimates and corresponding confidence bounds for the query result are computed during the entire query execution process without incurring any noticeable overhead.

In the remaining of this section, we analyze the resource utilization during query process and point out the possible place for optimization. Then we show the architecture of *OLA-RAW*, describe the task of primary components and their unique characteristic. Also, we present the details of how to implement key elements, such as sample generation and result estimator adopted in *OLA-RAW*. Finally, we describe how to maintain and preserve the extracted samples to improve the query performance for the future queries. To illustrate the idea clearly, we assume that raw data is stored in comma-separated value (CSV) files.

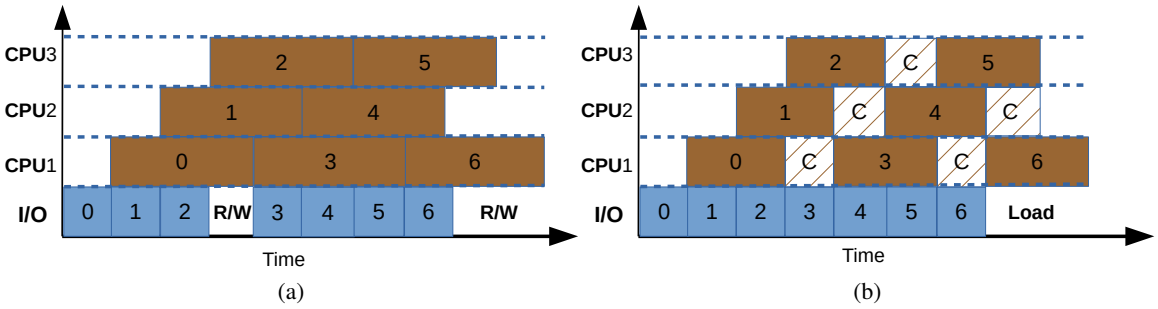


Figure 7.4: Analyze a query execution. (a) CPU-Bound Process. (b) I/O-Bound Process.

7.5.1 Quantify query process.

Query execution from either from the database or raw files, contains two sequential steps, one is reading necessary data from disks, called I/O operations and denoted as W_{io}, and the other is running computation on the reading data, which is called CPU operation indicated as W_{cpu}. Data partition and task parallelism are two mainly strategies to speed up the processing. Data is usually divided into small pieces, called chunk, and could be immediately processed by a worker thread. By doing this, two steps could overlap with each other which means the system utilize I/O and CPU components at the same time. Assume system I/O speed rate is denoted as S_{io} and computation rate is S_{cpu}, then the time to finish reading and computation would be $T_{io} = W_{io}/S_{io}$ and $T_{cpu} = W_{cpu}/S_{cpu}$ respectively. And query execution time is decided by the longer part $\max\{T_{io}, T_{cpu}\}$.

According to the relationship between T_{io} and T_{cpu}, query process can be mainly categorized into three types. When $T_{io} \approx T_{cpu}$, the process can be viewed as 'balanced', CPU computation entirely overlaps with I/O operation, system utilization is 100%, including I/O and CPU, all the time. But this is an ideal situation and cannot happen, since when the query starts, the system cannot do anything until thoroughly reading a full chunk. Figure 7.4 shows other two categories, one is I/O bound if $T_{io} > T_{cpu}$, and when $T_{io} < T_{cpu}$, the query is called CPU bound.

To comprehensively describe the problem, let's take a query, called $Q1$, as an example. We use the modern multi-threads system with 3 threads to execute $Q1$, besides it needs to access 6

chunks. Figure 7.4a shows workflow when $Q1$ is CPU-Bound. As we can see, the system always has spare CPU resources before reading chunks 5. The upper left region shows the available CPU resource. Assume the maximal number of chunks kept in memory at the same time is 3, then I/O cannot read more data until former chunks have been processed and dropped, so there will be idle I/O resource available, like holes showing in the figure between chunks 2 and chunk 3. At last, we can also find that after chunk 6 is read from disk, there are no I/O operations anymore, besides, after chunk 6 is finished, more and more CPU resource is available before query termination.

When the query is I/O-Bound 7.4b, the same situation happens at the beginning until all worker threads become busy. As we can see during query process, there is no additional I/O interval before reading all data. However, there are many CPU slots show up waiting for more data. This situation is even worse when process time is shorter than reading time for a single chunk. In this case, only one worker thread is enough for query execution and the rest workers are idle all the time.

In reality, most queries are either CPU bound or I/O bound, CPU and I/O cannot fully align well with each other. We aim to detect those extra system resource and expiate data process by assigning additional useful work, such as more computation when the process is I/O bound and extra read/write when execution is CPU bound, without affect final execution time.

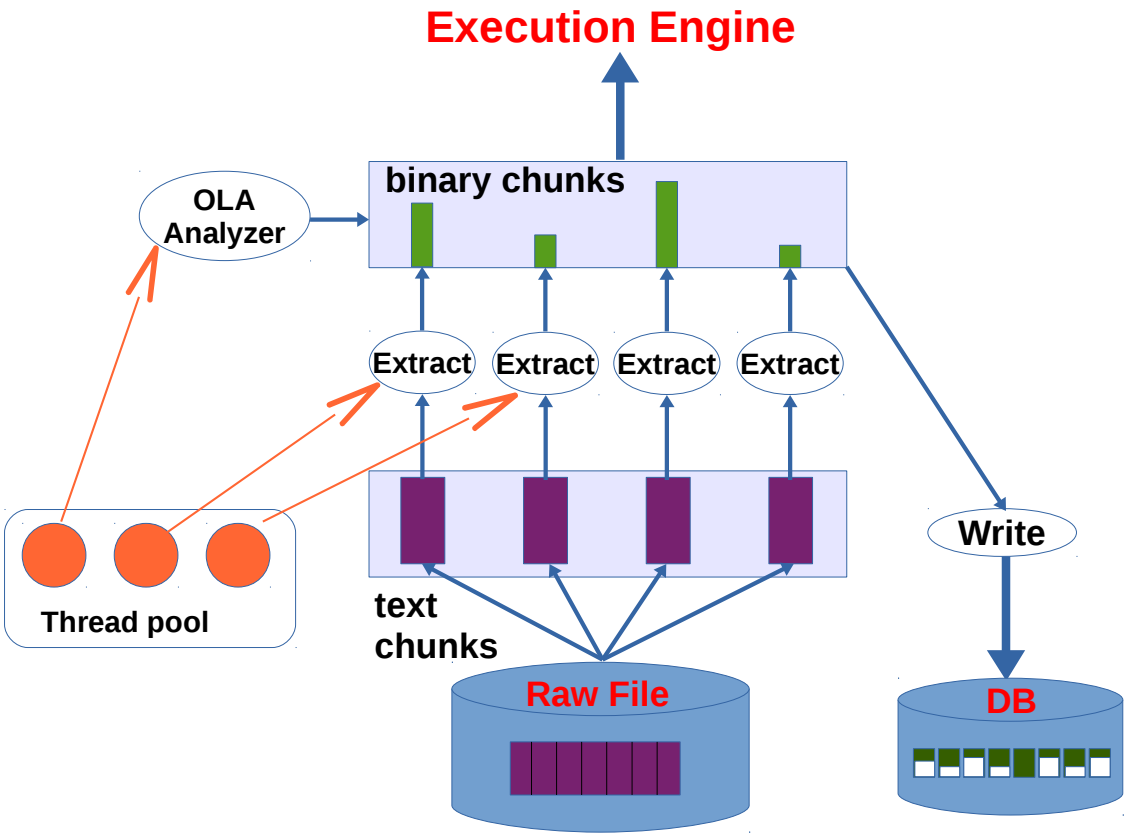


Figure 7.5: Architecture of OLA-RAW.

