Vertical Partitioning for Query Processing over Raw Data

Weijie Zhao
UC Merced
wzhao23@ucmerced.edu

Yu Cheng
UC Merced
ycheng4@ucmerced.edu

Florin Rusu
UC Merced
frusu@ucmerced.edu

ABSTRACT

Traditional databases are not equipped with the adequate functionality to handle the volume and variety of "Big Data". Strict schema definition and data loading are prerequisites even for the most primitive query session. Raw data processing has been proposed as a schema-on-demand alternative that provides instant access to the data. When loading is an option, it is driven exclusively by the current-running query, resulting in sub-optimal performance across a query workload. In this paper, we investigate the problem of workload-driven raw data processing with partial loading. We model loading as fully-replicated binary vertical partitioning. We provide a linear mixed integer programming optimization formulation that we prove to be NP-hard. We design a two-stage heuristic that comes within close range of the optimal solution in a fraction of the time. We extend the optimization formulation and the heuristic to pipelined raw data processing, scenario in which data access and extraction are executed concurrently. We provide three case-studies over real data formats that confirm the accuracy of the model when implemented in a state-of-the-art pipelined operator for raw data processing.

1. INTRODUCTION

We are living in the age of "Big Data", generally characterized by a series of "Vs"[1]. Data are generated at an unprecedented *volume* by scientific instruments observing the macrocosm [1, 25] and the microcosm [37, 34], or by humans connected around-the-clock to mobile platforms such as Facebook and Twitter. These data come in a *variety* of formats, ranging from delimited text to semi-structured JSON and multi-dimensional binaries such as FITS.

The volume and variety of "Big Data" pose serious problems to traditional database systems. Before it is even possible to execute queries over a dataset, a relational schema has to be defined and data have to be loaded inside the database. 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 many of the attributes in the schema are never used in queries.

Motivated by the flexibility of NoSQL systems to access schemaless 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[2] includes support for JSON data type and corresponding functions. Vertica Flex Zone[3] and Sinew [33] 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 [24, 36] mechanisms. Instead of loading all the columns before querying, in adaptive partial loading [28] data are loaded only at query time, and only the attributes required by the query. This idea is further extended in invisible loading [2], where only a fragment of the queried columns are loaded, and in NoDB [15], data vaults [17], SDS/Q [6], and RAW [21], where columns are loaded only in memory, but not into the database. SCANRAW [8] 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 schemaless raw data, they have several shortcomings, as the following examples show.

Example 1: Twitter data. The Twitter API[4] 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 [33] 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 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 an heuristic, quite often sub-optimal.

[1] http://www.ibmbigdatahub.com/infographic/four-vs-big-data/

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

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

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

Example 2: Sloan Digital Sky Survey (SDSS) data. SDSS[5] 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 [30]. In order to evaluate the effectiveness of the loading, we extract a workload of 1 million SQL queries executed over the SDSS catalog[6] 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.

Problem statement. Inspired by the above examples, in this paper, 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 [21]. 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 [15, 17, 6, 21] 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 [28, 2, 8] 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.

Contributions. To the best of our knowledge, this is the first paper that incorporates query workload in raw data processing. This allows us to model raw data processing with partial loading as fully-replicated binary vertical partitioning. Our contributions are guided by this equivalence. They can be summarized as follows:

- 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.

Outline. The paper is organized as follows. Raw data processing, the formal statement of the problem, and an illustrative example are introduced in the preliminaries (Section 2). The mixed integer programming formulation and the proof that the formulation is NP-hard are given in Section 3. The proposed heuristic is presented in detail in Section 4. The extension to pipelined raw data processing is discussed in Section 5. Extensive experiments that evaluate the heuristic and verify the accuracy of the optimization formulation over three real data formats are presented in Section 6. Related work on vertical partitioning and raw data processing is briefly discussed in Section 7, while Section 8 concludes the paper.

2. PRELIMINARIES

In this section, we introduce query processing over raw data. Then, we provide a formal problem statement and an illustrative example.

