Skipping-oriented Partitioning for Columnar Layouts

Liwen Sun, Michael J. Franklin, Jiannan Wang[†] and Eugene Wu[‡]

University of California Berkeley, Simon Fraser University[†], Columbia University[‡]
{liwen, franklin}@berkeley.edu, jnwang@sfu.ca, ewu@cs.columbia.edu

ABSTRACT

As data volumes continue to grow, modern database systems increasingly rely on data skipping mechanisms to improve performance by avoiding access to irrelevant data. Recent work [39] proposed a fine-grained partitioning scheme that was shown to improve the opportunities for data skipping in row-oriented systems. Modern analytics and big data systems increasingly adopt columnar storage schemes, and in such systems, a row-based approach misses important opportunities for further improving data skipping. The flexibility of column-oriented organizations, however, comes with the additional cost of tuple reconstruction. In this paper, we develop Generalized Skipping-Oriented Partitioning (GSOP), a novel hybrid data skipping framework that takes into account these row-based and column-based tradeoffs. In contrast to previous column-oriented physical design work, GSOP considers the tradeoffs between horizontal data skipping and vertical partitioning jointly. Our experiments using two public benchmarks and a real-world workload show that GSOP can significantly reduce the amount of data scanned and improve end-to-end query response times over the state-of-the- art techniques.

1. INTRODUCTION

Data skipping has become an essential mechanism for improving query performance in modern analytics databases (e.g., [1,8,16, 26,40]) and the Hadoop ecosystem (e.g., [2,43]). In these systems, data are organized into *blocks*, each of which typically contains tens of thousands of tuples. At data loading time, these systems compute statistics for each block (e.g., such as min and max values of each column) and store the statistics as metadata. Incoming queries can evaluate their filter predicates against such metadata and decide which blocks can be safely skipped (i.e., do not need to be read or accessed). For example, suppose a query contains a predicate `age=20`; if the metadata of a block says its min value on column `age` is `25`, then this block can be skipped by this query. Reading less data not only saves I/O, but also reduces CPU work, such as decompression and deserialization. Therefore, data skipping improves query performance even when data are memory- or SSD-resident.

Proceedings of the VLDB Endowment, Vol. 10, No. 4
Copyright 2016 VLDB Endowment 2150-8097/16/12.

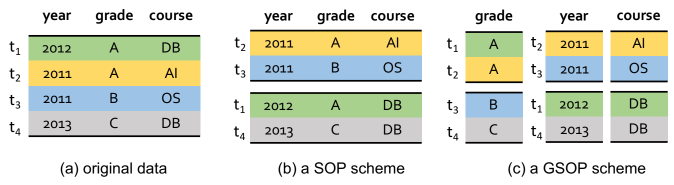

Figure 1: Example of partitioning schemes.

The opportunity for data skipping highly depends on how the tuples are organized into blocks. While traditional horizontal partitioning techniques, such as range partitioning, can be used for this purpose, in recent work [39], we proposed a skipping-oriented partitioning (SOP) framework, which can significantly improve the effectiveness of data skipping over traditional techniques. The SOP framework first analyzes the workload and extracts representative filter predicates as *features*. Based on these features, SOP characterizes each tuple by a *feature vector* and partitions the data tuples by clustering the feature vectors.

While SOP has been shown to outperform previous techniques, its effectiveness depends on workload and data characteristics. Modern analytics applications can involve wide tables and complex workloads with diverse filter predicates and column-access patterns. For this kind of workloads, SOP suffers from a high degree of *feature conflict*. Consider the table in Figure 1(a). Suppose SOP extracts two features from the workload: F_1:`grade='A'` and F_2:`year>2011∧course='DB'`. In this case, the best partitioning scheme for feature F_1 is $t_1t_2|t_3t_4$, since t_1 and t_2 satisfy F_1 while t_3 and t_4 do not. For the same reason, the best partitioning scheme for feature F_2 is $t_1t_4|t_2t_3$. Therefore, the conflict between F_1 and F_2 lies in that their best partitioning schemes are different. Since SOP generates a single horizontal partitioning scheme that incorporates *all* features (e.g., Figure 1(b)), when there are many highly conflicting features, it may be rendered ineffective.

The key reason why SOP is sensitive to feature conflict is that it produces only monolithic horizontal partitioning schemes. That is, SOP views every tuple as an *atomic* unit. While this perspective is natural for row-major data layouts, it becomes an unnecessary constraint for columnar layouts. Analytics systems increasingly adopt columnar layouts [20] where each column can be stored separately. Inspired by recent work in column-oriented physical design, such as database cracking [23,38], we propose to remove the "atomic-tuple" constraint and allow different columns to have different horizontal partitioning schemes. By doing so, we can mitigate feature conflict and boost the performance of data skipping. Consider the example in Figure 1(c), where we partition column `grade` based on F_1 and independently partition the columns `year` and `course` based on F_2. This hybrid partitioning scheme successfully resolves the conflict between F_1 and F_2, as the relevant columns for each can be partitioned differently. Unfortunately, this columnar approach incurs overhead for *tuple-reconstruction* [24], i.e., the process of assembling column values into tuples during query processing. Since column values are no longer aligned, to query such data, we may need to maintain tuple ids and *join* the column values using these tuple ids. Thus, although combining horizontal and vertical partitioning has large potential benefits, it is unclear how to balance the benefits that a particular vertical partitioning scheme has on skipping against its tuple-reconstruction overheads.

To this end, we propose a generalized SOP (GSOP) framework, with a goal of optimizing the overall query performance by automatically balancing skipping effectiveness and tuple-reconstruction overhead. GSOP generalizes SOP by removing the atomic-tuple constraint and allowing both horizontal and vertical partitionings. For a given data and workload setting, GSOP aims to pick a hybrid partitioning scheme that maximizes the overall query performance.

Given the goal of GSOP, we can think of several naïve approaches. The first approach is to apply a state-of-the-art vertical partitioning technique (e.g., [10,14,27,46]) to divide the columns into groups and to then use SOP to horizontally partition each column group. Such an approach, however, is oblivious to the potential impact of vertical partitioning on skipping horizontal blocks. Another naïve approach is to first horizontally partition the data into blocks using SOP and then vertically partition each block using existing techniques. In this approach, although the columns are divided into groups, they still have the same horizontal partitioning scheme, since this approach runs SOP (using all features) before applying the vertical partitioning. Thus, this apporach does not really help mitigate feature conflict. Both naive approaches fail to incorporate the interrelation between horizontal and vertical partitionings and how they jointly affect data skipping.

In the design of GSOP, we propose a *skipping-aware column grouping* technique. We develop an objective function to quantify the trade-off of skipping effectiveness vs. tuple-reconstruction overhead. One major technical challenge involved in using such an objective function is to estimate the effectiveness of data skipping, i.e., how much data can be skipped by queries. As described in Section 4.3, directly assessing skipping effectiveness is prohibitively expensive, so we propose an efficient yet accurate estimation approach. Finally, we devise an algorithm to search for the column grouping scheme that optimizes the objective function.

To mitigate feature conflict, GSOP separates the set of *global features* extracted in SOP into sets of *local features*, one set for each column group. We refer to this problem as *local feature selection*. To solve this problem, we develop principled ways of: (1) identifying which features should be assigned to each column group, (2) weighting features w.r.t. a column group, as a feature may have different weights for different column groups, and (3) determining the appropriate number of features to use for each column group, as the number of features needed for different groups can vary greatly.

We prototyped GSOP using Apache Spark [29]. Note that GSOP is an offline process that is executed once at data loading time. In a data warehouse environment, for example, when a new date partition arrives, GSOP reorganizes this raw partition into an optimized layout and appends it to the table. This process does not affect previously existing data. In the prototype, we store the GSOP-partitioned data using Apache Parquet [2], a columnar storage format for the Hadoop ecosystem. We then queried the data with Spark SQL and measured the performance. Experiments on two public benchmarks and a real-world workload show that GSOP can significantly improve the end-to-end query response times over SOP. Specifically, in TPC-H, GSOP reduces the data read by $6.5\times$ and improve the query response time by $3.3\times$ over SOP.

To summarize, we make the following contributions:

- We propose a GSOP framework for columnar layouts, which generalizes SOP by removing the atomic-tuple constraint.
- We develop an objective function to quantify the skipping vs. reconstruction trade-off of GSOP.
- We devise a skipping-aware column grouping algorithm and propose techniques to select local features.
- We prototype GSOP using Parquet and Spark and perform an experimental evaluation using two public benchmarks and a real-world workload. Our results show that GSOP can significantly outperform SOP in the presence of feature conflict.

2. BACKGROUND

2.1 SOP vs. Range Partitioning

How data is partitioned into blocks can significantly affect the chances of data skipping. A common approach is to perform a range partitioning on the frequently filtered columns. As pointed out in [39], however, range partitioning is not an ideal solution for generating blocks at a fine-granularity. To use range partitioning for data skipping, we need a principled way of selecting partitioning columns and capturing inter-column data correlation and filter correlation. SOP extracts features (i.e., representative filters which may span multiple columns) from a query log and constructs a fine-grained partitioning map by solving a clustering problem.