7.5.2 Architecture

Figure 7.5 describes the super-scalar pipeline architecture of *OLA-RAW*. Multiple *Extract* and *OLA-Analyzer* stages operate on different portions of the data in parallel, i.e., data partitioning parallelism [26]. The scheduler is responsible for controlling a pool of worker threads and assigning worker threads to different stages dynamically at runtime. *READ* and *WRITE* are also controlled by the scheduler thread to coordinate disk accesses optimally and avoid interference. *OLA-RAW* adopts the standard producer-consumer paradigm.

The *READ* stage of the process is responsible for reading data from the original flat file. It involves reading the lines of the file one-by-one and passing them to *EXTRACT*. Instead of sequentially generating data chunks, the *OLA-RAW* controls the *READ* to produce the data chunks in a specific order. As an optimization –already implemented by the file system– multiple lines co-located on the same page are read together. Thus, the file is logically split into horizontal portions containing a sequence of lines, i.e., chunks. Chunks represent the reading and processing unit.

EXTRACT has three main tasks: changing sequential read data into a random order, tokenizing and parsing data into binary format. To obtain random tuples from each chunk, the *EXTRACT*, following system-defined order, processes text tuple one by one and identifies attributes of each tuple. *EXTRACT* utilizes the selective tokenizing and parsing [7] to *reduce the size of workload*, and position buffer or position map [7] to *store the work done*. The output of each tuple is a vector containing the starting position for every attribute in the tuple. These vectors are continuously put together, whenever the size reach to some threshold, like 4096, then *EXTRACT* converts attribute from the text format into the binary representation corresponding to their type. This extraction typically involves the function that takes as input a string parameter and returns the attribute type, e.g. atoi. Besides, the *EXTRACT* adopts sampling theory to minimize the workload by periodically checking whether parsed tuples are enough to generate accurate approximation result. If the estimation is not sufficient, the *EXTRACT* continues to produce another group of binary tuples until the estimate is close enough. Then *EXTRACT* could either stop immediately or continue parsing. The action strategy is controlled by *OLA-RAW* and will be described in next subsection.

OLA-Analyzer gathers estimation result and statistic information for all chunks, and it would periodically utilize this chunk-based information to estimate the final result for the whole dataset. Write is a standalone thread which is in charge of flush data into the database.

7.5.3 Sampling and Estimator Implementation

In this subsection, we discuss the fundamental issue of *OLA-RAW* implementation. **How to generate random sample from raw file in an efficient way?**

7.5.3.1 Random Permutation

The primary sampling-related problem is how to generate random order, which defines the order we process the data. In our system, we choose the Knuth shuffle algorithm[4] to generate a permutation of n items uniformly at random without retries, which can be denoted as $P = \{p_i | 0 < i < n\}$.

[4] `https://en.wikipedia.org/wiki/Fisher-Yates_shuffle`

The discussion in sections 7.4 indicates that there are two types of permutations in *OLA-RAW*, one is chunk access permutation, which defines chunks access order, and the other is called tuple permutation, which is assigned to each data chunk. The tuple permutation is used to determine the order we process the tuples inside the chunk. For example, suppose a chunk contains 5 tuples, one possible permutation is $< 5, 3, 1, 2, 4 >$. Once a chunk gets its tuple permutation—stored in the system—it will be used for this data chunk in the future as well. By using the C++ standard function *std::srand* and *std::random_shuffle*, we can rebuild the permutation using the same seed. Therefore, the system could only record the seed instead of real permutation. The system builds a component called permutation manager which is in charge of generating and maintaining a set of independent permutations in the memory using a map structure, with seed value as key and concrete permutation as contend. *OLA-RAW* utilizes the spare CPU resource during start stage to initialize the permutations map without additional cost to the query execution.

7.5.3.2 Sampling

The main bottleneck of in-situ query processing is the access to raw data. The simple solution for this question is random read, which means every time the reader choose a random offset of the raw file, then reads and extracts a single tuple. Although random read could directly produce the sample without any other cost, it dramatically degrades I/O performance which makes it impractical. Instead of randomly retrieving single tuple at a time, *OLA-RAW* access a chunk of data –having hundreds even thousands– at each time. This optimization strategy enables the system to read text chunks at much faster speed. After reading a single chunk, one permutation p_i is randomly chosen from P and assigned to the chunk. For each chunk, we maintain its corresponding permutation in meta-data. After extracting a text chunk, tuples in memory are in the order of its corresponding permutation, instead of the original order in the raw file. Sequentially process the data from binary chunk equals continuously to extract samples from the original chunk in the raw file, by simple random samples without replacement algorithm.

7.5.3.3 Estimator

OLA-RAW implements a two-level result estimator to control the query processing based on the discussion in section 7.4. The first level is called chunk estimator which is integrated with EXTRACT stage, depicted in figure 7.5. It is used to approximate the result only based on belonging chunk data, in other words, the estimated result only describes the data distribution in a single chunk. The second level is query estimator that is implemented as independent component *OLA-Analyzer* in figure 7.5. *OLA-RAW* builds a status map to record the local estimation for processed chunks, based on which *OLA-RAW* could estimate the finally result with error bounds through equation 7.2. During query processing, *OLA-RAW* periodically predict the final result. If the approximation result is good enough, *OLA-RAW* immediately stop the current query process.

How many tuples are enough? Although the accuracy of estimated result increases as more and more samples being chosen and processed, the execution becomes longer as well. Equation (7.2), estimation error bound is decided by two parameters n –the number of chunks– and m_i, the number of selected tuples for each chunk. Increasing any of them reduces the error bound. Extracting too few samples out of each chunk cannot generate good enough estimation even all the

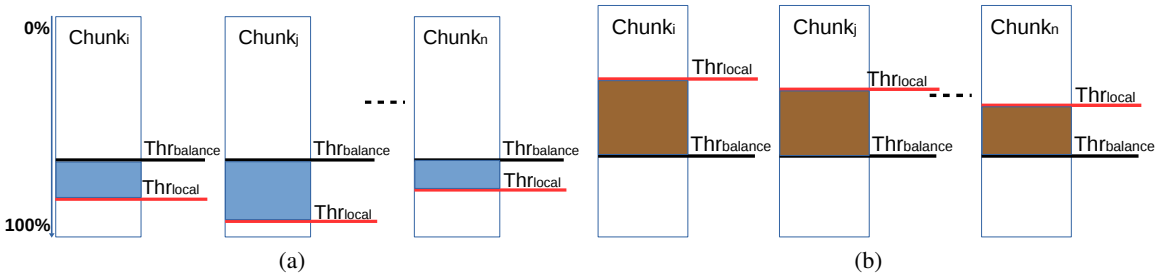


Figure 7.6: Threshold in chunk process. (a) CPU-Bound Process. (b) I/O-Bound Process.

chunks have been read and processed. It leads to sample regeneration from the raw file, taking the super high cost. On the opposite, processing too many samples for each chunk is not efficient either, especially when the processing is CPU-bound because it increases the processing time for each chunk.