2.1 Query Processing over Raw Data

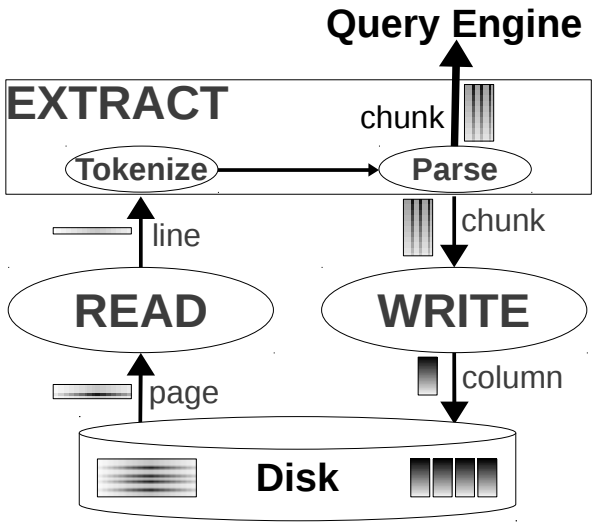


Figure 1: Query processing over raw data.

Query processing over raw data is depicted in Figure 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, a procedure to extract tuples with the given schema from the raw file, and a driver query. 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 – one per line – from 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 stages in `EXTRACT`—`TOKENIZE` and `PARSE`. `TOKENIZE` identifies the schema attributes and outputs a vector

[5] www.sdss.org/dr12/

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

containing the starting position for every attribute in the tuple—or a subset, if the driver query does not access all the attributes. In PARSE, attributes are converted from raw format to the 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 records or column arrays are grouped into a chunk—the unit of processing.

At the end of EXTRACT, data are loaded in memory and ready for query processing. Multiple paths can be taken at this point. In external tables [24, 36], data are passed to the query engine and discarded afterwards. In NoDB [15] and in-memory databases [32, 17], data are kept in memory for subsequent processing. In standard database loading [28, 2], data are first written to the database and only then query processing starts. SCANRAW [8] invokes WRITE concurrently with the query execution, only when spare I/O-bandwidth is available. The interaction between READ and WRITE is carefully scheduled in order to minimize interference.

2.2 Formal 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 [33]. 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 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 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 paper 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 [7], the conversion of the attributes stored in processing representation is executed prior to the workload execution. We let the online problem [3], in which conversion and storage are intertwined, for future work.

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 2: Variables in MIP optimization.

2.3 Illustrative Example

Table 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 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 3: Parameters in MIP optimization.

3. 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) [5] optimization with 0/1 variables. Table 2 and 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 5):

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

3.1 Objective Function

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

$$\begin{aligned}
T_{load} = &raw_0 \cdot \frac{S_{RAW}}{band_{IO}} + \\
&|R| \cdot \sum_{j=1}^{n} \left(t_{0j} \cdot T_{t_j} + p_{0j} \cdot T_{p_j} + save_j \cdot \frac{SPF_j}{band_{IO}} \right)
\end{aligned} \tag{2}$$

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

$$\begin{aligned}
T_i = &raw_i \cdot \frac{S_{RAW}}{band_{IO}} + \\
&|R| \cdot \sum_{j=1}^{n} \left(t_{ij} \cdot T_{t_j} + p_{ij} \cdot T_{p_j} + read_{ij} \cdot \frac{SPF_j}{band_{IO}} \right)
\end{aligned} \tag{3}$$

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 $band_{IO}$.

3.2 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 $\mathcal{O}(m \cdot n)$ 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 "$\leq$" operator – for a total of $\mathcal{O}(3 \cdot n)$ constraints. C_4 is a reduced form of C_3, applicable to query processing. The largest number of constraints, i.e., $\mathcal{O}(m \cdot n^2)$, 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.

3.3 Computational Complexity

There are $\mathcal{O}(m \cdot n)$ 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 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., $\frac{SPF_j \cdot |R|}{band_{IO}} \to 0$, while the raw data reading time is set to one, i.e., $\frac{S_{RAW}}{band_{IO}} \to 1$. 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 [35] 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 1 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 [35] and the solution is obtained by solving k-element cover, it implies that k-element cover

Algorithm 1 Reduce k-element cover to minimum k'-set coverage

Input: 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
Output: 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**

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 1 is correct and complete.*

The proof of this theorem is available in the extended version of the paper [38].

