Skipping-oriented Data Design for Large-Scale Analytics

Liwen Sun

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-203
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-203.html

December 12, 2017

Skipping-oriented Data Design for Large-Scale Analytics

by

Liwen Sun

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

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Michael J. Franklin, Co-chair
Professor Ion Stoica, Co-chair
Professor Joshua Blumenstock

Fall 2017

Skipping-oriented Data Design for Large-Scale Analytics

Abstract

Skipping-oriented Data Design for Large-Scale Analytics

by

Liwen Sun

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Michael J. Franklin, Co-chair

Professor Ion Stoica, Co-chair

As data volumes continue to expand, analytics approaches that require exhaustively scanning data sets become untenable. For this reason, modern analytics systems employ data skipping techniques to avoid looking at large volumes of irrelevant data. By maintaining some metadata for each block of data, a query may skip a data block if the metadata indicates that the block does not contain relevant data. The effectiveness of data skipping, however, depends on how the underlying data are organized into blocks.

In this dissertation, we propose a fine-grained data layout framework, called "Generalized Skipping-Oriented Partitioning and Replication" (GSOP-R), which aims to maximize query performance through aggressive data skipping. Based on observations of real-world analytics workloads, we find that the workload patterns can be summarized as a succinct set of features. The GSOP-R framework uses these features to transform the incoming data into a small set of feature vectors, and then performs clustering algorithms using the feature vectors instead of the actual data. A resulting GSOP-R layout scheme is highly flexible. For instance, it allows different columns to be horizontally partitioned in different ways and supports replication of only parts of rows or columns.

We developed several designs for GSOP-R on Apache Spark and Apache Parquet and then evaluated their performance using two public benchmarks and several real-world workloads. Our results show that GSOP-R can reduce the amount of data scanned and improve end-to-end query response times over the state-of-the-art techniques by a factor of 2 to 9.

To my family

Contents

List of Figures

List of Tables

Acknowledgments

I am heartily thankful to my advisor, Michael Franklin, for his guidance, patience, encouragement and support throughout my graduate school career. Mike has always provided me the freedom and flexibility to pursue my own research projects. On the other hand, he has never been short of great research ideas or insightful feedback, which are built upon his decades of experience in the field. Some of his ideas and feedback were so profound that I did not fully grasp initially, and I remember several occasions over the course of my Ph.D. studies that I had the epiphany: "this is exactly what Mike told me before!". I am also grateful to Mike for insisting on high standards for my depth of thinking as well as my writing and oral presentation skills, which I believe will tremendously benefit my career beyond graduate school.

I would also like to acknowledge my research collaborators. Jiannan Wang has been a great collaborator and friend. Jiannan sat next to me when he was a post-doctoral researcher in the AMPLab. He was always available when I needed to discuss, even though he was busily involved in multiple projects. Jiannan has an innate ability to identify and modularize challenging problems involved in research, which has been an invaluable resource for me. I also enjoyed collaborating with Eugene Wu when he was a post-doctoral researcher in the lab. As a young researcher, Eugene has a surprisingly board knowledge of the field, which enabled him to give me refreshing perspectives on various research topics. I am also thankful to my fellow graduate students Sanjay Krishnan and Reynold Xin for the inspiring discussions we had and for their contributions to the early work of this dissertation.

I would like to express my gratitude to the professors who served as my committee members and examiners during my graduate studies. I thank Ion Stoica, Ali Ghodsi, Ray Larson and Joshua Blumenstock for serving on my qualifying exam and dissertation committee, whose feedback has greatly shaped this dissertation. I also thank Joe Hellerstein and Eric Brewer for hosting my database preliminary exam.

I spent two summers interning in the industry, where I was fortunate to work with awesome intern mentors. Shawn Jeffery was my intern mentor at Groupon. He offered me the opportunity to join project Arnold at its early stage, which gave me first-hand experience in the planning, design and development of a real-world data integration system. Dilip Joseph was the mentor for my internship at Facebook. He provided great help and guidance with testing and applying my dissertation research in Facebook data warehouse, a.k.a., one of the largest data warehouses on the planet! I am grateful to both intern mentors for their tireless support.

I have never taken for granted the opportunity of being a member of the AMPLab and EECS community at Berkeley, where I have received tremendous help and collaborative support from many brilliant individuals. I thank Sameer Agarwal, Ameet Talwalker and Di Wang for sharing their expertise to improve my research. Neil Conway, Bill Marczak, Gene Pang and Beth Trushkowsky helped me practice the database preliminary exam. Peter Bailis, Dan Haas, Evan Sparks, Shivaram Venkataraman, Zhao Zhang and other AMPLab members have provided helpful feedback for my various practice talks. I have also received administrative support from the fantastic staff at AMPLab and Berkeley EECS: Kattt Atchley, Carlyn Chinan, Jon Kuroda, Boban Zarkovich, Xuan Quach, Audrey Sillers and Angela Waxman.

I would not have been admitted to the Berkeley Ph.D. program without the research training received at the University of Hong Kong. I am especially indebted to my M.Phil. thesis advisors, Reynold Cheng and David Cheung, and other faculty members in the database group at the University of Hong Kong: Ben Kao and Nikos Mamoulis. They introduced me to the wonderful world of database research.

My dissertation 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.

I owe my deepest gratitude to my family and friends for their love, care and support.

Chapter 1

Introduction

Modern applications, ranging from business decision making to scientific discovery, are increasingly becoming data-driven. As data volumes for these applications continue to grow, the efficiency of large-scale data processing becomes crucial for unlocking insights from enormous data in a timely manner. For interactive analysis, data analysts and scientists pose queries over large datasets and expect to receive responses in near real-time. In the batch analytics scenario, a goal is to minimize resource consumption, such as CPU and disk I/O, for performing analytics tasks so that the cost of running the data infrastructure remains manageable. In both cases, the efficiency of data processing, in terms of either latency or compute-resource consumption, is of tremendous value in this data-driven age.