To avoid these situations, we design a method to make sure *OLA-RAW* generates accurate estimation by accessing each chunk from raw file only once for any given query execution. We define a threshold Thr_{local}, with the red line in Figure 7.6, for each chunk, which is related to the accuracy of finally estimated result. This threshold is used to control how many samples have to be extracted from each chunk. This threshold makes sure the estimation based on samples inside each chunk is accurate enough that after processing the entire dataset the estimate is good enough. Since local data calculate the threshold, its value would be very different from each chunk. For example, assume the query is to measure the average value of attribute A in the data set with error bound 95%, then the Thr_{local} would be 95% as well. Tuples extraction continues until the accuracy of the local result is better than 95%. By doing this, we make sure enough size of the sample is produced to answer the question by only scan the whole dataset only once. So Thr_{local} can be viewed as the bottom line of parameter m_i during the query process.

7.5.4 Samples Maintenance

Sampling is an expensive operation, especially for raw file processing, due to the high cost to access raw data. Instead of sampling data from raw files for every query, we maintain the samples as a precomputed synopsis. After sample being used for processing the queries, we keep them in the system –in memory or database. For the following queries, the system could immediately generate an estimated result only based on stored samples. If the accuracy of estimation is good enough, the query execution immediately finishes without accessing any data from the raw file, which is an optimal situation. Otherwise, the system has to produce more new samples from raw file to improve the accuracy of estimated result. After the query finishing, newly generated samples is preserved accumulatively by the system. As executing more and more queries, at some point, the system can successfully produce accurate estimation directly based on loaded samples for most queries without accessing raw file anymore.

What kinds of samples should be preserved? The goal of sample maintenance is to speed

up the following queries execution, converge the resulting error as fast as possible. Compared to raw file procedure, directly getting samples from database or memory is super efficient. The natural choice would be kept all the processed data extracted for former queries executions. Due to the memory or I/O resource limitation, it is impossible to cache all of them in memory or flush them to disk without affect query execution. In most cases, the convergence speed of online aggregation estimation to the exact result is faster at the beginning and gradually decrease as sample size increasing. Therefore instead of taking the entire chunks as cache unit in traditional system [7, 20], or accessing all chunks in stratified sampling [70], *OLA-RAW* applies much more fine-grained cache policy that system only keeps the least amount of samples, maybe 10% of entire data. At the same time, *OLA-RAW* avoids accessing raw file more than once for the same query, and takes the Thr_{local} as the indicator, which means we need to store no less than Thr_{local} tuples from selected chunk.

When to load the samples? There are many options we could choose, such as flushing the samples during query processing, or cache them in the memory and loading after query execution. It is clear that directly writing data into the database during query processing would degrade the I/O operation and affect query performance. Due to the memory size limitation, post loading strategy is not an efficient solution, especially when data size is much larger than the memory size. *OLA-RAW* applies the speculative loading strategy [20], which utilizes the spare I/O resource during query process to store samples into the database without any performance cost, besides, whenever there is no I/O request for the running query, the system would flush samples in maximal speed.

How to make sure the complimentary samples have not been selected before? *OLA-RAW* can easily remember processed tuples and continuously to process new data. In the meta-data, every chunk has two pieces of information: permutation seeds and offset. Through permutation seeds, *OLA-RAW* can find or rebuild a determined random order. Utilizing offset, *OLA-RAW* could easily know how many tuples have been extracted and kept in the database. For example, considering there are 10 tuples in chunk i, denoted as $t_1, t_2, ..., t_{10}$, the random order key is S and offset is 3. When *OLA-RAW* execute a new query, at very beginning the random permutation is regenerated as $7, 9, 2, 5, 10, 1, 3, 4, 6, 8$, offset value indicates that tuple $7, 9$ and 2 could be read directly from the database. If the estimation based on loaded samples is not accurate enough, *OLA-RAW* could directly jump over the processed samples by offset and continues to produce new tuples immediately from correct position in its permutation order, starting from 5 in this example.

7.5.5 Query Processing

For any query, the system produces an estimation result based on the samples in the database. If the accuracy is good enough, the query would be immediately terminated; otherwise *OLA-RAW* has to access the raw file and extract more new samples until the estimated result is good enough. During queries processing, generated samples from the raw file will be stored in the database in an efficient way. Equation (7.3) express the cost models for *OLA-RAW* to execute a query.

$$\text{Cost(n, m)} \quad = \quad G(C_s) + F(\hat{n} \times C_{i/o}, \sum_{i=0}^{\hat{n}} m_i \times C_t) \tag{7.3}$$

$G(C_s)$ defines the cost to produce estimation from loaded samples. The rest parts indicates that reading additional $\hat{n}$ chunks with cost $\hat{n} \times C_{i/o}$ and processing m_i tuples of inside $chunk_i$ which costs

$\sum_{i=0}^{\hat{n}} m_i \times C_t$. The function F depends on whether two stages could happen in parallel. If the chunk read and extraction have to be sequentially run, F means the summation of two parts. But if the parallel mechanism is available, then F means choosing the maximal section from two parts.

7.5.5.1 More chunks or more tuples?

Selecting more chunks in the samples decreases the first part of equation 7.2, and extract more tuples from each chunk minimize the second part of equation 7.2 as well. Therefore, it is critical for us to determine the optimal values of n and m_i by which we could minimize the standard errors in the fastest way. During the query execution, *OLA-RAW* keeps all status information in memory which includes variances. *OLA-RAW* dynamically decides to access more chunks or tuples based on the standard variance of estimated result (7.2). If the variance of the first part is dominated, *OLA-RAW* accesses and processes more new chunks; otherwise, *OLA-RAW* extracts more tuples for each chunk, to minimize the variance inside the chunk.

When *OLA-RAW* decides to decrease the variance for the second part – parsing more samples –, it has to pick up the candidate from processed chunks. One solution is randomly choose one chunk by another which is simple. The other solution –*OLA-RAW* adopted– is to choose the candidates using their variance value in the decreasing order, which means the chunk with higher variance would be chosen early. This method aims to find the fastest way to decrease the variance, in sample theory, it tries to pick up the distinctive chunk compared to others.

7.5.5.2 Maximize system utilization

OLA-RAW adopts online aggregation methods into raw file processing to minimize both I/O and computation workload for in-situ data processing. In the following, we discuss how further to accelerate the query processing from system utilization aspect and try to make our procedure as a balanced process, which maximizes the system usage. We design another threshold, $Thr_{balance}$ and showed in Figure 7.6 as black lines, indicates that how many tuples can be extracted using the same amount of process time compared to reading a chunk from the raw file. As we know Thr_{local} is used to define the least number of tuples to be extracted. Based on the comparison between Thr_{local} and $Thr_{balance}$, the process can be categorized into three types and might affect our query plan.

CPU-bound process When $Thr_{balance}$ is less than Thr_{local}, the execution becomes CPU-bound. In this case, the system spends more time to transform tuples to reach the Thr_{local} than raw file chunk accessing time, indicating as the blue region in Figure 7.6a. In this situation, *OLA-RAW* immediately stop extracting tuples when it reaches on Thr_{local} threshold, where $\grave{m}_i = m_i$. Since analyzing much fewer data compared to standard raw file process, the query execution is much faster.