SOP techniques can co-exist with traditional partitioning techniques such as range partitioning, as they operate at different granularities. In a data warehouse environment, for example, data are often horizontally partitioned by date ranges. Traditional horizontal partitioning facilitates common operations such as batch insertion and deletion and enables partition pruning, but is relatively coarse-grained. While partition pruning helps queries skip some partitions based on their date predicates, SOP further segments each horizontal partition (say, of 10 million tuples) into fine-grained blocks (say, of 10 thousand tuples) and helps queries skip blocks inside each unpruned partition. Note that SOP only works within each horizontal partition and does not move data across partition boundaries. Thus, adding a new date partition or changing an existing partition of the table does not affect the SOP schemes of other partitions.

2.2 The SOP Framework

The SOP framework is based on two interesting properties observed from real-world analytical workloads [39]:

(1) Filter commonality, which says that a small set of filters are commonly used by many queries. In other words, the filter usage is highly skewed. In a real-world workload analyzed in [39], 10% of the filters are used by 90% of the queries. This implies that if we design a layout based on this small number of filters, we can benefit most of the queries in the workload.

(2) Filter stability, which says that only a tiny fraction of query filters are newly introduced over time, i.e., most of the filters have occurred in the past. This property implies that designing a data layout based on past query filters can also benefit future queries.

Given these two workload observations, we next go through the steps of SOP using Figure 2 as an example.

1). Workload Analysis. This step extracts as *features* a set of representative filter predicates in the workload by using frequent itemset mining [13]. A feature can be a single filter predicate or multiple conjunctive predicates, which possibly span multiple columns. A predicate can be an equality or range condition, a string matching operation or a general boolean user-defined function (UDF). Note that we do not consider as features the filters on date and time

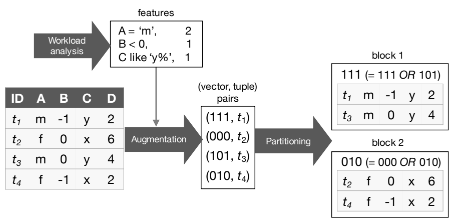
Figure 2: Using SOP to partition data into 2 blocks.

columns, as their filter values tend to change over time. In Figure 2, we extract three features, each of which is associated with a weight indicating the importance of the feature. Note that SOP takes into account *subsumption relations* when extracting features. For example, whether a filter `B<0` is chosen as a feature does not depend only on how many times `B<0` itself occurs in the workload but also depends on how many queries it subsumes. A feature subsumes a query when the feature is a more relaxed condition than the query predicates. Thus, the presence of a filter like `B<-1` in the workload can increase the chance of `B<0` being selected, as `B<0` subsumes `B<-1`. As explained later in this section, we consider subsumption relations for skipping data during query processing.

2). Augmentation. Given the features, at load time, SOP then scans the data. For each tuple, it batch-evaluates these features and stores the evaluation results as an augmented *feature vector*. Given m features, a feature vector is a m-dimensional bit vector, the i-th bit of which indicates whether this tuple satisfies the i-th feature or not. For example, in Figure 2, t_4 is augmented with a feature vector (010), which indicates that t_4 satisfies the second feature `B<0` but does not satisfy the other two.

3). Partitioning. In this step, SOP first *groups* the (vector, tuple)-pairs into (vector, count)-pairs. This is an important optimization for accelerating the partitioning algorithm. Then, a clustering algorithm is performed on the (vector, count)-pairs, which generates a *partition map*. This map guides individual tuples to their destination blocks. After the tuples are organized into blocks, SOP annotates each block with a *union vector*, which is a bitwise `OR` of all the feature vectors in the block. In Figure 2, the feature vectors of t_1 and t_3 are 111 and 101, respectively, so the union vector for their block is $111 = 111$ `OR` 101. As discussed below, union vectors carry important information for skipping. Once we obtain the union vectors, the individual feature vectors can be discarded.

Given the data partitioned by SOP, when a query arrives, we first check which features subsume this query. We then decide which blocks can be skipped based on their union vectors. Recall that a union vector is a bitwise `OR` of all the feature vectors of this block. Thus, when the i-th bit of the union vector is 0, we can know that no tuple in this block satisfies the i-th feature. For example, consider the following query:

```
SELECT A, D FROM T WHERE A='m' and D= 2
```

In Figure 2, only feature `A='m'` subsumes this query, as `A='m'` is a more relaxed condition than the query predicate `A='m' and D=2`. Since `A='m'` is the first feature, we look at the first bit of these two union vectors. As the union vector of block 2 (i.e., 010) has a 0 on the first bit, we can skip the block 2.

Note that SOP is typically executed once at data loading time. For example, when a new date partition arrives, SOP reorganizes its layout before it is appended to the table.

3. GENERALIZING SOP

In this section, we discuss how to generalize SOP by exploiting the properties of columnar data layouts.

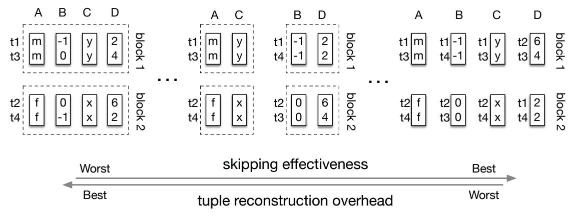

Figure 3: The spectrum of partitioning layouts.

3.1 A Simple Extension for Columnar Layouts

Modern analytics databases [8,9,40] and the Hadoop ecosystem [2, 43] adopt columnar layouts. In a columnar layout, each column can be stored separately. To process a query, these systems read all the requested columns and assemble them back into tuples through a process commonly called *tuple reconstruction*.

A simple extension to SOP for columnar layouts is to partition each column individually. By allowing each column to have its own partitioning scheme, we can mitigate feature conflict and thus enjoy better skipping. While this simple extension reduces the reading cost through better data skipping, it introduces overhead for tuple reconstruction. Since the columns are in different orders now, each column value has to be associated with an original *tuple-id*. A query then needs to read the tuple ids in addition to the actual data, and *join* the columns back using these tuple ids, as opposed to simply *stitching* them together when they are aligned. For example, one way to join these columns is to first sort them by their tuple ids and then stitch them together. Therefore, for this extension, it is unclear whether the benefit of skipping can outweigh the overhead introduced for tuple reconstruction.

3.2 Spectrum of Partitioning Layouts

The existing SOP framework in [39] and the simple extension in Section 3.1 represent two extremes of partitioning layouts in a column-oriented storage. Let us now consider the entire spectrum as depicted in Figure 3. The left end of the spectrum represents the data partitioned by the existing SOP framework. When all columns follow the same partitioning scheme, the skipping effectiveness is limited by feature conflict, but there is no overhead for tuple reconstruction. The other end of the spectrum is the data partitioned by the extension discussed in Section 3.1. When each column can have its own partitioning scheme, the skipping effectiveness is the best due to the least feature conflict, but the overhead for tuple reconstruction is the greatest. Clearly, which one of these two layouts is better depends on the workload and data characteristics. However, what is interesting is the middle ground of the spectrum, where columns can form groups. Each column group can have its own partitioning scheme, which all of its column members must follow. The potential of such a middle ground is to provide a good balance between skipping effectiveness and tuple reconstruction overhead such that the overall query performance can be optimized. We illustrate this point using an example.

EXAMPLE 1. *Figure 3 shows three different layouts of the table in Figure 2. These three layouts are all columnar but represent different points on the partitioning layout spectrum. Suppose we run the following SQL query on this table:*

```
SELECT B, D FROM T WHERE B<0 and D=2
```

Let us do a back-of-the-envelope calculation of the cost of query processing for each of these three layouts. To simplify Figure 3, we omit showing the block metadata, such as union vectors.

Left end. The table is partitioned into two blocks, each of which has all four columns. For this query we cannot skip any block because both blocks have some tuple that satisfies the query predi-

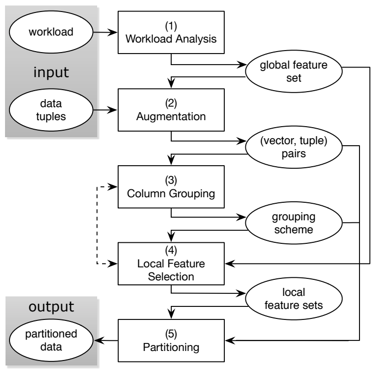

Figure 4: The GSOP framework.

cates (i.e., t_1 in block 1 and t_4 in block 2). Thus, we have to read column B and D in their entirety, which are 8 data cells in total.

Right end. Each column is partitioned into two blocks. The query only looks at column B and D. We can skip block 2 of column B, because no value in it satisfies $B < 0$. Similarly, for column D, we can skip its block 1, as none of its values satisfies $D = 2$. Thus, we need to read 4 data cells in total. For tuple reconstruction, however, we have to additionally load 1 tuple id for each data value read. In total, the cost of this query includes reading 4 data values, reading 4 tuple ids, and joining column B and D.

Middle ground. The columns are first divided into two groups: (A, C) and (B, D). Each column group is then partitioned into two blocks. The query only needs to look at group (B, D), in which we can skip block 2 as it has no value that satisfies the query predicates. Thus we only have to read block 1 of column group (B, D), which has 4 data cells. Since columns B and D are in the same group and are thus aligned, there is no overhead for tuple reconstruction. Hence, the total cost is to read only 4 data cells.

The above example shows that, for this particular query, the middle ground is a clear winner, as it enjoys both the skipping effectiveness of the right end and the zero assembly overhead of the left end. Obviously, for a given workload and data, the optimal layout could be any point on this spectrum. The SOP framework is limited to the left end. We next present a generalized SOP (GSOP) framework that incorporates the full spectrum.