COROLLARY 2. *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, [29] proves that minimum k-set coverage cannot be approximated within an absolute error of $\frac{1}{2}m^{1-2\epsilon} + \mathcal{O}(m^{1-3\epsilon})$, for any $0 < \epsilon < \frac{1}{3}$, unless P = NP. Consequently, the MIP formulation cannot be approximated.

4. 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.

4.1 Vertical Partitioning

Vertical partitioning [26] 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 [16] 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 [26, 9, 4] 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.

4.2 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 2). Thus, heuristic algorithms are required.

Algorithm 2 Query coverage

Input: Workload $W = \{Q_1, \ldots, Q_m\}$; storage budget B
Output: 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 2. 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. (2)

and Eq. (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 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.

4.3 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 3 Attribute usage frequency

Input: Workload $W = \{Q_1, \ldots, Q_m\}$ of R; storage budget B; set of loaded attributes $saved = \{A_{s_1}, \ldots, A_{s_k}\}$
Output: 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: $\quad idx = \text{argmax}_{j \notin attsL} \{cost(attsL) - cost(attsL \cup A_j)\}$
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 3. 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 1.

4.4 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 4. 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 $\mathcal{O}(m + n)$.

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.

4.5 Comparison with Heuristics for Vertical Partitioning

A comprehensive comparison of vertical partitioning methods is given in [19]. With few exceptions [26, 27], 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 [27]. At a high level, vertical partitioning algorithms can be classified along several axes [19]. 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 [26, 9, 4] 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 [12, 13, 27, 11, 20] starts with a series of schemas, e.g., one for each attribute or one for each subset of attributes accessed