At the same time, the system has some spare I/O resource available which could be utilized to flush samples into the database to speed up future queries. In the optimal scenario, all the samples could be preserved in the database. However, if loading budget is limited, *OLA-RAW* prefers to flush chunks with higher Thr_{local}, since in CPU-bound condition the processing time is decided by computation. As showed in Figure 7.6a, $Chunk_j$ has highest load priority than $Chunk_i$ and $Chunk_n$. The system aims to store sample as much as possible.

I/O-bound process When $Thr_{balance}$ is greater than Thr_{local}, the execution is I/O-bound. In this case, the system has plenty of spare CPU resource available even after transforming enough tuples into binary format to reach the Thr_{local}, showed as the red region in figure 7.6b. Then more tuples could be processed until all the CPU resource is used or there are no tuples to be extracted, without increasing the final query time. We defined the number of extracted tuples as $\grave{m}_i$ then $\grave{m}_i > m_i$. Through equation (7.2), we can observe that as more and more tuples are processed the second part decreases. Compared to analyze to Thr_{local} threshold only, the number of required chunks, denoted as $\grave{n}$, to be processed will decrease as well, where $\grave{n} < n$. Since the necessary number of chunks decreases, the query time reduces as well.

For I/O-bounded query, there is no change to store the samples during query execution. *OLA-RAW* will flush the sample stored in memory after each query. Compared to CPU-bound query, the strategy to flush chunks is much different. The candidates of chosen order for chunks are based on their Thr_{local} in increase order. Since during I/O bound process, the query execution time is decided by chunks accessing time from the raw file. As showed in figure 7.6b, $Chunk_n$ would has high load priority than $Chunk_i$ and $Chunk_j$. By applying these optimizations, we could minimize the number of chunks that generated from raw files.

7.6 Experimental Evaluation

The objective of the experimental evaluation is to investigate the `OLA-RAW` 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 for many configuration parameters. Specifically, the experiments we design are targeted to answer the following questions:

- How does the `OLA-RAW` behave when the parallelism condition changes?
- How much data are processed to answer the user query and meets the result accuracy?
- What is the performance of `OLA-RAW` compared to external tables and database loading & processing?
- How does the dynamic `OLA-RAW` architecture handle various data characteristics and query characteristics?
- How is the resource utilization of the system?

Implementation. `OLA-RAW` is implemented as a `C++` prototype. Each stand-alone thread, as well as the workers, are performed as `pthread` instances. The code contains special function calls to harness detailed profiling data. In the experiments, we use `OLA-RAW` implementations for CSV and tab-delimited flat files, as well as FITS [11] files. To different data types, the performances of stages `EXTRACT` are very different. We integrate `OLA-RAW` with a state-of-the-art multi-thread in-situ data processing system [20] shown to be I/O-bound for a large class of queries. This integration guarantees that query processing is not the bottleneck except in rare situations and allows us to isolate the `OLA-RAW` behavior for detailed and accurate measurements.

System. All experiments are executed on a standard server with 2 AMD Opteron 6128 series 8-core processors (64 bit) – 16 cores – 40 GB of memory, and four 2 TB 7200 RPM SAS hard-drives configured RAID-0 in software. Each processor has 12 MB L3 cache while each core has 128 KB L1 and 512 KB L2 local caches. The cached and buffered read rates are 3 GB/second and 565

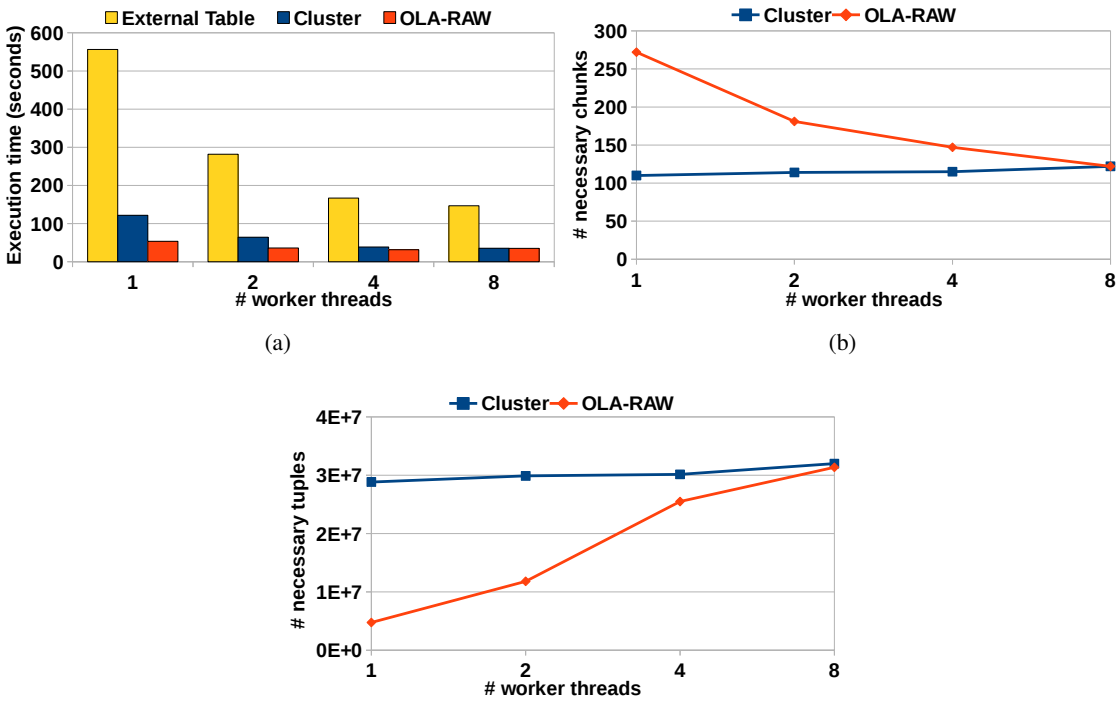


Figure 7.7: Execution time (a), necessary chunks (b), and tuples (c) as a function of the number of worker threads.

MB/second, respectively. Ubuntu 14.04.2 SMP 64-bit with Linux kernel 3.2.0-56 is the operating system.

Methodology. In the following experiments, we run benchmark queries with three methods – `OLA-RAW`, *SCANRAW-CLUSTER* – Cluster Sampling implemented on *SCANRAW* [20] and *External Table* – and compared their results. We perform all experiments at least three times and report the average value as the result. If the test consists of a single query, we always enforce to read data from the 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.