3.3 The GSOP Framework

GSOP takes a workload and a set of input data and outputs data in an optimized layout. In practice, when a new date partition is being inserted into the table, we apply GSOP to reorganize its layout in order to benefit future queries. Like SOP, GSOP works within each horizontal partition, so loading a new partition does not affect the existing partitions of the table. Thus, GSOP is an offline process at load time and usually does not to be re-run unless the workload patterns change dramatically. As illustrated in Figure 4, GSOP consists of the following steps:

1). Workload Analysis. This is the same as Step 1 in Section 2.2. We analyze a given workload (e.g., a query log) and extract representative filters as features. Here we call these features *global features*. In addition, for each global feature, we maintain a list of queries from the workload that this feature can subsume.

2). Augmentation. This is the same as Step 2 in Section 2.2. We scan the input data once and batch evaluate the global features on each tuple. Each tuple is augmented with a *global feature vector*.

3). Column Grouping. Before horizontally partitioning the data, we first divide the columns into column groups based on an objective function that incorporates the trade-off between skipping effectiveness and tuple-reconstruction overhead (we discuss the details of defining such an objective function, developing an efficient way to evaluate it, and devising algorithms to search for column grouping schemes in Section 4). This step outputs a column grouping scheme.

4). Local Feature Selection. For each column group, we select a subset of global features as *local features*. These local features will be used to guide the partitioning of each column group. This is a crucial step for enhancing skipping effectiveness. The local features are more specific to each column group and hence may involve much less conflict than the global features. Note that the column grouping process (Step 3) needs to call this step as a subroutine repeatedly. Thus, we need to select local features very efficiently. We will cover the details of this step in Section 5.

5). Partitioning. We next partition each column group individually. To partition each column group, we need *local feature vectors* that correspond to the local features. Since a set of local features is a subset of global features (computed in Step 2), for each column group, we can *project* the global feature vectors to keep only the bits that correspond to the local features. For each column group, we then invoke Step 3 in Section 2.2 to partition the data based on their local feature vectors.

Comparing with the SOP framework, we can see that GSOP adds little complexity. The main technical challenges are in the two new steps: column grouping and local feature selection. We explain how column grouping (Section 4) and local feature selection (Section 5) work in detail.

Handling Normalized Schemas. The GSOP framework is motivated by modern analytics scenarios, where data is often stored as single denormalized tables [36,44]. Some traditional relational applications, however, manage data in normalized schemas, where queries involve joining normalized tables. In practice, we can apply GSOP on normalized schemas through a simple *partitial denormalization* step. First, through workload analysis, we identify the columns that have occurred in a join query in the workload. At data loading time, we pre-join these columns as a *partial denormalization* of the tables and leave in the original normalized tables the columns that never appeared in a join query in the workload. We then apply GSOP on this partially denormalized table. Most of the incoming queries can then be redirected, through proper query rewriting, to this partially denormalized table and enjoy the skipping benefits provided by GSOP. Compared to a full denormalization, this paritial denormalization based on workload information incurs less joining cost and leads to smaller resulting data sizes. In Section 7, we show that applying GSOP on normalized TPC-H tables via partitial denormalization can significantly reduce the amount of data scanned and improve the query response time.

4. COLUMN GROUPING

4.1 Motivation

Column grouping is an important database technique and has been extensively studied (e.g., [14,25,30,35,37,46]). While many column grouping approaches exist, the general principle is to put into the same group the columns that are frequently queried together in the workload [25] and adopt a row-major layout within each column group. In contrast, GSOP still uses columnar layout inside column groups, and more importantly, the grouping decision in GSOP needs to incorporate the opportunities of skipping horizontal blocks within each column group. We illustrate these considerations using Example 2.

EXAMPLE 2. *Consider the following workload for the table in Figure 2:*

Q_1: `SELECT A, C FROM T WHERE A = 'm'`

Q_2: `SELECT B, D FROM T WHERE B < 0`

Q_3: `SELECT B, C FROM T WHERE C like 'y%'`

By considering only the column co-access patterns, the column pairs AC, BD and BC would have equal weights of being grouped together, as each of them occurs once in the workload. In GSOP, however, we should consider how these groups may potentially affect data skipping. After evaluating these filters on the data, we can see that filters `A='m'` *and* `C like 'y%'` *are perfectly correlated, as t_1 and t_3 satisfy both* `A='m'` *and* `C like 'y%'` *while t_2 and t_4 do not satisfy either. Thus, GSOP may prefer the column group AC, as this type of filter correlation plays an important role in skipping horizontal blocks. Existing column grouping techniques do not take into account such information.*

4.2 Objective Function

Let C be the set of columns in the table. We denote by $\mathbb{G} = \{G_1, G_2, \ldots, G_m\}$ a column grouping scheme of the table. Thus, $\mathbb{G}$ is a partitioning over the column set C, i.e., $\bigcup_{G_i \in \mathbb{G}} G_i = C$ and $G_i \cap G_j = \emptyset$ for any $i \neq j$. Given a query q, let $C^q \subseteq C$ be the set of columns that query q needs to access. We denote by $\mathbb{G}^q \subseteq \mathbb{G}$ the column groups that query q needs to access, i.e., $\mathbb{G}^q = \{G_i \in \mathbb{G} \mid G_i \cap C^q \neq \emptyset\}$.

Skipping Effectiveness. We call each column value of a tuple a *data cell*. We quantify the skipping effectiveness of a column grouping scheme as the number of data cells we have to scan, i.e., cannot be skipped. For ease of presentation, we assume scanning a data cell incurs a uniform cost 1, but our model can be easily extended to a more general case. For every column group $G_i \in \mathbb{G}^q$, query q needs to scan $|G_i \cap C^q|$ columns. Let r_i^q denote the number of rows that query q needs to scan in group G_i. Thus, the scanning cost that query q spends on G_i is $|G_i \cap C^q| \cdot r_i^q$. The overall scanning cost for query q is:

$$\sum_{G_i \in \mathbb{G}^q} |G_i \cap C^q| \cdot r_i^q. \tag{1}$$

Equation 1 computes the skipping effectiveness of the column grouping scheme $\mathbb{G}$ w.r.t. query q. Clearly, the value r_i^q plays an essential role in Equation 1. We will discuss how to obtain the value of r_i^q in Section 4.3.

Tuple Reconstruction Overhead. Since different column groups can be partitioned in different ways, we need a way to reconstruct the values from multiple column groups back into tuples. In every column group, we store a tuple-id for each row to indicate which tuple that row originally belongs to. When a query reads data from multiple column groups, i.e., $|\mathbb{G}^q| > 1$, it also has to read the tuple-ids from each column group. The query does not have to read the tuple-ids when it only uses data from a single column group. After reading all the data columns, we need to join these column values back together as tuples. While there are many ways of implementing the join, we simply assume a sort-merge join in our cost estimation. The model can be easily modified for other join implementations. In a sort-merge join, we have to sort each column group by their tuple ids and then stitch all column groups back together. As we can see, compared with the case where all columns have a monolithic partitioning scheme, the tuple-reconstruction overhead here mainly comes from two sources: 1) reading tuple-ids and 2) sorting column values by tuple-ids. When a query only reads data from a single column group, this overhead is zero. Let us now consider the case where a query needs to access multiple column groups. Since we do not need to read the tuple ids for the values that can be skipped, the cost for query q to read tuple ids in G_i is simply r_i^q, no matter how many columns q reads in G_i. Let $\mathsf{sort}(x)$ denote the cost of sorting a list of x values. For column group G_i, we need to sort r_i^q values. Therefore, the overhead to tuple-reconstruction for query q on column grouping scheme $\mathbb{G}$ is:

$$\mathsf{overhead}(q, \mathbb{G}) = \begin{cases} \sum_{G_i \in \mathbb{G}^q} (r_i^q + \mathsf{sort}(r_i^q)) & \text{if } |\mathbb{G}^q| > 1 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Objective Function. Based on Equations 1 and Equation 2, the cost of processing query q w.r.t a column grouping scheme $\mathbb{G}$ is:

$$\mathsf{COST}(q, \mathbb{G}) = \sum_{G_i \in \mathbb{G}^q} |G_i \cap C^q| \cdot r_i^q + \mathsf{overhead}(q, \mathbb{G}) \tag{3}$$

The cost of processing the entire workload W is the sum of all the queries in the workload. Thus, we have:

$$\mathsf{COST}(W, \mathbb{G}) = \sum_{q \in W} \mathsf{COST}(q, \mathbb{G}) \tag{4}$$

We are aware that modern column-store systems employ advanced compression techniques and compression-aware execution strategies [19]. For simplicity and generality, however, our cost model does not factor in these advanced techniques. As shown in Section 7, our simple cost model works well when the data is stored in Parquet, which adopts standard compression techniques such as RLE encoding and Snappy. We consider extending the model to incorporate data compression and compression-aware execution techniques as interesting future work.

4.3 Efficient Cost Estimation

As shown in Equation 3, in order to evaluate the objective function, we need to obtain the values of $\mathbb{G}^q$, C^q, and r_i^q. While $\mathbb{G}^q$ and C^q can be easily derived without looking at the data, it is challenging to obtain the value of r_i^q, i.e., the number of rows that query q needs to scan (after skipping) in G_i. In the following, we first show how to compute the exact value of r_i^q. Since the exact-computation approach is prohibitively expensive, we then propose an efficient estimation approach. Our experimental results in Section 7 show that the estimation approach takes $50\times$ less time to execute than the computation approach while providing high-quality estimations.