1.1 Efficiency in Data Processing

Designers of modern analytics systems are striving to identify opportunities for improving the performance of data processing. One dimension of this effort focuses on improving the *throughput* of data scanning, where the goal is to process a given amount of data more quickly. For example, by keeping the data in RAM memory with fault-tolerance mechanisms [44], accessing these data can be much faster than reading data off hard-drive disks. Parallelization can also greatly reduce the latency for many tasks, especially for those that are *embarrassingly parallel*. In highly scalable data processing frameworks, such as Hadoop and Spark, the degree of parallelization is easily configurable for particular tasks. Another way to improve the read throughput is through data compression. As analytics systems often store the data in a column-oriented fashion [43], where values of the same type are stored contiguously, the data exhibits a high compression potential. Compression saves I/O cost as it reduces the size of raw bytes to be read, but it incurs the overhead in CPU cost for decompressing the data. This trade-off has been shown to be worthwhile in most cases [43]. In fact, some modern systems [25, 5] support compression-aware query execution, i.e., the ability to process some classes of queries without the need of decompressing data first.

Another dimension of improving the efficiency of data processing is to reduce the need of data

access. Most analytics tasks do not need to scan the dataset in its entirety. Each individual query is usually concerned with a certain aspect of the data and thus can be answered by going through only a subset of rows and columns. Many techniques have been proposed to exploit this nature. To prevent a query from reading irrelevant columns, columnar-oriented data layouts have been widely adopted in modern analytics databases, where each column is stored separately. Avoiding access to unnecessary rows can be much more difficult than achieving the same for columns, as the set of columns is mostly fixed in a table while the set of rows is ever growing. For the same reason, the rows needed by a query cannot be explicitly expressed, which can be only known based on the *predicates* of the query. In some scenarios, however, the preciseness of the query results is not necessary and only an approximation is sufficient. For these scenarios, approximate query processing techniques (e.g., [57]) can be leveraged. By reading only a sample of the data, these techniques can obtain an approximate query result much more quickly and thus avoid the need of scanning every row for a precise answer.

However, in many cases, it is important to reduce the unnecessary access of data for efficiency but without sacrificing the preciseness of the query results. Along this line, data skipping is a promising technique that has been gradually employed in modern analytics system. Examples of systems that employ special methods for data skipping include Amazon Redshift [2], IBM DB2 Blu [68], Vertica [5], Google Powerdrill [3], Snowflake [16], InfoBright [27], and some Hadoop file formats [73, 14]. In the next section, we discuss how existing systems use data skipping. We then discuss how the methods developed in this dissertation can improve the effectiveness of data skipping over the current state-of-the-art.

1.2 What is Data Skipping

Partition pruning is an important data skipping technique in traditional database systems, such as Oracle, Microsoft SQL server and Postgres. In a data warehouse environment, it is a common practice to horizontally partition the table into partitions. A major benefit of doing so is on the ease of management of tables. The data in a data warehouse typically go through an Extract-Transform-Load (ETL) process and are inserted to the table in batches. By horizontally partitioning the table, the new rows can be simply appended to the table as a new horizontal partition. As queries are concerned with the more recent data, the old partitions can also be "rolled-out" to save storage space without affecting the other partitions.

For query performance the benefit of horizontal partitioning comes from partition pruning. Tables are typically partitioned on time-related columns, such as dates. Not surprisingly, most queries are often interested in the records of a certain (recent) period instead of all the historical records. Thus, queries commonly use time ranges in their filter predicates. For such queries, instead of scanning the entire table, partition pruning allows the query to check the date ranges of each partition and thus decide which partitions can be *pruned* before the table scan. Figure 1.1 shows an example of partition pruning. The table *Sales* contains 4 date partitions. In each partition, all records share the same value on the date column *dt*, which is marked as metadata for each partition. When a query arrives, it can first check its predicate on *dt* against these metadata

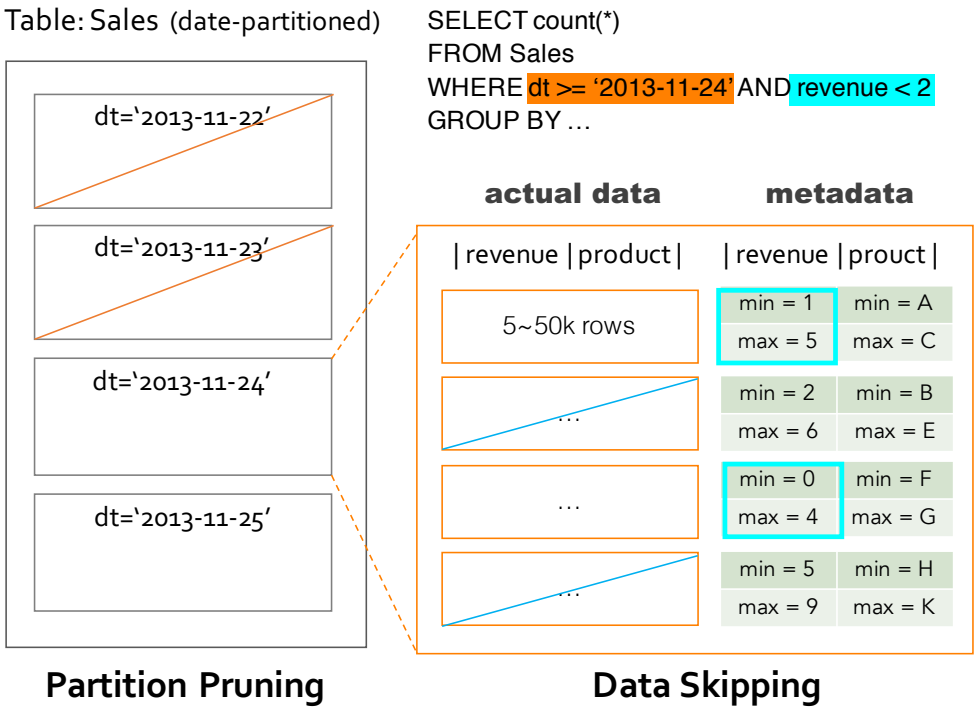


Figure 1.1: Example of Data Skipping

before actually scanning the table. A simple check would show that the query in Figure 1.1 can safely prune the first two partitions of table *Sales* and only needs to scan the last two partitions. While partition pruning provides an effective way to prune data, the remaining partitions can still contain a lot of tuples.