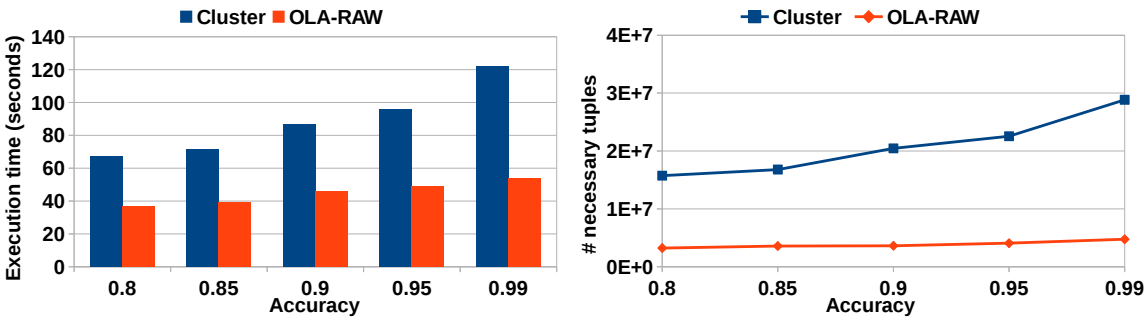
7.6.1 Micro-Benchmarks

Data. We generate a suite of synthetic CSV files to study `OLA-RAW` 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. Each tuple contains 16 columns. Each column represents a set of random number using a Zipfian distribution with different parameters, e.g. the first column contains sample numbers with parameter equal 1.0 (fully random), while the value of 16th column is 0.1, which is skew data. Overall, there are five files in the suite, and the smallest file contains 2^{20} rows – while the largest is 40 GB in size – 2^{28} rows with 16 columns each. We model dataset based on [3, 7]. 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(`C_i`) FROM FILE WHERE Accuracy` $\geq \alpha$
, where $0 \leq i \leq K$ columns projected out. By default, K is set to the number of columns in the raw file, e.g., 16 for the majority of the reported results. α defines the user-defined result accuracy, such as 80%, ground truth is defined as 100%. For the estimated result, its upper and lower bound converge to it no bigger than 1%. This simple processing interferes minimally with `OLA-RAW` thus allowing for exact measurements to be taken.

Parallelism. In the first experiment, depicted in Figure 7.7, we compare the behavior of `OLA-RAW` against External Table and Cluster Sampling, when the number of workers in the thread pool changes from 1 to 8. Figure 7.7a shows the execution time, which is measured for queries when the value of α for estimated result is greater than 99%. The data parameter we chose is 1.0, fully random data of Zipfian distribution. Notice that all these three regimes are directly supported in `OLA-RAW` with simple modifications in the controller policy. In our experiments, *READ* is always a stand-alone thread, which means I/O stage always overlaps with computations. Single worker thread guarantees chunks are processed sequentially by the thread. With two or more worker threads, all conversion stages could be overlapped. While the general trend is standard – increasing the degree of parallelism results in better performance – many findings require clarification. The execution time level-off beyond four workers. The reason for this is that processing becomes I/O-bound. Increasing the number of worker threads does not improve performance anymore, which defines the lower bound of any query execution.

As expected, *External Table* reads and extracts all the data, so it always takes much longer time than the other two methods. *OLA-RAW* converges to the lower bound of the execution time



(a) (b)

Figure 7.8: Execution time (a), necessary tuples (b) as a function of accuracy.

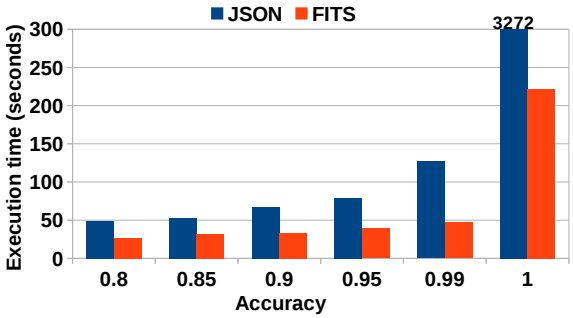


Figure 7.9: OLA-RAW in FITS and JSON files.

at the very beginning – even for the single thread. *SCANRAW-CLUSTER* is slower than *OLA-RAW* when it is CPU-bound. It gradually approaches to *OLA-RAW* and becomes the same after the processing becomes I/O-bound. Figure 7.7b/Figure 7.7c shows the number of chunks/tuples processed by *SCANRAW-CLUSTER*/*OLA-RAW*. Increasing the system performance has very limited affection on cluster sampling, the number of processed chunks and tuples are very stable for a different number of worker threads. In other words, cluster sampling cannot obtain the corresponding effective gain with the increasing of the computation power. However, the effect of parallel processing on *OLA-RAW* is significant. When query processing is CPU-bound, *OLA-RAW* accesses much more chunks than *SCANRAW-CLUSTER*, but the query execution time of *OLA-RAW* is much less than it of *SCANRAW-CLUSTER*. Computation workload decides the execution time in CPU-bound circumstances. Figure 7.7c shows that though *OLA-RAW* accesses more chunks, the number of processed tuples by *OLA-RAW* is much smaller than by *SCANRAW-CLUSTER*. As more and more worker threads involved, *OLA-RAW* automatically switches into *SCANRAW-CLUSTER*, keeping I/O and CPU stages fully overlapped. Independent of the number of workers, *OLA-RAW* adaptively chooses an optimal access plan to minimize query execution time.

Accuracy. Figure 7.8 depicts the effect of *OLA-RAW* performance when the accuracy of estimated result varies in CPU-bound processing. In this experiments, we repeatedly run the same query with increasing accuracy requirements. All queries are running directly on raw files. The purpose is to measure the effect of exactness on *OLA-RAW* execution time. To illustrate the characteristic of *OLA-RAW*, we execute queries with *SCANRAW-CLUSTER* as a comparison. Figure 7.8a shows that increasing the accuracy results in longer execution time for both methods. *OLA-RAW* is always faster than *SCANRAW-CLUSTER*. The growth rate of the execution time of *OLA-RAW* is much slower than it of *SCANRAW-CLUSTER*. Figure 7.8b describes the reasons. The number of additional tuples – to increase the accuracy of estimated result – for *OLA-RAW* is much less than the number for *SCANRAW-CLUSTER*. In other words, *OLA-RAW* executes in a more efficient way to obtain the expected result.

Query sequence. Figure 7.10 depicts *OLA-RAW* performance for a query sequence consisting of 10 standard queries, i.e., `SELECT SUM(`C_i`) FROM FILE WHERE Accuracy` $\geq \alpha$, with α evenly increase from 80% to 99%. Each accuracy configuration is executed twice, such as 80%, 80%, 85%, 85%, ... ,99%, 99%. Executing instances of the same query allows us to detect and quantify the effect of the sample loading strategy on query performance. The methods we compare is *SCANRAW-CLUSTER* and *OLA-RAW*. The size of the memory cache used is limited which can store seven full chunks for Scanraw-Cluster, due to the fine-grain cache strategy OLA-RAW could save much more small incomplete chunks. To thoroughly compared performance in different situations, we did two types of experiments; systems are configured with 1 and 12 worker threads respectively, making the process as CPU-bound and I/O bound.

Figure 7.10a shows the overall execution time after i queries in the sequence, where i goes from 1 to 10. It clearly shows that *OLA-RAW* is always faster than *SCANRAW-CLUSTER*. Besides, after executing more and more queries, the difference becomes significant. To illustrate the reason, we measure the execution time for each single query and describe it in Figure 7.10b. We can see that for the first query, both methods are slow since it is necessary to access all data from the raw file. In spite of that, OLA-RAW is better. It processes much less amount of data than *SCANRAW-CLUSTER*. Since the query is CPU-bounded, all the processed data can be flushed to database

during query execution. Therefore, both methods can achieve all the samples directly from database in the second query.