Computation Approach. To compute the exact value of r_i^q, we can actually perform the partitioning on column group G_i and see how many tuples query q reads after skipping. As discussed in Step 5 of the GSOP framework (Section 3.3), in order to partition a column group G_i, we need to perform the following steps: a) extract local features w.r.t. G_i, b) project the global feature vectors onto local feature vectors, and c) apply partitioning to G_i based on the local feature vectors. After these steps, column group G_i is horizontally partitioned into blocks, each of which is associated with union vectors as metadata. We then can obtain r_i^q by simply running query q through the metadata of these blocks and see how much data the query needs to scan. As we can see, this way of computing r_i^q is time-consuming. The cost bottleneck is step c), as it involves solving a clustering problem [39]. In the process of searching for a good column-grouping scheme (details in Section 4.4), we need to obtain r_i^q repeatedly for a large number of column-group combinations. Therefore, computing the exact value of r_i^q as a subroutine for the column-grouping search is prohibitively expensive. We next discuss how we can efficiently estimate r_i^q.

Estimation Approach. Recall that r_i^q is the number of rows query q scans in G_i after data skipping. One simple approach is to use the selectivity of q as an estimation of r_i^q. This way we could leverage the existing techniques of selectivity estimation. However, the value of r_i^q and the selectivity of q on G_i can differ dramatically, since data skipping is block-based. For example, suppose a query has a highly selective predicate, i.e., only a small number of tuples satisfy the predicate, r_i^q can still be quite large if this small number of tuples are distributed over many different blocks. For this reason, we need an estimation approach that takes into account the block-based skipping mechanism.

As mentioned in Section 4.3, partitioning the local feature vectors (step c) is the cost bottleneck of the computation approach. Thus, in our estimation approach, we only perform step a) and step b) of the compuation approach. After step b), each row in column group G_i has a corresponding local feature vector, which is a boolean vector and stores the evaluation results of all local features on this row. Instead of actually partitioning the data as step c) of the computation approach, we exploit a simple property of this partitioning process. That is, the partitioning process would always prefer to put the rows having the exactly same local feature vectors into the same block. Therefore, in step c) of our estimation approach, we simply group the rows that have the same local feature vector. Let V be the set of *distinct* vectors after grouping in G_i. For each $v \in V$, we denote by $\mathsf{count}(v)$ the number of rows whose local feature vector is v. Let b be the size of each block. We can calculate that the minimum number of blocks needed to accommodate the rows whose local feature vector is v is $\lfloor \frac{\mathsf{count}(v)}{b} \rfloor$. These blocks all have v as their *union vector*. As discussed in Section 2.2, we can check whether a query can skip a block by looking at its union vector. Specifically, for an incoming query q, we first check which features can subsume q. Then given a union vector v, we only need to look at the bits that correspond to thesse subsuming features; if there is a 0 in these bits, we can skip this block. Using this approach, given query q, we can divide V into two sets: V_{skip}^q and V_{read}^q, where V_{skip}^q consists of the vectors that q can skip and V_{read}^q consists of the vectors that q cannot skip. Thus, the minimum number of blocks that q can skip is: $\sum_{v \in V_{skip}^q} \left\lfloor \frac{\mathsf{count}(v)}{b} \right\rfloor$ Let n be the total number of rows. Since each block has b rows, we can deduce that the maximum number of rows that query q needs to scan is: $n - b \cdot \sum_{v \in V_{skip}^q} \left\lfloor \frac{\mathsf{count}(v)}{b} \right\rfloor$. We use this formula as the estimate of r_i^q. Notice that this formula provides an upper-bound of r_i^q. Intuitively, since our goal is to minimize the objective function, using an upper-bound estimation can guide us to a solution with the least upper-bound of the objective value, which hopefully would not deviate too much from the solution using the exact objective value. As shown in Section 7, using our estimation approach can signicantly speed up the column grouping process without sacrificing the quality of results.

4.4 Search Strategy

For a given table, the number of possible column grouping schemes is exponential in the number of columns. In practice, we cannot afford a brute-force approach that enumerates all the possible grouping schemes. Therefore, we adopt a bottom-up heuristic search strategy, which has been shown to be very effective in existing column grouping techniques [25]. Initially, each column itself forms a group. We then iteratively choose two groups to merge until all columns are in one group. At each iteration, we enumerate all pairs of column groups and evaluate how their merge would affect the objective function. We then pick the merge that leads to the minimum value of the objective function. Starting from c columns, we need c iterations to merge all columns into one group. After these c iterations, we pick the iteration where the objective function has the minimum value and return the grouping scheme from that iteration.

As we can see, the search process frequently invokes the objective function evaluation as a sub-routine. Specifically, since we enumerate all pairs of column groups at each iteration, we need to evaluate the objective function $O(c^2)$ times for a table with c columns. Thus, computing the exact value of this function every time is prohibitively expensive. When trying to merge every pair of groups, we use the estimation approach discussed in Section 4.3 to obtain an estimate of the objective function. After enumerating all pairs in a iteration, we select the merge that leads to the minimum value on the estimated objective function. Before starting the next iteration, however, we perform the computation approach in Section 4.3 to obtain the exact value of the objective function for this merge. This is to prevent the errors introduced by the estimation approach from being propogated to future iterations. Thus, for each iteration, we invoke the estimation approach $O(c^2)$ times and the computation approach only once. This is much faster than performing the computation approach $O(c^2)$ times for each iteration. We also observe that the obtained values of r_i^q can be reused for later iterations. As an optimization, we cache all the estimated and exact values of r_i^q we have obtained and reuse them when needed.

5. LOCAL FEATURE SELECTION

Identifying Candidate Local Features. We have obtained a set of global features generated from workload analysis. Suppose we have three global features as shown in Figure 2 and a column grouping scheme where each column itself forms a group, i.e., $G_1 = \{A\}$, $G_2 = \{B\}$, $G_3 = \{C\}$, $G_4 = \{D\}$. A simple approach is to choose, for each column group, the features that involve the columns from this column group. This way, we will choose feature `A='m'` for G_1, as this feature involves column A. Similarly, we choose `B<0` for G_2, and `C like 'y%'` for G_3. We choose no feature for G_4, since no feature involves column D. Although all the features chosen by this approach are relevant, some important features may be missing. Consider the workload in Example 2 in Section 4.1. When a query like Q_3 comes, it can only skip data in column C from group G_3 but has to read the entire column D in G_4, as we did not choose feature `C like 'y%'` for G_4. This example reveals that, for identifying candidate features, we also have to look at the co-occurrence of columns and features in the queries. For example, if we observe that in the workload the queries with filter predicate `C like 'y%'` frequently request column D in their `SELECT` statement, we may want to include `C like 'y%'` as local features for G_4.

Based on this idea, we can formally define the candidate local features as follows. Given a workload W and a column G, let $W^G \subseteq W$ be the set of queries that need to access columns in column group G. Let F be the set of global features generated through workload analysis. Given a query $q \in W$, we denote by F^q the features that subsume query q. Hence, the candidate set of local features for a column group G can be defined as: $\mathsf{CandSet}(G) = \bigcup_{q \in W^G} F^q$.

Feature Weighting and Selection. Given a set of candidate features, in order to choose the most important features, we first need to compute the importance (i.e., weight) of each candidate feature. In SOP, feature selection was modeled as a frequent-itemset mining problem, where the weight of a global feature is its occurrence frequency in the workload. For local feature selection, however, we cannot simply use this weight because the weight of a local feature should indicate its importance on a column group instead of on all columns. For this reason, we quantify the weight of a feature f w.r.t a column group G as the number of queries that are not only subsumed by this feature but also need to access the column group. Hence, we have: $\mathsf{weight}(G, f) = \left|\{q \mid f \in F^q \text{ and } q \in W^G\}\right|$. Using this formula, we can create a ranked list of local features for each column group. Note that only the features in the candidate set are considered.

We now have to determine how many features to use for partitioning each column group. Using too few features may render the partitioning ineffective, while using too many features does not improve skipping but increases the cost for partitioning. One simple way is to set a heuristic number, say 15, for all column groups. This number, however, may not be suitable for all column groups, as their ranked lists of features may have different correlation characteristics. Recall that, after the features are selected, we will evaluate these features against each tuple and generate a set of feature vectors, which will be input to the partitioning algorithm. We notice that the number of distinct feature vectors is a good indicator of whether the number of features selected is appropriate. If the number of distinct feature vectors ends up too small, we can include more features without affecting the skipping on existing features; if this number is too large, this means the existing features are already very conflicting, so adding more features would not improve skipping much. Another practical constraint is that the efficiency of the partitioning algorithm depends on the number of distinct vectors. Based on our tuning, we have found that around $3,000$ distinct vectors provide a good balance between partitioning effectiveness and efficiency across different datasets.

For a given number k, we need to quickly estimate how many distinct feature vectors will be generated if we choose the first k local features. We estimate this number by sampling the feature vectors. In the augmentation step, we have evaluated all global features and augmented each tuple with a feature vector, each bit of which corresponds to a global feature. We now take a sample of these feature vectors. For a given k, we pick the first k local features and *project* each vector in this sample on the bits that correspond to these k features. We can then obtain the number of distinct vectors by merging the vectors which have the same values on these projected bits. Using this procedure as a sub-routine, we perform a binary search to determine the value of k that can produce an appropriate number of distinct vectors (e.g., around $3,000$). After obtaining the value of k, we return the top-k features with the highest weights as selected features.