To build upon the idea of partition pruning, data skipping was first proposed in [48] and has been gradually adopted by most analytics systems [34, 27, 5, 3, 70, 68, 2, 16]. The idea of data skipping is to maintain metadata for all columns, not just the date column, for data *blocks*, which are typically maintained at a finer-granularity than horizontal partitions. A horizontal date partition can be further divided into a set of fairly small blocks (e.g., 1,000's or 10,000's of tuples). Each block can be associated with some metadata such as min and max values for all columns in the block. Before scanning a block, data skipping first evaluates the query filter against this metadata and then decides if the block can be skipped, i.e., the block does not need to be accessed. Unlike partition pruning, block-based data skipping can skip data even if the filter predicate is not on the partitioning column. Figure 1.1 shows data skipping in action inside partition *dt='2013-11-24'* of table *Sales*. This date partition is segmented as four horizontal blocks. For each block, the table maintains the min and max values for both columns *revenue* and *product*. Apart from the predicate on column *dt*, the query has another predicate *revenue < 2*. Thus, before scanning all 4 blocks in this date partition, data skipping allows the query to check this predicate against the block-level metadata. After the check, the query knows that the first block and the third block can

be safely skipped, as their metadata, specifically the min values of the revenue column, indicate that their records do not satisfy the predicate *revenue < 2*.

In a sense, data skipping was similar to indexing (e.g., B+-tree) in traditional database systems, as they both provide a mechanism to navigate large-scale datasets. As compared to indexing, data skipping has several salient features that make it suitable for modern analytics systems. First, even if queries can skip blocks, any single block is large enough to enable sequential scans. Traditional tree-structured indexes incur random disk accesses. In data skipping, a query goes through every block and always performs sequential scans, where the metadata check incurs minimal overhead. Second, the blocks in data skipping are large enough to hold a columnar layout internally, which is preferred for analytics workloads, whereas most traditional tree indexes are constrained to row-oriented layouts. Third, the storage overhead for data skipping (i.e., block-level metadata) is much smaller than that of traditional indexes.

Overall, data skipping speeds up table scans by reducing the need for data access, which can improve the efficiency of data processing whether the metric is latency or resource consumption. It not only saves disk I/Os but also reduces the CPU cost involved in processing the data, such as deserialization, decompression and per-row predicate evaluation. Therefore, data skipping is beneficial no matter on what hardware medium (e.g., disk, memory or SSDs) the data resides.

1.3 Physical Layout Design for Data Skipping

The opportunities for data skipping highly depend on how the data are organized into blocks. Specifically, it is desirable that the rows that a query may need to read are clustered in a small number of blocks so that the irrelevant rows can be skipped altogether. The approaches used to organize data into blocks can be summarized as follows:

Range partitioning

As described above, range partitioning has been mainly used for partition pruning. A range partitioning scheme requires the input of one or a few columns that are typically filtered on by the queries as well as the hierarchy of partitioning on these columns, i.e., which column is partitioned first and which is the second, and so on. The number of partitions and the partitioning ranges on each of these columns also have to be specified. Examples of systems that use range partitioning for data skipping include Amazon Redshift [2], and Google Powerdrill [3]. While range partitioning has been a useful technique to divide tables into a set of large partitions, such as date partitions, it may not be ideal for generating blocks for data skipping, as it has to take into account the data skew, workload skew and inter-column correlations. These issues cannot be overlooked when the table needs to be partitioned into a large number of fine-grained blocks.

Sorting

Sorting has been considered as an important mechanism in column-oriented data storage. Sorting on a column can greatly speed up navigation for queries and also make the data compress better.

Similar to range partitioning, sorting can be specified on one or more columns, and when multiple columns are involved, an order has to be enforced on these columns so that a secondary sorting column can take affect only when there is a tie in the primary sorting column. When rows are sorted, they are simply split into equal-size blocks for data skipping. An example of the systems that use sorting for data skipping is Vertica [5]. Similar to the problem of range partitioning, sorting-then-splitting is typically performed at the column level, which is too coarse-grained to generate blocks for data skipping. For example, this approach cannot express that only some value range of a column needs to be sorted while the other values do not. It is difficult for sorting to capture inter-column correlations as well.

Splitting based on Natural Order

As opposed to the above approaches, some systems do not re-organize the data for data skipping. They simply split the rows and put them into equal-sized blocks as the rows are appended to the system. In effect, this approach only counts on the natural ordering of the data for skipping blocks. Examples of these systems include IBM DB2 [68] and Hadoop file formats such as ORC [73] and Parquet [14]. The problem with such an approach is that data skipping is limited to the columns that are correlated with the natural ordering of the data, e.g., time-related columns. The chance of skipping blocks based on other columns can be very low. The advantage of such an approach, however, is that it is simple and does not need to re-organize the data at load time, which involves a faster data loading phase than the other approaches.

While these simple data skipping approaches have served as a main mechanism to navigate large datasets, the existing systems have not adopted physical layout design techniques that can realize the full potential of data skipping. They either use the traditional techniques that are developed for other purposes, such as range partitioning, or choose not to re-organize the data at all, which gives up the opportunity of trading the offline loading cost for online query performance. Moreover, these systems are limited to horizontal partitioning of the data, while there are many layout possibilities that can potentially benefit data skipping, such as vertical partitioning, a hybrid of horizontal and vertical partitioning, and replication.

To this end, this dissertation aims to develop innovative physical layout design frameworks that can realize the full potential of data skipping and thereby improve query efficiency. In the next section, we list the main contributions of this thesis in the development of skipping-oriented layout design frameworks.

1.4 Main Contributions

In this section, we present the main contributions of this dissertation.

1.4.1 Feature-driven Workload Analysis

The goal of physical data layout design is to organize data in a way that can facilitate data access. The most effective way to understand how queries access data is to analyze the workload. A

key contribution in this dissertation is the development of a workload-driven technique for data layout design that leverages representative filter predicates in the query workload. In modern analytics databases, the query workload can be obtained as history query logs or as query templates, which can be used to generate queries by filling in the parameter values. While workload-driven physical design has been extensively studied in the database literature [56, 9, 35], in this dissertation, we focus on analyzing the usage of filter predicates and how they can be used for facilitating data skipping.