Algorithm 4 Heuristic algorithm

Input: Workload $W = \{Q_1, \ldots, Q_m\}$; storage budget B
Output: 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
```

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 [12, 26, 13, 27, 4, 11, 20] 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 [9] 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 6) and a discussion on their differences in the related work (Section 7).

5. 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 [8], 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 4 that makes the algorithm feasible for pipelined processing.

5.1 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:

$$T_i^{pipe} = |R| \cdot \sum_{j=1}^{n} read_{ij} \cdot \frac{SPF_j}{band_{IO}} + \max\left\{ raw_i \cdot \frac{S_{RAW}}{band_{IO}}, |R| \cdot \sum_{j=1}^{n} \left(t_{ij} \cdot T_{t_j} + p_{ij} \cdot T_{p_j} \right) \right\} \quad (4)$$

This is the only modification to the MIP formulation for sequential processing given in Section 3. Since the max function is non-linear, solving the modified formulation becomes impossible with standard MIP solvers, e.g., CPLEX[7], 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. (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 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:

$$\begin{aligned} C_7 &: cpu_i + io_i = 1;\ i = \overline{1,m} \\ C_{8-10} &: cpu.x + io.x = x;\ x \in \{raw_i, t_{ij}, p_{ij}\} \\ C_{11-13} &: cpu.x \leq cpu_i;\ i = \overline{1,m} \\ C_{14-16} &: io.x \leq io_i;\ i = \overline{1,m} \end{aligned} \quad (5)$$

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

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

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 5th attribute is considerably smaller than the time to tokenize the 205th 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 $PT = \left\lceil \frac{\frac{S_{RAW}}{band_{IO}} - |R| \cdot \sum_{j=1}^{n} T_{t_j}}{\frac{|R| \cdot \sum_{j=1}^{n} T_{p_j}}{n}} \right\rceil$ 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:

$$\begin{aligned} C_{17} &: \sum_{j=1}^{n} p_{ij} - PT < cpu_i \cdot n;\ i = \overline{1,m} \\ C_{18} &: PT - \sum_{j=1}^{n} p_{ij} \leq io_i \cdot n;\ i = \overline{1,m} \end{aligned} \tag{6}$$

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. (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:

$$\begin{aligned} T_i = & io.raw_i \cdot \frac{S_{RAW}}{band_{IO}} + |R| \cdot \sum_{j=1}^{n} read_{ij} \cdot \frac{SPF_j}{band_{IO}} + \\ & |R| \cdot \sum_{j=1}^{n} \left(cpu.t_{ij} \cdot T_{t_j} + cpu.p_{ij} \cdot T_{p_j} \right) \end{aligned} \tag{7}$$

5.2 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 4, 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. 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 [8] 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 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

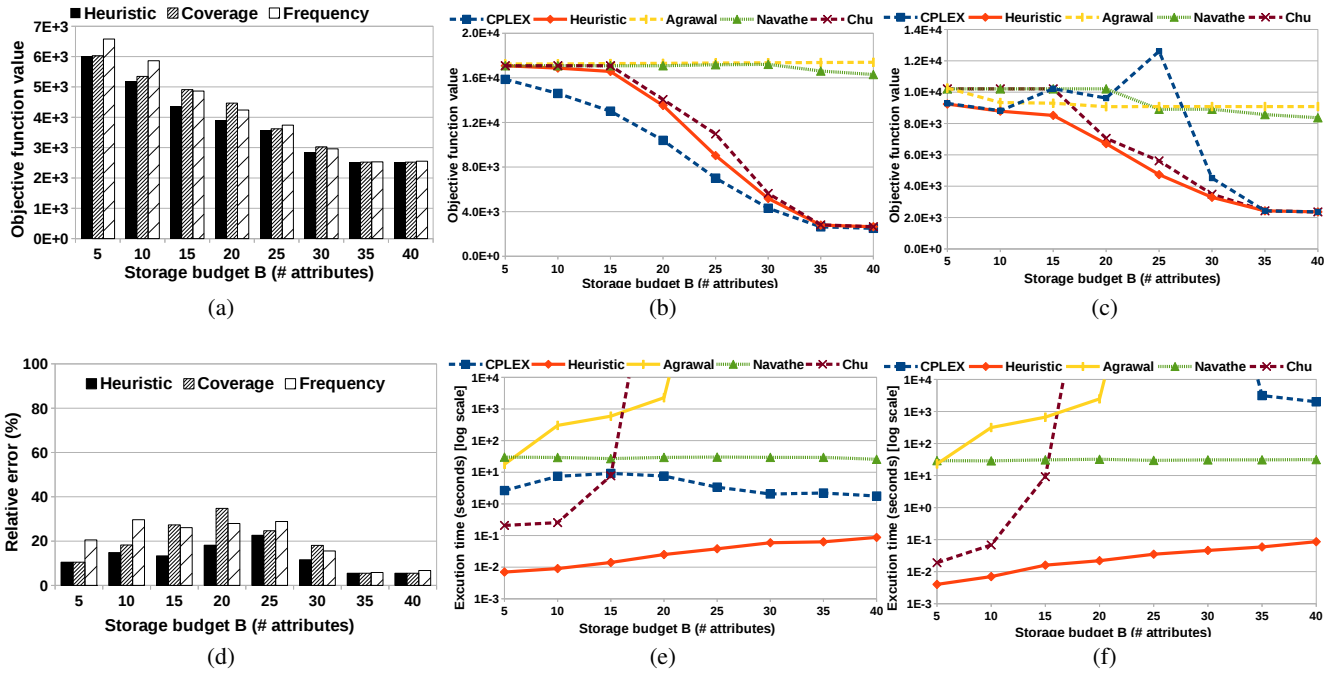


Figure 2: Experimental results for the heuristic algorithm. Comparison between the stages: (a) objective function value; (d) relative error with respect to the optimal solution. Comparison with CPLEX and vertical partitioning algorithms in objective function value (b,c) and execution time (e,f) for serial (b,e) and pipelined (c,f) raw data processing.

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.

6.1 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 [4], Navathe [26], and Chu [9]. The same objective function (Eq. (1)) is used across all of them. A detailed description of our implementation can be found in the extended version of the paper [38].

The *Agrawal algorithm* [4] 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* [26] 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* [9] 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 subparts. 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 $\mathcal{O}(2^m)$. The solution proposed in [9] 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 2a and 2d 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 2a) and the relative error with respect to the optimal value (Figure 2d). 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 2b 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 [9] is the most accurate. The other two algorithms [26, 4] 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 2e depicts the execution time for the same scenario as in Figure 2b. 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 [9] and Agrawal [4]. 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 2c. 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 2f). 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.

6.2 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 1 is general and the MIP models corresponding to this architecture fit reality. We use the implementation of the architecture in the SCANRAW operator [8] 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 model will be accurate. Figure 3a confirms this on the SDSS workload of 32 queries. In this case, there is a perfect match between the model and the SCANRAW execution.

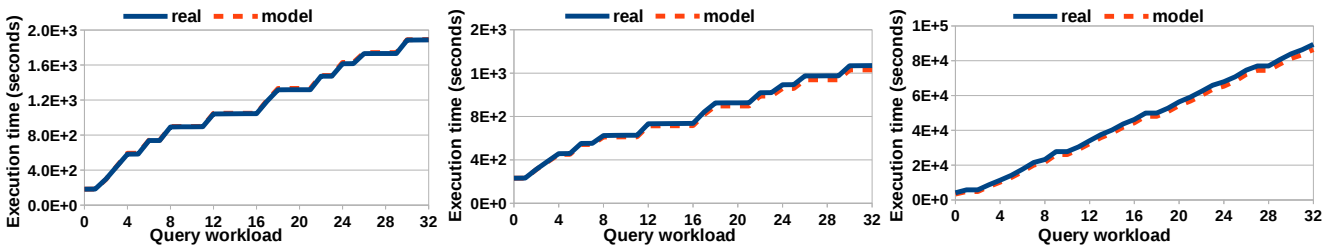
6.3 Case Study: FITS Format