The second query is much faster than the first query in both systems. However, two strategies keep different amounts of samples in the database. The running time for the second query on OLA-RAW is much less than *SCANRAW-CLUSTER*, which reads more data from the database. As the accuracy increases, like query $3, 5, 7$ and 9, estimation only from the database is not enough. Systems have to extract more samples from the raw file, which incurs the execution time increasing. We can see that *OLA-RAW* is faster all the time because the size of loaded samples in *OLA-RAW* is always smaller than the one of *SCANRAW-CLUSTER*.

Figure 7.10c shows the behavior with 12 worker configuration. In this case, the query execution is I/O bound, and the system has not spare I/O resource to flush samples into the database during query execution, the data loading only happens at the end. We have two important observations. First, at the beginning stage, although two methods are nearly the same, *SCANRAW-CLUSTER* is even better in the second query. Second, *OLA-RAW* becomes more and more efficient after two queries. Similarly, we record and describe the execution time for each query in Figure 7.10d. In the first query, I/O bounded query procedure makes both systems have plenty of spare CPU resources to analyze all tuples for every accessed chunk. Therefore the execution time of two methods are the same – they read the same amount of data. When flushing the samples, assuming both system cache the same number of samples in memory, *OLA-RAW* keeps more chunks (although they are incomplete) of samples (chosen the maximum value of two parameters, 10% physical size or Thr_{local}). *SCANRAW-CLUSTER* stores less number of full chunks in memory. When executing the second query, both systems can get these cached samples from the database. Since *OLA-RAW* loads only a portion of samples for each chunk during the first query, it has to access more chunks. That is the reason why *OLA-RAW* is slower than *SCANRAW-CLUSTER* in this query. However, after running more queries, *OLA-RAW* pays off very quickly – only the first query in the experiment – and becomes faster than *SCANRAW-CLUSTER*. The reason is that *OLA-RAW* organizes the samples in database efficiently. Compared to *SCANRAW-CLUSTER*, *OLA-RAW* always reads and processes fewer data from the database.

Resource utilization. We compared the CPU and I/O utilization of *OLA-RAW* and *SCANRAW-CLUSTER* for processing the first query using one worker thread without any loading mechanism. The results are shown in Figure 7.10e and Figure 7.10f. In this situation, the execution is CPU-bound. We observe that in *SCANRAW-CLUSTER*, I/O stops and waits until the worker finishes the whole chunk and evacuates space to get more chunks. The processing time is longer than read time. That is why the I/O resource utilization periodically goes up and down. The interesting behavior of *OLA-RAW* is that it dynamically switches between *READ* and *EXTRACT* to utilize resources optimally. Whenever the execution is CPU-bound, *OLA-RAW* extracts the least amount of samples (no less than Thr_{local}) and immediately get a new chunk. This makes the disk utilization always keep high (random accessing data chunks makes disk utilization hard to reach 100%) during the processing. We also do experiments when the process is I/O-bound. In this situation, both methods show the similar behaviors, and *OLA-RAW* utilizes the spare CPU resources to process more tuples and maximize the CPU-resource as much as possible.

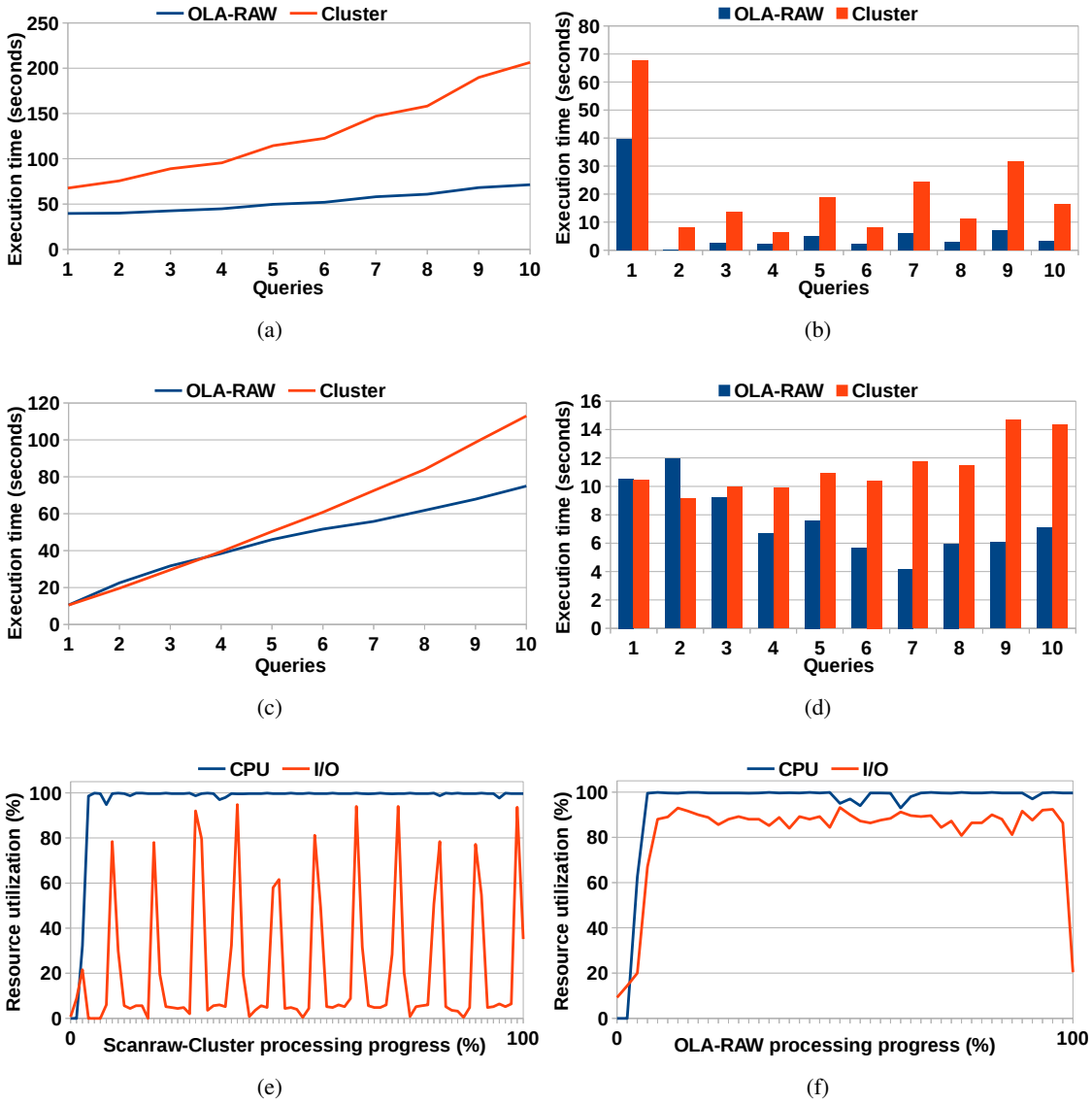


Figure 7.10: Execution time (a), necessary chunks (b), and tuples (c) as a function of the number of worker threads.

7.6.2 Alternative File Format

In order to evaluate *OLA-RAW* across multiple file formats, we also implement the support for the FITS (Flexible Image Transport System) and JSON (JavaScript Object Notation) file formats. FITS is the one of the most commonly used digital file formats in astronomy. A widely used tool to handle FITS files is the C library CFITSIO developed by NASA. It allows users to read data and apply filters to rows, columns or spatial regions.