In the analysis of real-world workloads, we observe two important properties of filter predicate usage. The first property is *filter skewness*, which states that a small number of filter predicates are commonly used by most of the queries. This suggests that if the data layout design is focused on this small number of frequently used predicates, most queries in the workload will be able to benefit. The second property is *filter stability*, which states that only a few brand new predicates are introduced over time and that most of the predicates are recurring. This latter observation suggests that we can utilize the patterns of a past workload to guide the data layout design and thus help future queries better skip data.

Motivated by the above properties observed from real-world workloads (See Chapter 3 for details), we propose to extract *features* from workloads. A feature is one or a set of filter predicates that are frequently used in the workload. Note that a feature captures the complete information of a filter predicate, including the columns, operators, and constants. For example, $price > 3$ is a feature, which is composed of the column *price*, the operator >, and the constant 3. We extract a filter as a feature if it *subsumes* a sufficient number of queries in the workload. The notion of subsumption is important in the context of data skipping. For example, $price > 3$ subsumes $price > 5$, because $price > 3$ is a more general condition. In this case, the data that can be skipped by $price > 3$ can also be skipped by $price > 5$. The idea of feature-driven workload analysis is to summarize the workload using a succinct set of features, which can subsume most of the queries in the workload. The physical layout design can then be centered around the data skipping for these features.

1.4.2 Fine-grained and Flexible Layouts

Existing systems that use data skipping only consider horizontal partitioning in their layout design. Even though data skipping was first designed to skip blocks of rows, it can also be generalized to skip any block of data, i.e., a block composed of any subset of columns and rows. Let us refer to the intersection of a row and a column as a *data cell*. A table can be simply viewed as a set of data cells. Ideally, we should be able to pack the data cells as blocks in flexible ways as long as the effectiveness of data skipping can be maximized. This means that we can break some conventional constraints in designing data layouts. An example of such a constraint is that the data cells from the same row should be put in the same block. The rationale behind this constraint is that, if the data cells from the same row reside in different blocks, reconstructing these data cells back into rows would incur significant overhead. This is a reasonable argument in many cases, but should be revisited in the context of aggressive data skipping. For example, reading 10% less

data may not justify the overhead of row-reconstruction, but reading 90% less data is a different story.

In this dissertation, a key contribution is to consider highly flexible and fine-grained layout designs, which incorporate a wide range of layout options for improving data skipping, such as horizontal partitioning, column grouping, and replication. In these designs, a column or a row is no longer an atomic unit in the partitioning and replication. A block can contain parts of rows and parts of columns. Some data cells of a block may be replicated in other blocks and are thus organized in a different manner. The reason why we allow such fine-grained and flexible layout designs is to maximize data skipping opportunities. Depending on the query type, data skipping can play an important role in the overall query cost. While the layout designs are centered around data skipping, they also have to factor in other costs in query performance as well as in storage cost. The main overhead to query performance due to the flexibility in layout designs is in row-reconstruction. As different columns may have orders and may be replicated in different ways, queries will incur additional overhead of reconstructing these data cells back into the original rows before they can be sent to the subsequent stages of the query processing, such as group-bys. Since the storage cost is not free, the layout designs also has a goal of finding the most efficient way of data replication, which trades space cost for query cost.

1.4.3 Optimization Problem Formulation

A third contribution of this dissertation is that we formulate optimization problems in order to balance the trade-offs in fine-grained and flexible layout designs. For the partitioning scheme, the objective function of the optimization is to minimize the query cost, which is composed of the data scan cost and tuple-reconstruction cost. For the replication scheme, the objective function of optimization is to minimize the query cost for a given storage budget. In constructing these objective functions, we prioritize simplicity and generality over accuracy in modeling the real-world costs. For example, we model the scan cost of a query as the number of data cells read. This is a simple and general model, but may not be accurate in a real-world setting, as the data may be of different types and stored in different compression schemes. After all, our goal is not to model the cost as accurately as possible, but to serve as a guide in the search for the best data layout design. Even for such simple objective functions, however, it turns out to be prohibitively expensive to evaluate them on a given layout design. We develop methods to efficiently and accurately *estimate* these objective functions instead of directly evaluating them. Given the flexibility and fine-granularity of the layout designs, there is a huge search space. We develop efficient search algorithms to find physical layout schemes using the objective functions as a guide.

1.4.4 Predicate-based Metadata and Skipping

The metadata used for data skipping in existing systems are statistics about the underlying data. The most common statistics used are min and max values. Some systems also support bitmaps, dictionaries, and bloom filters. These metadata can only support value-based skipping, i.e., when the query predicate asks for a certain value or value range. While such predicates are very com-

monly used, modern analytics workloads involve various kinds of more complex filter predicates and user-defined functions (UDF). Examples of the predicates that cannot take advantage of these value-based skipping mechanisms include string matching (e.g., *message like '%iphone%'*), inter-column comparison (e.g., *page_viewer_id <> page_creator_id*), or any general user-define functions (UDF) (e.g., *get_json_object(jstr, 'conn_type') = 'wifi'*).

In this dissertation, we introduce *predicate-based* metadata and skipping mechanism. First, we maintain the workload features, or frequent filter predicates, used in the layout design. Second, we keep a bit vector called the *block vector*, where each bit corresponds to one feature and indicates whether this block has some data that satisfy this feature. This approach supports data skipping for any kind of filter predicates. A query can have a great chance of skipping data as long as the query predicate is *subsumed* by one of the features, which means the query predicate is a more general condition than one of the features. Since these features together can subsume most of the queries, there is a high chance that a query can take advantage of this predicate-based skipping mechanism. Note that our predicate-based skipping mechanism is designed to work in conjunction with value-based skipping.

1.4.5 Three Frameworks for Data Layout Design

The main contribution of this dissertation is the development of the following three frameworks for data layout design:

The skipping-oriented partitioning (SOP) framework is a workload-driven horizontal partitioning framework with the goal of maximizing data skipping. SOP lays the foundation for the entire work of this dissertation, as the more advanced frameworks developed in later chapters are built upon SOP. We first analyze a real-world workload and observe two important properties, which motivate the design of SOP. We then explain the workflow of SOP and how data can be queried using SOP.