6. QUERY PROCESSING

In this section, we discuss how we process queries in GSOP.

Reading Data Blocks. We first describe the process of reading data blocks. Figure 5 illustrates this process. When a query arrives, we first check this query against the global features and see which global features subsume this query. This information is represented in a query vector. The query vector $(1, 1, 0)$ says that features F_1 and F_2 subsume this query and thus can be used for skipping data. We also extract the columns that this query needs to read and pass it to the *column grouping catalog*. The column grouping catalog stores the column grouping scheme. In the example of Figure 5, the query needs to read column A from group G_1 and columns B, D from group G_2. The catalog also maintains a *mask vector* for each column group. The mask vector of a column group encodes which local features were used for partitioning this group. For example, the mask vector of G_1 is $(1, 0, 1)$, which tells us that its local features are F_1 and F_3.

The query then goes through the actual data blocks. In Figure 5, each column group is partitioned into 2 blocks. Each block is annotated with a *union vector*. As in SOP, we decide whether to skip a block by looking at its union vector. A value 1 on the i-th bit of a union vector indicates some data in this block satisfies feature F_i. A value 0 on the i-th bit says that no data in this block satisfies feature F_i. In this case, any query subsumed by feature F_i can safely skip this block. Unlike SOP, in our new framework, a bit of union vectors can also be *invalid*. An invalid i-th bit tells us that

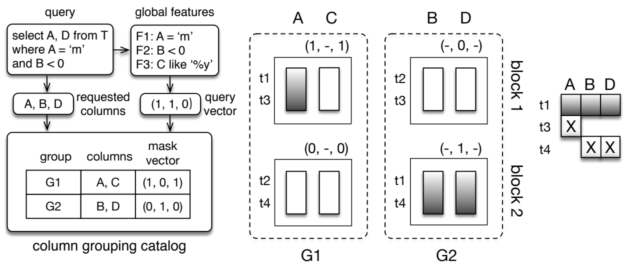

Figure 5: Query Processing

F_i is not a local feature of this column group and thus cannot be used for skipping. Therefore, all blocks of a column group should have invalid values on the same bits of their union vectors. In Figure 5, both union vectors of G_1 have their second bit invalid, which indicates that F_2 is not a local feature of G_1. In practice, we do not need a special representation for the invalid bits. The query can learn which bits of the union vector should be ignored from the column grouping catalog. To skip a block, we compute a `OR` between the query vector and its union vector. If the result has at least one bit as 0, except the invalid bits, we can skip this block. When we cannot skip a block, we read the columns requested in this block. In Figure 5, we end up reading column A from block 1 of G_1 and columns B and D from block 2 of G_2.

Tuple Reconstruction. If the requested columns of the query span multiple column groups, as shown in Figure 5, we need to assemble the columns back into tuples using their original tuple ids, as columns across different groups may not be aligned. When all the requested columns of a query are in one column group, we do not need to read tuple-ids. Before actually reading any data, the query can learn whether to read tuple ids based on the column grouping catalog. The tuple ids can be stored as a column within each block. Note that we only need to read the tuple ids in the blocks that cannot be skipped.

Once we have read the columns along with their tuple ids, we can reconstruct the tuples. As mentioned, both SOP and GSOP are applied to to each individual horizontal partition, e.g., each date partition. We assume all the tuples of a horizontal partition reside in one machine. Therefore, even in a distributed architecture, tuple reconstruction does not require shipping data across machines. We also assume that the columns to be assembled can fit in main memory. Take Hadoop Parquet [2] files as an example. Typically, each Parquet file is around 1 GB in size and has fewer than 15 million tuples. Using a Hadoop or Spark-based execution engine, this means tuple reconstruction can be handled within a single mapper.

Once the columns have been read into memory, we simply sort each of the columns based on their tuple ids. Columns within a group can be stitched first and then sorted together. After all column groups have been sorted, they can be easily stitched into tuples. Notice that we only keep the tuples for which *all* of the requested columns are present. In the example shown in Figure 5, we only return tuple t_1 while dropping t_3 and t_4, even though we have read some of their columns. This is because if we have not read a column from a tuple, that means this tuple has been skipped by some local features and thus can be safely ignored.

7. EXPERIMENTS

7.1 System Prototype

We implemented GSOP using Apache Spark and stored the partitioned data as Apache Parquet files. First, given a query log, we extracted global features as in [39]. We then used SparkSQL to

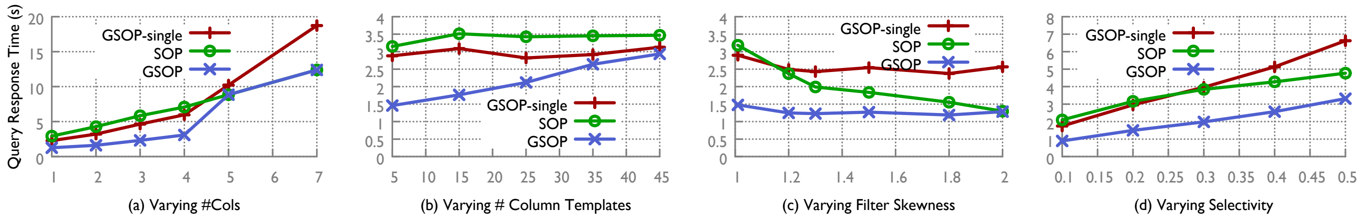

Figure 6: Query performance (Big Data Benchmark).

batch evaluate these features on a table and generate a global feature vector for each tuple. We implemented our column grouping and feature selection techniques in Scala. Finally, we performed the actual data partitioning using Spark.

Note that we do not apply GSOP on the entire table at once. Instead, we process roughly 10 million rows at a time and partition them using GSOP. The resulting partitioned data can fit in a Parquet file of roughly 1GB in size. In effect, we store a table as a set of GSOP-enabled Parquet files. In a Parquet file, data are horizontally partitioned as row-groups, and within each row group, data are organized as a set of column chunks. Each row group corresponds to a block in GSOP. Parquet files do not natively support the notion of column groups. We implemented column groups in Parquet by simply marking certain columns absent in each row group. Suppose a table has 3 columns A, B, C. If we mark C absent in a Parquet row group, then this row group becomes a block of column group A, B in GSOP. We also made optimizations so that the absent columns do not incur overhead. Since Parquet already supports per-row-group metadata, we simply added the fields needed in GSOP, such as union vectors. With predicate pushdown, a query can inform each Parquet file what columns it requests and what filter predicate it has. Then each GSOP-enabled Parquet file will use this information to skip row-groups, read the unskipped row-groups, and finally return the data as reconstructed tuples. In the prototype, we implemented most of the query processing component in Parquet internally; only minimal changes were needed in the upstream query engine, i.e., SparkSQL. We turned on the built-in compression mechanisms in Parquet, such as RLE and Snappy.

7.2 Workloads

Big Data Benchmark. The Big Data Benchmark [3] is a public benchmark for testing modern analytics systems. Due to its flexibility in query generation, we use this benchmark for sensitivity analysis by varying query parameters. We populated the data using a scaling factor of 0.1 and generated a denormalized table with 11 columns and 15 million rows. Note that this table can fit in a single Parquet file. As mentioned, even for large-scale datasets, we should apply GSOP within each individual Parquet file. Thus, our focus with this dataset is to perform micro-benchmarking on a single run of GSOP over a single Parquet file. We will perform large-scale performance tests in TPC-H and SDSS, as discussed later. This benchmark consists of four classes of queries: scan, aggregation, join, and UDF. We only used the scan queries as they are simple and thus easy for us to understand how GSOP performs under different parameter settings. We generated the scan queries in the form of:

`SELECT` $A_1, A_2, \cdots, A_k$ `FROM T WHERE filter(`B, b`)`

where `filter(`B, b`)` can be one of these three cases: $B < b$, $B = b$ and $B > b$, and b is a constant.

First, we use $0 < s < 1$ to set the selectivty of filter predicates. For example, if we set $s = 0.2$, we would only use the predicates whose seletivity is 0.2 ± 0.01 for `filter(`B, b`)` in the generated queries. Given selectivity s, we have a pool of filter predicates which satisfy this selectivity requirement. In real-world workloads, some predicates are used more often than others. We thus pick predicates from this pool under a zipf distribution with parameter z. We set k as the number of columns in the `SELECT` statement of each query. Real-world queries usually do not access column randomly. To model the column affinity in the queries, we restrict that the columns $A_1, A_2, \cdots, A_k$ can only be generated from *column templates*. We use parameter t to control the number of column templates in the workload. For example, when $t = 1$, all queries will have exactly the same k columns in their `SELECT` statement; when $t = \binom{11}{k}$, we have a template for each k-column combination, and thus the queries will in effect randomly select $A_1, A_2, \cdots, A_k$ from the 11 columns of the table.

Given a setting of s, z, k and t, we generate 100 queries for training and 100 queries for testing.

TPC-H. TPC-H [6] is a decision support benchmark consisting of a suite of business-oriented ad-hoc queries with a high degree of complexity. We choose a scale factor of 100 to generate the data. We used the TPC-H benchmark for two scenarios. Since our techniques are focused on the layout design of single tables, in the first scenario, we denormalized all TPC-H tables. We considered this resulting denormalized table of 70 columns and 600 million rows as the input to GSOP. In the second scenario, we considered the original TPC-H normalized schema as the input to GSOP and used the approach discussed in Section 3.3.