OLA-Raw supports in-situ query directly on FITS files containing binary tables, using regular SQL statements. Accessing binary file formats is different from CSV file since extraction is not needed – each tuple and attribute can be directly retrieved from the known location, which makes query execution on FITS binary file I/O-bound. There are many JSON libraries support parsing and manipulating JSON data. We integrate JSONCPP library into OLA-RAW. JSON tuples are usually entirely parsed and converted into Java objects. The extraction procedure is a CPU-bound process, which is very different from FITS.

OLA-RAW supports query execution directly on FITS and JSON files by integrating with CFITSIO and JSONCPP, respectively. We procedurally execute the same workload on FITS and JSON data. Since the FITS file is a binary format, single worker thread for extraction – only for reading – is sufficient. Besides, due to the limitation of JSONCPP, the parsing function cannot be executed by multi-threads at the same time. Based on the above reason, the experiments are running with a single worker thread. The experiments illustrate two extreme cases: I/O bound processing on FITS file and CPU-bound execution on JSON data. Figure 7.9 depicts the execution time for the same sequence of queries discussed above over FITS and JSON data – with size 17 GB and 37GB – without any loading mechanism.

There are three observations from the results in Figure 7.9: a) *OLA-RAW* immediately terminates query execution when estimation is good enough, which is much faster than getting the exact result. b) As the accuracy increases, the execution time becomes longer as well, but increasing rate is relatively slow. c) OLA-RAW can benefit both I/O-bound and CPU-bound processing.

7.7 Conclusion

OLA-RAW – a novel system for in-situ processing over raw files that integrates online aggregation and raw file processing seamlessly while preserving their advantages. *OLA-RAW* supports single-query optimal execution with a *parallel super-scalar pipeline implementation* that overlaps data reading, conversion into the database representation, and query processing. *OLA-RAW* implements *sample maintenances* as a gradual loading mechanism to store converted data inside the database. We implement *OLA-RAW* in a state-of-the-art in-situ data processing system and evaluate its performance across a variety of different datasets and file formats. Our results show that *OLA-RAW* achieves optimal performance for a query sequence at any point in the processing.

Chapter 8

Summary and Future Work

In this paper, we mainly focus on solving the problems related in-situ data processing over raw data and propose several novel solutions.

We firstly 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.

Besides, we incorporate query workload in raw data processing, which allows us to model raw data processing with partial loading as fully-replicated binary vertical partitioning. We study the problem of workload-driven raw data processing with partial loading. We model loading as binary vertical partitioning with full replication. Based on this equivalence, we provide a linear mixed integer programming optimization formulation that we prove to be NP-hard and inapproximable. We design a two-stage heuristic that combines the concepts of query coverage and attribute usage frequency. The heuristic comes within close range of the optimal solution in a fraction of the time. We extend the optimization formulation and the heuristic to a restricted type of pipelined raw data processing. In the pipelined scenario, data access and extraction are executed concurrently. We evaluate the performance of the heuristic and the accuracy of the optimization formulation over three real data formats – CSV, FITS, and JSON – processed with a state-of-the-art pipelined operator for raw data processing. The results confirm the superior performance of the proposed heuristic over related vertical partitioning algorithms and the accuracy of the formulation in capturing the execution details of a real operator.

Furthermore, we propose *OLA-RAW* – a novel system for in-situ processing over raw files that integrates online aggregation and raw data processing seamlessly while preserving their advantages. *OLA-RAW* supports single-query optimal execution with a *parallel super-scalar pipeline implementation* that overlaps data reading, conversion into the database representation, and query processing. *OLA-RAW* implements *sample maintenances* as a gradual loading mechanism to store

converted data inside the database. We implement *OLA-RAW* in a state-of-the-art in-situ data processing system and evaluate its performance across a variety of different datasets and file formats. Our results show that *OLA-RAW* achieves optimal performance for a query sequence at any point in the processing.

In future work, we aim to continue optimizing `OLA-RAW` by finding the optimal configurations. For instance, the proper chunk size is critical for the system performance because Bi-Level sampling interrupts sequential read which degrades I/O performance. How to minimize the I/O cost is critical. Also, the estimation checking incurs computation cost. How to define the proper checking frequency is significant to the system performance. Besides, we plan to focus on extending `SCANRAW` and `OLA-RAW` with support for multi-query processing over raw files. In the usual scenario, the workload is not known apriori. Queries are admitted dynamically at runtime. The objective remains to minimize the execution time over the entire workload. The existing `SCANRAW` operator represents a sound basis for pursuing this type of work. Following the steps of database design, we envision several avenues to extend the proposed research in the future. We can move from the offline loading setting to online loading, where query processing and loading are intertwined. We can assume that the workload is known beforehand, or running a single query each time. We can drop the strict requirement of atomic attribute loading and allow for portions – horizontal partitions – of an attribute to be loaded. Finally, we can consider a multi-query processing environment in which several queries can share the raw data access and attribute extraction.

Bibliography

[1] A. Dobra et al. Turbo-Charging Estimate Convergence in DBO. *PVLDB*, 2(1), 2009.

[2] D. Abadi, P. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends in Databases*, 5(3):197–280, 2013.

[3] A. Abouzied, D. Abadi, and A. Silberschatz. Invisible Loading: Access-Driven Data Transfer from Raw Files into Database Systems. In *Proceedings of 2013 EDBT/ICDT Extended Database Technology Conference*, pages 1–10, 2013.

[4] S. Agrawal, E. Chu, and V. Narasayya. Automatic physical design tuning: workload as a sequence. In *SIGMOD 2006*.

[5] S. Agrawal, V. Narasayya, and B. Yang. Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design. In *SIGMOD 2004*.

[6] A. Ailamaki, V. Kantere, and D. Dash. Managing Scientific Data. *Commun. ACM*, 53(6):68–78, 2010.

[7] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: Efficient Query Execution on Raw Data Files. In *Proceedings of 2012 ACM SIGMOD International Conference on Management of Data*, pages 241–252, 2012.

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

[9] S. Arumugam, A. Dobra, C. Jermaine, N. Pansare, and L. Perez. The DataPath System: A Data-centric Analytic Processing Engine for Large Data Warehouses. In *Proceedings of 2010 ACM SIGMOD International Conference on Management of Data*, pages 519–530, 2010.

[10] R. Avnur and J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data*, pages 261–272, 2000.

[11] D. Barnett. BAMTools, 2013.

[12] D. Barnett, E. Garrison, A. Quinlan, M. Stromberg, and G. Marth. BamTools: a C++ API and Toolkit for Analyzing and Managing BAM Files. *Bioinformatics*, 27(12):1691–1692, 2011.

[13] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, 1997.

[14] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel Data Analysis Directly on Scientific File Formats. In *SIGMOD 2014*.

[15] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani. Parallel Data Analysis Directly on Scientific File Formats. In *Proceedings of 2014 ACM SIGMOD International Conference on Management of Data*, pages 385–396, 2014.

[16] R. Blumofe and C. Leiserson. Scheduling Multithreaded Computations by Work Stealing. 46(5):720–748, 1999.

[17] N. Bruno and S. Chaudhuri. Automatic Physical Database Tuning: A Relaxation-Based Approach. In *SIGMOD 2005*.