The generalized skipping-oriented partitioning (GSOP) framework generalizes SOP by allowing different columns to have different partitioning schemes. We first identify the notion of "feature conflict", which can be a problem in SOP that severely degrades the skipping effectiveness for some workloads. We then list out a spectrum of partitioning layout schemes. While SOP can only cover one end of the spectrum, GSOP incorporates the full spectrum. The trade-off involved in the design of GSOP layout schemes is data scan savings vs. tuple-reconstruction cost. We develop an objective function for GSOP that factors in this trade-off. Built on the SOP workflow, GSOP introduces two new components, namely, column grouping and local feature selection.

The generalized skipping-oriented partitioning with replication (GSOP-R) framework further extends the GSOP framework by exploring how a modest amount of fine-grained data replication can be leveraged to improve the effectiveness of data skipping and query performance. We first define a GSOP-R scheme, which allows for fine-grained and flexible replications. We then develop an objective function to evaluate the goodness of a GSOP-R scheme, which factors in data scan cost, row-reconstruction cost, and storage cost. As the search space for the GSOP-R scheme is huge, we propose to search for GSOP-R schemes through *scheme transformation*. We define and explain 3 types of allowed scheme transformations, each of which has the potential of improving

the GSOP-R scheme. We then go through an iterative search process, which greedily picks the best local transformation and applies it to the current scheme.

1.4.6 System Prototype and Empirical Evaluation

In this dissertation, we describe the system prototype of our proposed techniques built on top of open source systems, such as Apache Spark [44] and Apache Parquet [14]. The prototype includes three components: workload analysis, data re-organization, and skipping-aware query processing. Workload analysis takes as input a query log and generates a set of workload features. The data re-organization component partitions and replicates data as the data is being loaded into the database. For the component of skipping-ware query processing, we simply leverage the built-in data skipping mechanisms that exist in most existing analytics systems. None of these components is tied to a particular system or requires major changes to the code base of existing systems.

To evaluate the performance of our proposed techniques, we use two public benchmarks, namely, TPC-H Benchmark [67] and Big Data Benchmark [18], and two real-world workloads, namely, Conviva [23] and Sloan Digital Sky Survey [63]. We compare the performance of our techniques with the state-of-the-art techniques. Our results show that our techniques can improve the query performance by a factor of 2 to 9 over the state-of-the-art.

1.5 Thesis Organization

The remainder of this dissertation is organized as follows. Chapter 2 is a review of the background for this dissertation. Chapter 3 presents the skipping-oriented partitioning (SOP) framework [65, 64]. Chapter 4 presents the generalized skipping-oriented partitioning (GSOP) framework [66]. Chapter 5 presents the generalized skipping-oriented partitioning with replication (GSOP-R) framework. Chapter 6 presents the conclusions and outlines the future work.

Chapter 2

Background

In this chapter, we provide the background necessary for understanding the techniques proposed in this dissertation. Since our techniques are developed in the context of a relational database management systems (RDBMS), we first give an overview of RDBMS in Section 2.1. We then discuss several storage architectures of RDBMS's in Section 2.2. We review physical design techniques in Section 2.3 and data skipping in Section 2.4. We conclude the chapter in Section 2.5.

2.1 Overview of Relational Database Management Systems

A relational database management system (RDBMS) is a system that lets users create, update, and administer the data based on the relational model invented by Edgar F. Codd [22]. In a relational DBMS, the data is stored as a collection of tables. Each table consists rows and columns. Each *row* corresponds to a record or entity. Each *column* corresponds to a field or attribute for all records in the table. Thus, we also refer to a row as a *tuple*, as it is composed of a set of attribute values. In an RDBMS, we can also specify the relationship between tables. For example, we can define two tables that share one or more common attributes. Users can interact with an RDBMS through many kinds of operations, such as defining and modifying tables, loading data into tables and querying tables. These operations are typically expressed using Structured Query Language, or SQL. We provide an overview of SQL in Section 2.1.2.

Depending on the application scenarios, we classify RDBMS's into two categories: *analytics databases* and *operational databases*. An analytics database is typically designed for supporting business intelligence and data analytics applications, while an operational database is often used for maintaining information regarding day-to-day operations. While both kinds of RDBMS's are based on the same data model, they can involve dramatically different implementation techniques and architectures. In this dissertation, we only focus on analytics databases. In the remainder of this section, we provide an overview of data loading operations and query processing in analytics databases, as these topics are closely related to the techniques developed in this dissertation.

2.1.1 Data Loading

An analytics database typically works in an environment called a *data warehouse*. A data warehouse serves as a central repository of data for an organization, where the data can be collected from various sources. The process of importing data into an analytics database is referred to as an *ETL* process, as it typically follows a Extract-Transform-Load (ETL) pipeline. An ETL process extracts and integrates information from data sources, or the raw data, and then transforms it into the format as defined by the database before loading it into database tables.

We now focus on the data loading process on a single table. In a data warehouse environment, the rows are often *batch-appended* to the table. In other words, there is a background process that accumulates new records and appends them in batches to the table in a pre-defined time interval, e.g., every hour. Once these records are loaded into the table, we can assume that these records rarely, if ever, change. An analytics database physically places the tables in its *storage engine*, which is highly optimized for providing efficient access to the data. In the storage engine, data typically resides on secondary storage that can be much slower to access than memory, e.g., hard disks. Since this dissertation is focused on the physical storage design, we dive into the details of different storage architectures for analytics databases in Section 2.2. We next discuss how these data can be queried.

2.1.2 Query Processing

In an RDBMS, users can query the data using SQL[1]. SQL is a declarative language, which describes the desired result without specifying the procedure of obtaining that result. This not only simplifies the user tasks but also separates the logical layer from the physical layer, which makes databases more extensible and provides tremendous opportunities for under-the-hood optimization. We now show a simplified SQL query structure:

SELECT *list of column names*
FROM *list of table names*
WHERE *filter predicates*
GROUP BY *list of column names*
ORDER BY *list of column names*

In a SQL query, the FROM clause first specifies the tables with which the query is concerned. When there is a single table in the FROM clause, the query result will contain a subset of rows and columns from this table. When there are multiple tables in the FROM clause, the row set of the query result will be a subset of the cartesian product of the rows from these tables, and the column set of the query result will be a subset of the union of columns from these tables. Which subsets of columns and rows appear in the query results is determined by the SELECT and WHERE clause, respectively. Specifically, the SELECT clause directly lists the subset of columns, while the WHERE clause uses *filter predicates* to qualify a subset of rows for the query result. In practice,

[1]For more details on SQL and RDBMS's, please refer to [53].

when the query involves multiple tables (in the FROM clause), there are usually predicates in the WHERE clause that specify how to *join* these tables together rather than performing a full cartesian product. After the query result set is determined, the GROUP BY and ORDER BY clauses describe how the query result should be grouped and ordered, respectively.