We selected ten query templates in TPC-H that have relatively selective filter predicates, namely, q_3, q_5, q_6, q_{11}, q_{12}, q_{16}, q_{19}, q_{20}, q_{21} and q_{22}. The number of columns accessed in these query templates are: 7, 7, 4, 4, 5, 4, 8, 3, 5 and 2, respectively. For each template, we generated 100 queries using the TPC-H query generator. This gave us 1000 queries in total, which we used as the training workload. We then independently generated 100 queries, 10 from each template, as test queries. The TPC-H query workload is a good example of a template-generated workload, which is very common in real-world data warehouse applications. In general, more templates used in the workload would potentially lead to higher feature conflict, where the performance benefit of GSOP over SOP would be more significant.

SDSS. Sloan Digital Sky Surveys (SDSS) [5] is a public dataset consisting of photometric observations taken on the sky. SDSS provides a SQL interface and also makes the SQL query logs publicly available [4]. We focused on a table called Star in the SDSS database server (DR7). The Star table contains the photometric parameters for all primary point-like objects in the sky. The table has over 260 million rows and 453 columns. We process 4 milion rows at a time, apply GSOP to partition these rows, and store the results in a Parquet file. We collected 2340 real queries issued on this table from 01/2011 to 06/2011. We sorted these queries by their arriving time. We used the first 3/4 queries as training workload to guide the partitioning of the Star table. We then ran the rest 1/4 queries on the table to test the effectiveness of the partitioning schemes. The mean and standard deviation of the number of columns in projections are 13.6 and 5.13, respectively. The maximum and minimum number of columns in projections are 23 and 3, respectively.

7.3 Settings

For the Big Data Benchmark, our focus is on micro-benchmarking. The experiments were conducted on an Amazon EC2 m3.2xlarge instance with Intel Ivy Bridge Processors and 80 GB of RAM, and a 2TB SSD. For the TPC-H and SDSS workloads, our focus is on large-scale query performance. The experiments were conducted on a Spark cluster of 9 Amazon EC2 i2.2xlarge instances, with 1 master and 8 slaves. Each i2.2xlarge instance is equipped with Intel Ivy Bridge Processors, 61GB of RAM, and 2× 800G SSD. Before running each query, we cleared the OS cache so that the query execution times were measured against SSD-resident data. All the query execution times were measured on an average of 3 runs.

7.4 Big Data Benchmark

Let us now discuss the results from the Big Data Benchmark. Given the workload parameters s, z, k and t, as described in Section 7.2, we vary these parameters and see how GSOP performs under different workload characteristics. By default, we set $s = 0.2$, $k = 2$, $z = 1.1$, and $d = 6$. In this set of experiments, we measure the average query response time of the 100 test queries on the data partitioned by three approaches: GSOP is our proposed framework, SOP represents the state-of-the-art partitioning framework [39] for data skipping, and GSOP-single represents our proposed extension to SOP, as discussed in Section 3.1, which partitions each column individually without column grouping. Recall that SOP and GSOP-single represent the two ends of the partitioning layout spectrum considered by GSOP. Thus, GSOP subsumes both SOP and GSOP-single. GSOP may generate the same partitioning layout as SOP or GSOP-single as appropriate. As we will see, GSOP performs no worse than either SOP or GSOP-single under all circumstances and can significantly outperform them in several settings.

In Figure 6(a), we vary the parameter k, i.e., the number of columns accessed, while the other parameters remain constant. We can see that, when the number of columns is small, GSOP-single is better than SOP. However, the cost of GSOP-single increases dramatically as k increases. This is because the cost of tuple-reconstruction overhead introduced by GSOP-single becomes dominant when k is large. When k is small, GSOP is slightly better than GSOP-single due to column grouping. When k is large, GSOP automatically switches to the same layout as SOP, which becomes the ideal layout when queries access over 70% of the columns. Note that the query response time is CPU-bound, where over 95% of the read time was spent on decompression and object parsing.

In Figure 6(b), we vary t, i.e., the number of column templates. We can see that GSOP can significantly outperform SOP and GSOP-single when t is small. A small t indicates the strong column affinity existed in the workload, which makes the column grouping provided by GSOP much more effective. When t is 45, the queries, in effect, access columns purely randomly. In this case, the benefit of column grouping in GSOP becomes marginal, but GSOP is still guaranteed to perform no worse than SOP or GSOP-single. As expected, neither SOP or GSOP-single is sensitive to t.

In Figure 6(c), we vary z, i.e., the skewness of filter usage. Notice that greater skewness in the filter usage results in *less* feature conflict. For example, if all the queries are using the exactly same filter predicate, then there would be no conflict at all. As we can see, as we increase z, SOP clearly becomes better, since the feature conflict is reduced. On the other hand, the feature conflict is already mitigated in GSOP-single and GSOP, and thus further reducing it does not improve much for GSOP-single or GSOP. GSOP constantly outperforms GSOP-single because GSOP lessens the tuple-reconstruction overhead through column grouping.

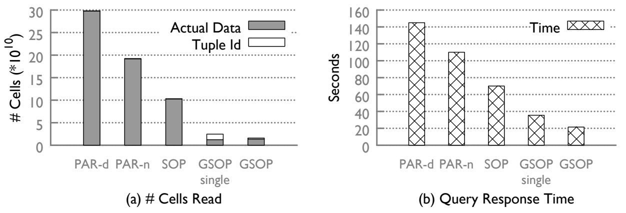

Figure 7: Query performance (TPC-H)

In Figure 6(d), we vary s, i.e., the query selectivity. Clearly, as we increase the selectivity, we see higher query execution costs in all approaches. It is interesting to note that GSOP-single is the most sensitive to the selectivity change. Recall that our objective function (Section 4.2) indicates that the tuple-reconstruction overhead depends on how much data we need to scan. While all approaches need to read more data as selectivity is increased, GSOP-single suffers more due to its increased tuple-reconstruction overhead.

The above results showed how different workload parameters may affect the data layout design. Clearly, no single static layout is the best across different settings. Thus, we need GSOP to help us automatically choose an appropriate layout based on workload and data characteristics.

7.5 TPC-H Benchmark

Query performance. We compare the performance of test queries on five different layouts. PAR-d is a baseline approach where we store the denormalized TPC-H table in Parquet. PAR-n is a baseline approach where we store the normalized TPC-H tables in Parquet. SOP, GSOP-single and GSOP represent three alternatives of our approaches as described in Section 7.4. For both normalized and denormalized scenarios, we will apply these approaches on the denormalized columns. Thus, the query performance results of SOP, GSOP-single and GSOP are the same for both normalized and denormalized scenarios and we only show them once here. Note that we issued the test queries as join queries on PAR-n and as single-table queries on the other four layouts.

In Figure 7(a), we measure the average number of actual data cells and tuple ids read by a test query. Overall, our approaches can significantly outperform the baseline approaches which rely on Parquet's built-in data skipping mechanisms. Our best approach GSOP can reduce the data read by 20× over PAR-d. Note that normalized tables are good for dimension-table-only queries. Interestingly, a significant proportion of our test workload are dimension-table-only queries, i.e., 40 out of 100. Even so, GSOP can reduce the data read by $14\times$ over PAR-n. Of our three approaches, as expected, SOP reads the most data cells but does not need to read any tuple ids. GSOP-single reads much less actual data cells due to mitigated feature conflict and improved data skipping, but has to read a lot of tuple-ids. By employing column grouping, GSOP reads much less tuple ids than GSOP-single while maintaining comparable skipping effectiveness. Note that Figure 7(a) only focuses on the amount of data read but does not factor in the CPU cost of joining the columns back into tuples. In Figure 7(b), we show the end-to-end query response time. First, PAR-n outperforms PAR-d. This is because PAR-n reads much less data, even though PAR-n involves joins in the queries. We can see that GSOP-single outperforms SOP, as the skipping benefit outweighs the tuple-reconstruction overhead for this particular workload. GSOP can significantly outperform GSOP-single due to its comparable skipping effectiveness with GSOP-single and yet much reduced tuple-reconstruction overhead. Overall, our proposed GSOP outperforms the baselines PAR-

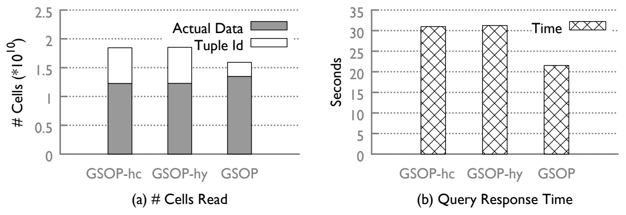

Figure 8: Query performance (TPC-H).

d and PAR-n by $6.7\times$ and $5.1\times$, respectively, and outperforms the state-of-the-art SOP by $3.3\times$.