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[8] – 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 3b.

6.4 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

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



(a) (b) (c)

Figure 3: Model validation: (a) serial CSV, (b) serial FITS, (c) pipeline JSON.

requested attributes. At least this is how the JSONCPP[9] 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 3c confirm the accuracy of the model over a workload of 32 queries executed in SCANRAW.

6.5 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 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.

7. 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 [36, 24]. 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 [15], Data Vaults [17], SDS/Q [6], RAW [21], and Impala [22]. Even when loading is an option, for example in adaptive partial loading [28], invisible loading [2], and SCANRAW [8], 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 [19]. The serial MIP formulation for raw data processing is inspired from the formulations for vertical partitioning given in [14, 10]. 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 4.5, 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 [9] is the closest to the query coverage stage. While query coverage is a greedy algorithm, [9] employs exhaustive search to find the solution. As the experimental results show, this is time-consuming. Other top-down heuristics [26, 4] 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 [26], 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 [4] is to find a set of vertical partitions that are subsequently evaluated for index creation. Since we select a single partitioning

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

scheme, the process is less time-consuming. Finally, the difference between the proposed heuristic and bottom-up algorithms [12, 13, 27, 11, 20] 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 [3], the entire workload sequence is known in advance. However, the vertical partitioning evolves with each query. Another series of algorithms [23, 31, 18] 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.

8. CONCLUSIONS AND FUTURE WORK

In this paper, 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.

Following the steps of database physical 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 it is given one query at a 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 raw data access and attribute extraction can be shared across several queries.

Acknowledgments. This work is supported by a U.S. Department of Energy Early Career Award (DOE Career).

9. REFERENCES

[1] A. Szalay et al. Designing and Mining Multi-Terabyte Astronomy Archives: The Sloan Digital Sky Survey. In *SIGMOD 2000*.

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

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

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

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

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

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

[8] Y. Cheng and F. Rusu. Parallel In-Situ Data Processing with Speculative Loading. In *SIGMOD 2014*.

[9] 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.

[10] 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.

[11] 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.

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

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

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

[15] I. Alagiannis et al. NoDB: Efficient Query Execution on Raw Data Files. In *SIGMOD 2012*.

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

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

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

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

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

[21] M. Karpathiotakis, M. Branco, I. Alagiannis, and A. Ailamaki. Adaptive Query Processing on RAW Data. *PVLDB*, 7, 2014.

[22] M. Kornacker et al. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR 2015*.

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

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

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

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

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

[28] S. Idreos et al. Here Are My Data Files. Here Are My Queries. Where Are My Results? In *CIDR 2011*.

[29] 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.

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

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

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

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

[34] The 1000 Genomes Project Consortium. A Map of Human Genome Variation from Population-Scale Sequencing. *Nature*, 467(7319):1061–1073, 2010.

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

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

[37] A. Wright and R. Webb. The Large Hadron Collider. *Nature Insight*, 448(7151):269–312, 2007.

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