This simple example of SQL query structure omits many of the SQL semantics, but is good enough to help us understand how queries can be processed in an RDBMS. When a user poses such a SQL query, the *query compiler* first parses the query string and generates a logical representation of the query. Then, the *query optimizer* picks a *physical execution plan* for the query, which provides detailed steps on how the query results can be obtained. Since SQL is a declarative language, a query can have multiple physical execution plans. In fact, different physical execution plans can incur dramatically different costs. We defer the discussion of query costs to Section 2.1.3. The goal of a query optimizer is to select the best physical execution plan, i.e., the plan that incurs the smallest cost. An RDBMS can employ either a *rule-based* or a *cost-based* query optimizer. A rule-based query optimizer follows a set of heuristics rules on how to translate a logical query plan to physical plan, while a cost-based query optimizer relies on data statistics to estimate the cost of each physical execution plan and then picks one with the smallest estimated cost. After the optimizer picks the physical execution plan, the query engine simply follows the steps of the physical plan to execute the query and returns the query result to the user. We now discuss several important operations involved in the query execution.

- **Scan.** The scan operation is typically the first operation of the query execution pipeline. It accesses the table either directly or through an index (details in Section 2.3). Since a query is only concerned with a subset of rows and columns, it is the scan operation's job to apply filters on columns (SELECT clause) as well as on rows (WHERE clause) and eliminate the parts of the table that are irrelevant to the query. Thus, the scan operation can easily become the bottleneck of the entire query execution, because the other operations only consume the data produced by the scan operation, which may be only a small fraction of the entire table.

- **Join.** When the query involves multiple tables, the query engine must join these tables together. A join operation can be performed on the output of the scan operations performed on every table involved in the join. On the other hand, the join condition can also be "pushed down" to the scan operation so that the scan and join operations are effectively performed at the same time. While there are multiple ways of performing a join, the query execution engine simply follows the join method specified by the physical execution plan. The physical execution plan also describes the orders of joins when there are more than two tables involved.

- **Group.** The group operation is frequently used in analytics queries. This operation often consumes the data produced by scan or join operations and groups these rows based on the columns specified in the GROUP BY clause.

- **Aggregate**. Although not shown in the example SQL structure above, aggregations, such as sum, min, average, are commonly used in analytics queries. Aggregations are typically performed as a final step of the query processing pipeline. However, they can also be combined with other operations, such as group. Some advanced RDBMS's can even perform aggregations early in the plan as part of scan operations.

2.1.3 Performance Metrics

Generally, analytics databases are evaluated using two main performance metrics: *query response time* and *resource consumption*. Query response time measures the time interval from the issuing time of the query to its completion time. The time of query execution can be spent on CPU and I/O. Query response time is an important metric for interactive analysis applications, where database users usually expect to see the query result in near real-time. Resource consumption, on the other hand, measures how many resources are consumed for evaluating a query. Modern analytics database are distributed, where many computers can work in parallel for executing a single query. Thus, we need a measurement on how much cumulative CPU and I/O time is spent on evaluating a query. As the hardware resources within an organization are limited, reducing the resource consumption can help the organization keep the cost of running the data infrastructure under control. As discussed in Chapter 1, the objective of our proposed techniques is to reduce the amount of data scanned. Thus, our techniques can improve both performance metrics. In the performance evaluations throughout this dissertation, we focus on query response time, as it is more commonly used in performance benchmarks.

2.2 Storage Architecture of Analytics Databases

As opposed to the transactional workloads that arise in operational databases, which touch relatively small amounts of data, analytics workloads typically involve accessing a large subset of the data in order to derive aggregated insights. Thus, the design of analytics databases is focused on processing large-scale scans and aggregations in an efficient and scalable manner.

2.2.1 Row Stores

Despite the dramatic difference in target workloads and design goals between analytics databases and operational databases, in the early days, traditional database vendors adopted similar technology and architectures for both kinds of systems. We categorize such traditional analytics databases as *row stores*.

In a row store architecture, the data is physically stored as a set of rows (or records, tuples), and all fields of a row are stored contiguously. Figure 2.1(a) shows a table in a row store architecture. A row store architecture not only aligns with the logical view of a database table, which is a set of rows, but also supports many database operations naturally. First, a row-store can easily handle the insertion of new database records, as it can append these incoming records as new rows to

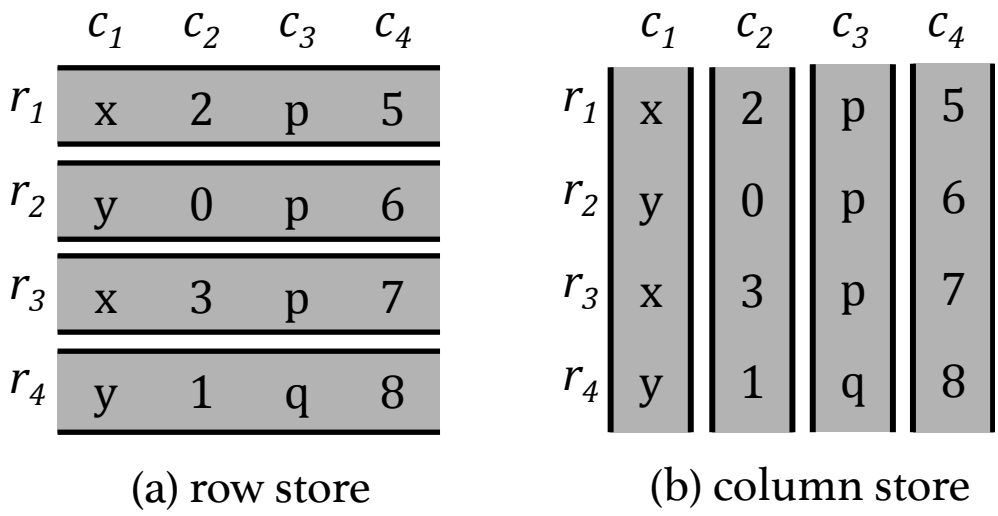


Figure 2.1: Row Store vs. Column Store

the table without affecting existing rows. Second, a typical query execution engine processes a row at a time, and thus can directly read data from a row store without additional conversion steps. Third, many update operations are row-based, e.g., updating a particular record or a set of records that satisfy certain predicates. Such operations can be efficiently performed in a row store, since all attributes of a record can be found in the same place.