Column grouping. In Section 4, we proposed column grouping techniques for GSOP. The GSOP framework, in general, can invoke any column grouping technique as a sub-routine. We now compare our proposed column grouping technique with the state-of-the-art. We picked two state-of-the-art column grouping techniques, namely HillClimb [32] and Hyrise [27], as they showed superior performance in a recent experimental study [25]. In Figure 8, we evaluate the query performance on the data prepared by GSOP using three different column grouping sub-routines: GSOP uses our own grouping algorithm GSOP-hc uses HillClimb and GSOP-hy uses Hyrise. For our workload, Hyrise and HillClimb generate 18 and 17 column groups, respectively, while GSOP only generates 8 groups. Let us take a close look at the column grouping results[1]. The columns l_extendedprice and l_discount always apprear together (in 400 out of the 1000 queries). Thus, all three algorithms put these two columns in the same group. However, these algorithms differ on what other columns should be grouped together with them. For instance, l_extendedprice, l_discount also co-occur with l_shipdate, l_shippriority, l_orderkey, o_orderdate, c_mktsegment in 100 queries (from template q_3) and with p_size, p_container, p_brand in another 100 queries (from template q_{19}). Given these co-occurrence patterns, Hyrise and HillClimb both chose to put l_extendedprice and l_discount alone in a column group, due to their relatively weaker correlations with other columns, but our algorithm put l_extendedprice, l_discount and l_shipdate, l_shippriority, l_orderkey, o_orderdate, c_mktsegment in the same group. While Hyrise and HillClimb only look at the column co-access patterns, our algorithm additionally incorporates feature conflict and skipping horizontal blocks.

In Figure 8(a), we can see that, by forming a smaller number of column groups, GSOP reads much less tuple ids while reading slightly more actual data. Note that Figure 8(a) does not factor in the joining cost. If we look at the end-to-end query response time in Figure 8(b), GSOP improves GSOP-hy and GSOP-hc by 35%. This is because our proposed column grouping techniques, unlike existing techniques, can effectively balance the trade-off between tuple-reconstruction overhead and skipping effectiveness involved in GSOP.

Memory Consumption. Our proposed approaches, i.e., GSOP-single, GSOP, GSOP-hy and GSOP-hc, need to assemble columns in memory when the query reads data from multiple column groups. Since we only assemble columns within each Parquet file, in the worst case, we need to hold all the data from a single Parquet file in memory. In our experiments, the data size in each Parquet file is smaller than 1G after compression and 3G before compression. In practice, however, the actual data held in memory is much smaller, as queries usually access a small subset of rows and columns. For processing our test queries, the average memory footprint for reading a single Parquet file in GSOP-single, GSOP, GSOP-hy and GSOP-hc are $1.9G$, $0.8G$, $1.5G$ and $1.5G$, respectively. GSOP-single incurs column assembly for every query that accesses more than one column. Since GSOP generates a small number of wide column groups, out of 1000 test queries, GSOP only incurs column assembly for 400 queries, while GSOP-hy and GSOP-hc need to assemble columns for 900 queries. Thus, GSOP uses less memory than GSOP-hy and GSOP-hc.

[1]Please refer to [6] for the details of the workload.

Objective function evaluation. To quantify the goodness of a column grouping scheme, we develop an objective function in Section 4.2. In the search of column grouping schemes, we need to evaluate this objective function frequently. Since computing the exact value is expensive, we proposed estimation approaches in Section 4.3. We now evaluate the efficiency and effectiveness of these approaches. In this experiment, we compare three alternatives: Full Compt. is the approach that computes the exact value of the objective function, Sel. Est. is the baseline estimation approach based on traditional selectivity estimation, and Block Est. is our proposed block-based estimation approach. Figure 9(a) shows the running time of a column grouping process with Full Compt., Sel. Est., and Block Est. as objective-evaluation sub-routines. We can see that Full Compt. takes more than a day, which is prohibitively expensive. If using the estimation approaches Sel. Est. and Block Est. instead, we can finish this process within 44 minutes. Given

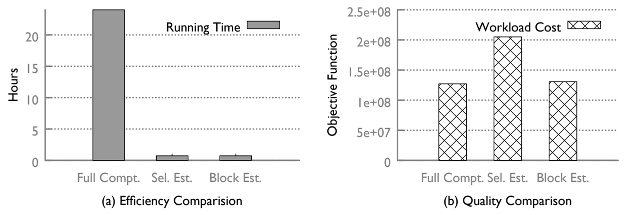

Figure 9: Objective function evaluation (TPC-H).

that the estimation approaches are much cheaper to run, we now evaluate their quality. To do this, we simply compute the exact objective value on the column grouping results generated by Full Compt., Sel. Est. and Block Est.. Recall that our goal of the column grouping is to minimize the objective function. In Figure 9(b), we can see that, using Block Est. we can produce a column grouping scheme whose objective value is almost as small as Full Compt.. On the other hand, Sel. Est. generates much worse results. This is because our proposed estimation approach incorporates the fact that data skipping is block-based, while the traditional selectivity-estimation approach is not ideal for our estimation here.

Local feature selection. Given a column grouping scheme, we need to select a set of local features for each column group. We propose techniques to automatically determine the number of local features used for each column group. For the 8 column groups we have generated in TPC-H, we observe that the number of local features selected are: 21, 48, 23, 100, 23, 100, 26, 101, 6 and 1. We find that this number can vary greatly for different groups. This result validates our argument in Section 5 that different sets of local features may have different correlation characteristics and we cannot simply set a fixed number of features for all column groups.

Loading cost. We now examine the loading costs in two scenarios. In Figure 10(a), the input data is a single denormalized table, and in Figure 10(b), the input data is a set of normalized tables. For both cases, we stored the input data in text. We view our partitioning approaches as two phases. Phase 1 is the *preparation* phase, where we perform workload analaysis, column grouping and local feature selection. In practice, Phase 1 needs to run once upfront and only needs to be re-run only when there is a dramatic change to the workload or data characteristics. Phase 2 is the *loading* phase, where we load and reorganize the actual data *within* individual Parquet files. In Figure 10(a), we compare five alternative approaches

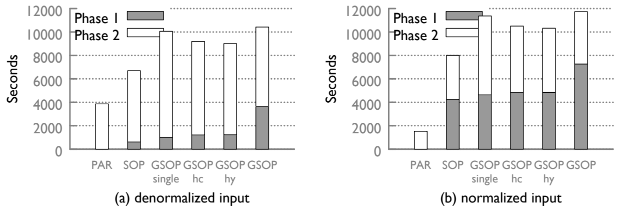

Figure 10: Loading cost (TPC-H).

and PAR-d, which is a baseline cost of simply loading text into Parquet. GSOP spends the most time in Phase 1, because GSOP considers more information in column grouping. The cost of Phase 2 depends on the number of column groups, as we need to run a partitioning algorithm for each individual column group. Thus, SOP has the cheapest Phase 2 and GSOP-single has the most expensive Phase 2. Phase 2 of GSOP is cheaper than GSOP-hy and GSOP-hc, as GSOP generates a smaller number of columns groups. In Figure 10(b), we consider the case when the input is a set of normalized tables. To apply our approaches, we have to perform an extra step of partial denormalization (as part of Phase 1).Overall, for the denormalized scenario, GSOP takes $2.6\times$ the time as the baseline. Since data loading is an offline and one-time process, we believe that there are many applications for which this overhead is worth improving the query performance by $6.7\times$. When the input data is normalized, GSOP takes $7.6\times$ as much time as simply loading the normalized data, while providing a $5.1\times$ query performance improvement. We leave as future work the layout design techniques that support normalized tables without partial denormalization.

7.6 SDSS

We now examine the performance of GSOP on a real-world workload. In Figure 11, we plot the average query response times of 600 test queries against a baseline approach of using Parquet built-in data skipping mechanisms and five partitioning approaches. For this workload, GSOP-single performs better than SOP. Also, it is interesting to see that GSOP-hy and GSOP-hc exhibit quite different performance, and GSOP-hy is even worse than SOP. Since these techniques do not take into account feature conflict or horizontal skipping, their performance is highly unreliable. Note that GSOP-hy and GSOP-hc generate 8 and 20 column groups, respectively, while GSOP generates only 2 column groups. We also see that GSOP improves GSOP-single by only 30%. The reason why GSOP-single performs well for this workload is that most of the queries were concentrated on a very small set of columns and the tuple-reconstruction overhead is small. After all, GSOP outperforms the baseline by $4.7\times$ and the state-of-the-art SOP by $2.7\times$.

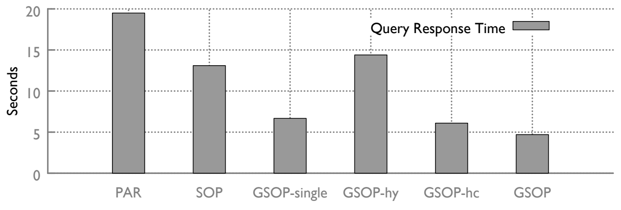

Figure 11: Query performance (SDSS).

8. RELATED WORK

In this section, we review related work.

Data Skipping. Partition pruning has been an important technique in database systems, where queries can skip partitions based on their partition key ranges. As a generalization, some early work proposed to store *small materialized aggregates* [21,31], such as min and max values of each column, to enable data skipping. Nowadays, most analytics systems have adopted data skipping. Examples include Amazon Redshift [8], Google Powerdrill [1], Hive [43], IBM DB2 [40], Parquet [2], Vertica [11], Snowflake [16] and so on. Data skipping can be used to reduce data scan whether the underlying data layout is row-major or column-major. These systems mostly rely on range partitioning to generate data blocks. Amazon Redshift also uses z-order instead of hierarchical range partitioning in order to given multiple columns equal chances of skipping. Recent work [39] proposed a fine-grained partitioning framework for data skipping. Our framework generalizes this work by allowing different columns to adopt different partitioning schemes.