[18] Y. Cheng, C. Qin, and F. Rusu. GLADE: Big Data Analytics Made Easy. In *Proceedings of 2012 ACM SIGMOD International Conference on Management of Data*, pages 697–700, 2012.

[19] Y. Cheng and F. Rusu. Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID. *Distributed and Parallel Databases*, 2014.

[20] Y. Cheng and F. Rusu. Parallel In-situ Data Processing with Speculative Loading. In *Proceedings of 2014 ACM SIGMOD International Conference on Management of Data*, pages 1287–1298, 2014.

[21] Y. Cheng and F. Rusu. SCANRAW: A Database Meta-Operator for Parallel In-Situ Processing and Loading. *ACM TODS*, 40(3), 2015.

[22] W. Chu and I. T. Ieong. A Transaction-Based Approach to Vertical Partitioning for Relational Database Systems. *IEEE Transactions on Software Engineering*, 19(8):804–812, 1993.

[23] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, 4(1–3):1–294, 2012.

[24] D. W. Cornell and P. S. Yu. An Effective Approach to Vertical Partitioning for Physical Design of Relational Databases. *IEEE Transactions on Software Engineering*, 16(2):248–258, 1990.

[25] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[26] D. J. DeWitt and J. Gray. Parallel Database Systems: The Future of Database Processing or a Passing Fad? *SIGMOD Rec.*, 19(4):104–112, 1991.

[27] C. T. Fan, M. E. Muller, and I. Rezucha. Development of sampling plans by using sequential (item by item) selection techniques and digital computers. *Journal of the American Statistical Association*, 57(298):387–402, 1962.

[28] M. N. Garofalakis and P. B. Gibbon. Approximate Query Processing: Taming the TeraBytes. In *Proceedings of 2001 VLDB International Conference on Very Large Databases*, 2001.

[29] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber. Scientific Data Management in the Coming Decade. *SIGMOD Rec.*, 34(4):34–41, 2005.

[30] R. S. J. M. S. H. E. J. E. v. E. J. M. B. G. S. M. W. T. H. Grillmair, C. J.; Laher. An Overview of the Palomar Transient Factory Pipeline and Archive at the Infrared Processing and Analysis Center. In *Astronomical Data Analysis Software and Systems XIX*, page 28, 2010.

[31] M. Grund, J. Kruger, H. Plattner, A. Zeier, P. Cudre-Mauroux, and S. Madden. HYRISE: A Main Memory Hybrid Storage Engine. *PVLDB*, 4(2):105–116, 2010.

[32] P. J. Haas and C. König. A Bi-Level Bernoulli Scheme for Database Sampling. In *SIGMOD 2004*.

[33] M. Hammer and B. Niamir. A Heuristic Approach to Attribute Partitioning. In *SIGMOD 1979*.

[34] R. Hankins and J. Patel. Data Morphing: An Adaptive, Cache-Conscious Storage Technique. In *VLDB 2003*.

[35] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *Proceedings of 1997 ACM SIGMOD International Conference on Management of Data*, pages 171–182, 1997.

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

[37] J. Hoffer. An Integer Programming Formulation of Computer Data Base Design Problems. *Information Sciences*, 11(1):29–48, 1976.

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

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

[40] S. Idreos, M. L. Kersten, and S. Manegold. Self-Organizing Tuple Reconstruction in Column-Stores. In *SIGMOD 2009*.

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

[42] M. Ivanova, M. L. Kersten, and S. Manegold. Data Vaults: A Symbiosis between Database Technology and Scientific File Repositories. In *Proceedings of 2012 SSDBM International Conference on Scientific and Statistical Database Management*, pages 485–494, 2012.

[43] A. Jindal and J. Dittrich. Relax and Let the Database Do the Partitioning Online. In *BIRTE 2011*.

[44] A. Jindal, E. Palatinus, V. Pavlov, and J. Dittrich. A Comparison of Knives for Bread Slicing. *PVLDB*, 6(6):361–372, 2013.

[45] A. Jindal, J. Quiane-Ruiz, and J. Dittrich. Trojan Data Layouts: Right Shoes for a Running Elephant. In *SoCC 2011*.

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

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

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

[49] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, and R. Durbin. The Sequence Alignment/Map Format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[50] K. Lorincz, K. Redwine, and J. Tov. Grep versus FlatSQL versus MySQL: Queries using UNIX Tools vs. a DBMS, 2003.

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

[52] T. Malik, X. Wang, R. Burns, D. Dash, and A. Ailamaki. Automated Physical Design in Database Caches. In *ICDE Workshops 2008*.

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

[54] MySQL 5.7 Manual. The CSV Storage Engine, 2013.

[55] N. M. Law et al. The Palomar Transient Factory: System Overview, Performance and First Results. *CoRR*, abs/0906.5350, 2009.

[56] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical Partitioning Algorithms for Database Design. *TODS*, 9(4):680–710, 1984.

[57] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *PVLDB*, 4(11):1135–1145, 2011.

[58] S. Papadomanolakis and A. Ailamaki. AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning. In *SSDBM 2004*.

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

[60] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *Proceedings of 2008 IEEE ICDE International Conference on Data Engineering*, pages 60–69, 2008.

[61] F. Rusu, F. Xu, L. L. Perez, M. Wu, R. Jampani, C. Jermaine, and A. Dobra. The DBO Database System. In *Proceedings of 2008 ACM SIGMOD International Conference on Management of Data*, pages 1223–1226, 2008.

[62] S. Agarwal et al. Blink and It's Done: Interactive Queries on Very Large Data. *PVLDB*, 5(12), 2012.

[63] S. Wu et al. Continuous Sampling for Online Aggregation over Multiple Queries. In *SIGMOD 2010*.

[64] D. Sanchez, D. Lo, R. Yoo, J. Sugerman, and C. Kozyrakis. Dynamic Fine-Grain Scheduling of Pipeline Parallelism. In *Proceedings of 2011 PACT Conference on Parallel Architectures and Compilation Techniques*, pages 22–32, 2011.

[65] M. Z. Shieh, S. C. Tsai, and M. C. Yang. On the Inapproximability of Maximum Intersection Problems. *Information Processing Letters*, 112(19):723–727, 2012.

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

[67] A. Szalay, A. Thakar, and J. Gray. The sqlLoader Data-Loading Pipeline. *Computing in Science & Engineering*, 10(1):38–48, 2008.

[68] T. Malik et al. Adaptive Physical Design for Curated Archives. In *SSDBM 2009*.

[69] D. Tahara, T. Diamond, and D. J. Abadi. Sinew: A SQL System for Multi-Structured Data. In *SIGMOD 2014*.

[70] Thompson, S. K. *Sampling, Third Edition*. 2012.

[71] S. Vinterbo. Privacy: A Machine Learning View. *Transactions on Knowledge and Data Engineering (TKDE)*, 16(8):939–948, 2004.

[72] J. S. Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11:37–57, 1985.

[73] J. S. C. G. S. G. D. L. B. S. G. H. T. P. S. K. Wei Mi, R. Laher. The Palomar Transient Factory Data Archive. In *Databases in Networked Information Systems Lecture Notes in Computer Science Volume 7813*, pages 66–67, 2013.

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

[75] W. Zhao, Y. Cheng, and F. Rusu. Workload-Driven Vertical Partitioning for Effective Query Processing over Raw Data. *CoRR*, abs/1503.08946, 2015.