While a row store is good for many operations mentioned above, it also exhibits drawbacks for some access patterns that are commonly seen in analytics workloads. One of the most notable observations about analytics workloads is that many queries only access a small subset of attributes. In a row store, however, queries have to read all attributes of the table from secondary storage because all attributes of a record are stored together and it is difficult to separately read the requested attributes without touching the other attributes. This incurs a lot of unnecessary data scans that do not contribute to the final query results, which leads to degraded query efficiency.

2.2.2 Column Stores

Motivated by the properties of analytics workloads, analytics systems have increasingly adopted a column-oriented architecture, which stores each column individually. Figure 2.1(b) shows a table in a column store architecture. Such an architecture may look counter-intuitive at first, as it complicates a lot of database operations that are row-oriented. For example, when inserting a new record to a table, a column store has to append each attribute value of this new record to every column individually. Since query execution engines are row-based, column stores have to reconstruct the data columns read back into rows before the data are fed into the subsequent stages of query processing.

Despite these apparent overheads, column stores can have tremendous performance benefits

for analytics workloads. First of all, column stores can make queries scan much fewer columns than row stores. As columns are separately stored, queries can scan just the columns requested by queries and avoid accessing irrelevant columns. Moreover, the unique properties of column stores enable many effective optimizations on data scans. Leveraging data compression is one of the most important optimizations to reduce data scans in column stores. Since all values of a column are of the same type, there are opportunities to adopt column-specific compression techniques. For example, null compression is effective for sparse columns. Column stores also allow different columns to be sorted in different orders for applying even more aggressive compression schemes, such as delta encoding and run-length encoding [25]. When the columns are in different orders, however, column stores need to maintain a set of row ids for each column in order to reconstruct them back into rows. Due to the greatly reduced size of the compressed data, the I/O cost of reading such data can be dramatically improved. To process the compressed data, the query engine usually needs to perform decompression first. The CPU overhead of decompression, however, can be easily justified by the I/O improvement. Advanced column-store query engines can also support processing the compressed data directly for certain types of queries [25].

As mentioned earlier, the disadvantage of column stores is on writes. For analytics workloads, however, this is not a huge problem. In many analytics scenarios, the workload conforms to a"write-once, read-many" model, where data is batch loaded into the database and does not change thereafter. For such a model, column stores only have to pay the loading cost of separating the attribute values of the new records into individual columns and compressing these columns. Many column-specific compression techniques, e.g., run-length encoding, support data compression on-the-fly, i.e., they allow new values to be added to a column without the need of decompressing existing values. In general, column stores have been considered suitable for analytics workloads and many analytics systems have adopted a column store architecture.

2.2.3 Hadoop Storage

In recent years, Hadoop [32] and Spark [44] have gained popularity for large-scale analytics. Many execution engines and storage systems have been built as part of the Hadoop ecosystem. The storage layer for the Hadoop ecosystem is centered around the Hadoop File System (HDFS). On HDFS, analytics file formats in Hadoop, such as ORC [73] and Parquet [14], have borrowed ideas from column stores and adopted a column-oriented data layout for efficient data retrieval. However, the architecture of these Hadoop file formats is different from a pure column store. In HDFS, data are stored as HDFS blocks, each of which is a horizontal partition of a file. Each HDFS block is typically replicated 3 ways and can reside in any machine of a HDFS cluster. The analytics file formats ORC and Parquet adopt a PAX-style [1] architecture, which takes advantage of columnar layouts and at the same time conforms to the HDFS block-based architecture. In these file formats, the data is first horizontally partitioned, so that each horizontal partition fits in a HDFS block. Within each HDFS block, the data uses a columnar layout. When a query reads data from a HDFS block, it enjoys the benefit of a column store. Another major benefit of this architecture is that all attribute values of a row are guaranteed to reside in the same HDFS block and thereby in the same machine. Thus, row-reconstruction can be done locally without the

need of assembling columns from multiple machines. In this dissertation, the prototypes of our proposed techniques are implemented on these Hadoop file formats.

2.3 Physical Database Design

Physical database design is the process of determining data layouts and auxiliary data structures for the database. The choice of physical database design has a great impact on the cost of data scan and thus the overall query performance. In this section, we survey several important physical database design techniques.

2.3.1 Indexing

Indexing is one of the oldest and most important physical database design techniques. Indexing is a broad term in the context of databases and can refer to creating and maintaining any auxiliary data structure that helps queries navigate the data. In this chapter, we focus the discussion on the traditional indexes such as B+-trees.

With the help of indexes, queries can quickly locate and retrieve certain records without the need of scanning the entire table. However, indexes are not free. First, an index can incur significant *storage cost*. Second, when there is an update to the data, the indexes built on top of these data also have to be updated accordingly. Thus, the *maintenance cost* has to be taken into account when picking indexes. The goal of picking indexes in physical database design is to pick a set of indexes that minimize the cost of a target workload, including query cost and maintenance cost, under a storage cost budget.