Horizontal Partitioning. Many research efforts have been devoted to workload-driven physical design (e.g., [34,35,37]). Such work aims for automatic physical design, such as indexes and materialized views [34], based on workload information. While workload-driven horizontal partitioning techniques have been studied [15,33, 35,45], they were built on top of range partitioning or hash partitioning. Instead of specifying which columns to range- or hash-partition on, GSOP is based on features, or representative filters, extracted from the workload. GSOP seeks to operate at a finer granularity than these traditional horizontal partitioning techniques. In fact, it is a good practice to apply GSOP on each individual range partition as a secondary partitioning scheme. Also, GSOP does not move tuples across machines. Schism [18] is also a workload-driven fine-grained partitioning technique, but it is designed for reducing cross-machine transactions for transactional workloads.

Vertical Partitioning. Vertical partitioning divides the columns of a table into groups. As an important database technique, vertical partitioning has been studied extensively [10,14,25,27,30,32,35,37, 46]. Most of the existing vertical partitioning techniques focus on the trade-off between a row store and a column store: when the table has many narrow vertical partitions, it resembles a column store, in which case the queries that access multiple partitions suffer from the cost of tuple reconstruction. When the table has a few wide vertical partitions, it resembles a row store, in which case the queries that assess a small number of columns suffer from the cost of reading unwanted columns. Thus, existing techniques base the partitioning decision on column co-access patterns in a workload. In contrast, our column grouping technique takes into account the opportunities of skipping horizontal blocks and balances the trade-off of skipping effectiveness vs tuple-reconstruction overhead.

Physical Design in Column Stores. Column store has become the mainstream architecture for analytics systems (e.g., [2,8,17,26, 40,43]). In a column store, columns can form column groups [11, 12,28]. Different from the vertical partitioning problem mentioned above, where a vertical partition is a row store, each column group here is still a column store. In GSOP, we adopt a PAX-style layout [7] within a column group: we horizontally partition each column group into blocks (e.g., of 10k rows) and use columnar layout within each block. Existing approaches [11,28] allow different column groups to choose different orders for efficient read or compression purposes. The main difference between GSOP and these approaches is that GSOP focuses on fine-grained skipping-oriented partitioning instead of simply choosing column-level sort orders.

Database cracking [23,24,38] studies the problem of automatically and adaptively creating indexes on a column store during query processing. While it is similar to our problem in several aspects, such as involving re-ordering of columns and balancing reading cost and tuple-reconstruction overhead, the problem GSOP targets is fundamentally different from cracking. The application scenario for database cracking is when we have no access to past workloads or the luxury of paying an upfront cost of organizing data. GSOP, to the contrary, is designed for the data warehouse scenarios where we can perform a statistical analysis on workloads and use this information to organize the data at data loading time.

Columnar Storage in Hadoop. Columnar layouts have been widely adopted in Hadoop. RC Files [41] is a PAX-style layout [7] for HDFS, where data is horizontally partitioned into HDFS blocks and each block uses columnar layouts internally. ORC Files [43] and Parquet [2] also adopted the PAX-style layout with performance optimizations over RC Files. Floratou et al. [22] proposed a pure columnar format for HDFS. Unlike a PAX-style layout, their solution allows different parts of a row to span different HDFS blocks, but makes sure these blocks reside in the same machine by modifying HDFS block placement policy. See [42] for a performance study on these HDFS formats. These formats commonly have built-in skipping mechanisms, sometimes known as *predicate pushdown*. We can apply our techniques to organize data stored in these formats and leverage their built-in skipping mechanisms.

9. CONCLUSION

The GSOP framework generalizes SOP by employing two new components: column grouping and local feature selection. We evaluated the effectiveness of GSOP using two public benchmarks and a real-world workload. The results showed that GSOP can always find a partitioning layout no worse than SOP and can dramatically outperform SOP in many settings. In particular, in the TPC-H benchmark, GSOP improved the query response time by $3.3\times$ over SOP. GSOP is a good example of boosting query performance by exploiting workload analysis and flexible data layouts. Along these lines, in the future, we plan to incorporate advanced statistical analysis on query workloads and consider even more layout options, such as replication. We are also interested in how these flexbile layout designs can be adaptive to workload or data changes.

Acknowledgements. We thank Endre Palatinus, Alekh Jindal and Jens Dittrich for providing the source codes in [25]. We also thank Ani Thakar for helping us work with the SDSS workload. We are grateful to the reviewers for their insightful feedbacks. This research is supported in part by DHS Award HSHQDC-16-3-00083, NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Apple Inc., Arimo, Blue Goji, Bosch, Cisco, Cray, Cloudera, Ericsson, Facebook, Fujitsu, HP, Huawei, Intel, Microsoft, Mitre, Pivotal, Samsung, Schlumberger, Splunk, State Farm and VMware.

10. REFERENCES

[1] Apache Drill. `https://drill.apache.org`.
[2] Apache Parquet. `http://parquet.apache.org`.
[3] Big Data Benchmark. `amplab.cs.berkeley.edu/benchmark`.
[4] CasJobs. `http://skyserver.sdss.org/casjobs/`.
[5] Sloan Digital Sky Surveys. `http://www.sdss.org`.
[6] TPC-H. `http://www.tpc.org/tpch`.
[7] A. Ailamaki *et al.* Data page layouts for relational databases on deep memory hierarchies. *VLDB Journal*, 11(3):198–215, Nov. 2002.
[8] A. Gupta *et al.* Amazon Redshift and the case for simpler data warehouses. In *SIGMOD*, pages 1917–1923, 2015.
[9] A. Hall *et al.* Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, 2012.
[10] A. Jindal *et al.* Trojan data layouts: Right shoes for a running elephant. In *SOCC*, pages 21:1–21:14, New York, NY, USA, 2011.
[11] A. Lamb *et al.* The Vertica analytic database: C-Store 7 years later. *VLDB*, 5(12):1790–1801, 2012.
[12] D. Abadi, D. Myers, D. DeWitt, and S. Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, pages 466–475, April 2007.
[13] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.
[14] I. Alagiannis, S. Idreos, and A. Ailamaki. H2O: A hands-free adaptive store. In *SIGMOD*, pages 1103–1114, New York, NY, USA, 2014. ACM.
[15] B. Bhattacharjee *et al.* Efficient query processing for multi-dimensionally clustered tables in DB2. In *VLDB*, pages 963–974, 2003.
[16] B. Dageville *et al.* The Snowflake elastic data warehouse. In *SIGMOD*, pages 215–226, 2016.
[17] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, pages 225–237, 2005.
[18] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3:48–57, 2010.
[19] D. Abadi *et al.* Integrating compression and execution in column-oriented database systems. In *SIGMOD*, SIGMOD, pages 671–682, 2006.
[20] D. Abadi *et al.* The design and implementation of modern column-oriented database systems. *Foundations and Trends in Databases*, 5(3), 2013.
[21] D. Ślęzak *et al.* Brighthouse: An analytic data warehouse for ad-hoc queries. *PVLDB*, 1(2):1337–1345, 2008.
[22] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *PVLDB*, 4(7), 2011.
[23] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.
[24] S. Idreos, M. L. Kersten, and S. Manegold. Self-organizing tuple reconstruction in column-stores. In *SIGMOD*, pages 297–308, 2009.
[25] A. Jindal, E. Palatinus, V. Pavlov, and J. Dittrich. A comparison of knives for bread slicing. *PVLDB*, 6(6):361–372, 2013.
[26] M. Armbrust *et al.* Spark SQL: relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.
[27] M. Grund*et al.* Hyrise: A main memory hybrid storage engine. *PVLDB*, 4(2):105–116, Nov. 2010.
[28] M. Stonebraker *et al.* C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.
[29] M. Zaharia *et al.* Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, 2012.
[30] M. Zukowski *el al.* DSM vs. NSM: CPU performance tradeoffs in block-oriented query processing. In *DaMoN*, pages 47–54, 2008.
[31] G. Moerkotte. Small materialized aggregates: A light weight index for data warehousing. In *VLDB*, pages 476–487, 1998.
[32] R. Hankins*et al.* Data morphing: An adaptive, cache-conscious storage technique. In *VLDB*, pages 417–428. VLDB Endowment, 2003.
[33] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.
[34] S. Agarwal *et al.* Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505, 2000.
[35] S. Agrawal *et al.* Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.
[36] S. Melnik *et al.* Dremel: interactive analysis of webale datasets. *PVLDB*, 3(1-2):330–339, 2010.
[37] S. Papadomanolakis *el al.* AutoPart: Automating schema design for large scientific databases using data partitioning. In *SSDBM*, pages 383–392, 2004.
[38] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *PVLDB*, 7(2):97–108, Oct. 2013.
[39] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, pages 1115–1126, 2014.
[40] V. Raman *et al.* DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.
[41] Y. He *et al.* RCFile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *ICDE*, pages 1199–1208, 2011.
[42] Y. Huai *et al.* Understanding insights into the basic structure and essential issues of table placement methods in clusters. *PVLDB*, 6(14), 2013.
[43] Yin Huai *et al.* Major technical advancements in Apache Hive. In *SIGMOD*, pages 1235–1246, 2014.
[44] Z Liu *et al.* JSON data management: Supporting schema-less development in rdbms. In *SIGMOD*, pages 1247–1258, New York, NY, USA, 2014.
[45] J. Zhou, N. Bruno, and W. Lin. Advanced partitioning techniques for massively distributed computation. In *SIGMOD*, pages 13–24, 2012.
[46] J. Zhou and K. Ross. A multi-resolution block storage model for database design. In *IDEAS*, pages 22–31, July 2003.