An index simply provides a way to access the data. Thus, a query can pick an index to access the requested data in the table, which we refer to as an *index scan*, or perform a full scan directly on the data. Note that an index scan is not necessarily cheaper than a full scan, even when the index scan accesses less data. This is because, when the data resides in hard-drive disks, an index scan may incur *random disk accesses*, while a full scan only involves *sequential disk accesses*. Random disk access deliver a lower rate of throughput due to the overhead of disk seeks. We categorize indexes as two types: primary and secondary. A primary index determines the physical order of the underlying data, while a secondary index can be in a different order from the actual data. Thus, a table can have only one primary index and multiple secondary indexes. When using a secondary index, a query may need to go to multiple places to retrieve the requested data, because the entries of the secondary index are in a different order from the actual data. This incurs random disk accesses when the data resides in hard disks. In this case, an index scan is only considered a preferred choice when it accesses much less data than full scan, e.g., less than 5% of the table. Thus, in some cases, a full scan is the best access path even with the presence of indexes. Given a query, it is the query optimizer's job to pick the best access path, i.e., the access path with the smallest estimated cost, which can be an index scan using one index or multiple indexes combined, or simply a full scan.

The above discussion is focused on traditional indexes on row-stores, such as B+-trees. These indexes work best for the workloads that need to access only a few rows, such as transactional workloads. For analytics systems, however, the use of such indexing techniques has become rather limited. Modern analytics systems mostly adopt a column-oriented architecture, where full scans can be fast due to the column-store techniques such as compression. With the help of techniques like data skipping, the performance of full scans can be even improved due to the effective reduction of unnecessary scans.

2.3.2 Horizontal Partitioning

Horizontal partitioning refers to the process of dividing a table into a set of horizontal partitions. Each horizontal partition contains a subset of rows of the table and is separately stored as a file or a directory.

Horizontal partitioning is useful in facilitating database management operations. In a data warehouse environment, we need to constantly insert new data to the table and delete old data from the table. If the table is not partitioned, we need to append the new rows to the table and explicitly find the rows that need to be deleted. If we horizontally partition the table based on time ranges, e.g., each partition contains a day's worth of data, we can easily add a new date partition to the table when the new data comes in. To delete old data, we can simply remove the oldest date partition from the table. In both cases, other partitions of the table are unaffected and can remain online to serve queries. These operations are referred to as "roll-in" and "roll-out" operations, which are commonly seen in data warehouses.

Horizontal partitioning also plays an important role in improving query performance. Given a large table, we can perform a horizontal partitioning on the values of one or more columns such that the values of the partitioning column(s) in each partition is bounded by a range. This is a classic horizontal partitioning technique called range partitioning. When a query comes, it can use the value ranges attached to each partition as metadata and prune the partitions whose value ranges fall out of the query predicate range. This way the query can avoid scanning the entire table.

Traditionally, horizontal partitioning has been applied using relatively coarse granularity. Each horizontal partition is typically at file level or directory level. Indexes can be built within each horizontal partition. In modern analytics systems, however, horizontal partitioning can be applied on a much finer granularity for further improvement on query performance. A horizontal partition can contain as few as 1,000's of rows. In this dissertation, we refer to such fine-grained horizontal partitions as blocks. By storing some metadata on each block, a query can skip blocks based on these metadata.

2.3.3 Vertical Partitioning

In contrast to horizontal partitioning, vertical partitioning refers to the process of dividing the table into a set of column groups. Each vertical partition only contains a subset of columns of the table.

Vertical partitioning in a row store can reduce the number of columns accessed by a query. As mentioned in Section 2.2.1, in a row store, a query may need to access all columns of the table even when it only requests a small subset of them. When the table is vertically partitioned, if all the columns requested by a query reside in the same vertical partition, then the query only needs to access this vertical partition and can avoid accessing other columns. On the other hand, when a query needs to read columns from multiple vertical partitions, it needs to pay the extra cost of assembling these columns, as different attribute values of a row may not be stored contiguously. Vertical partitioning in a row store can be viewed as middle point between a column store and a row store. The more vertical partitions a row store has, the more it looks like a column store. In the design of a vertical partitioning scheme for a row store, the goal is to balance the trade-off between the number of columns accessed and the row-reconstruction overhead for a target workload.

Vertical partitioning for a column store is usually referred to as column grouping. Since columns are already separately stored in a column store, vertical partitioning plays a different role in column store than in a row store. Although each column is stored separately in a column store, these columns are aligned. Creating vertical partitions, or column groups, in a column store allows each column group to have its own order. As discussed in Section 2.2.2, data compression is an essential element of column stores. The effectiveness of several compression algorithm highly depends on the ordering of the values. Column grouping not only makes each column group compress better, but also leverages group-specific ordering to improve data scans. However, the downside is that, when a query has to retrieve columns from multiple column groups, it has to sort these columns before they can be assembled as they are not aligned. Given a target workload, the goal of vertical partitioning in a column store is to find a good trade-off between the performance of column scan and the row-reconstruction overhead.

2.3.4 Materialized Views

Another important physical database design option is the creation of materialized views. Materialized views are pre-computed query results. Creating materialized views refers to the process of pre-evaluating queries offline and storing the results separately from the original data. The use of materialized views trades space and offline preparation time for online query processing time. Choosing what queries to materialize can be determined by analyzing the workloads or studying the query templates. Given a table and a set of materialized views, an incoming query can access an applicable materialized view where it can obtain results faster than evaluating it on the original data from scratch.

Materialized views do not have to be fully evaluated query results. A materialized view can also be a partially evaluated result generated by any operation of a query execution pipeline. For example, we can create a materialized view by selecting a set of rows out of the table, applying a group-by, or joining multiple tables. As different queries may share some intermediate results, we can choose to materialize intermediate results so that it can benefit multiple queries. Typically, a more general materialized view can benefit more queries, but provides limited improvement on each individual query it benefits. This is because a materialized view that prioritizes generality

only offers partially-evaluated results, and to use such a materialized view, queries have to go through further computations. On the other hand, a more specific materialized view is targeting a smaller set of queries but can provide significant savings. Materialized views generated by aggregations typically belong to this category. Picking the most cost-effective intermediate results to materialize can be done in a cost-based manner.

Materialize views can be considered as a form of data replication, as they replicate the data that are part of query results. In the next section, we discuss another form of replication: full-copy replication.

2.3.5 Full-copy Replication

In full-copy replication schemes, we simply make several copies of the entire data. Full-copy replication is critical for the purposes of availability, fault-tolerance, and reliability. When a machine that contains some data crashes, for example, some other copy of the same data from other machines can take over. Full-copy replications can increase locality. In a distributed setting, if a data copy happens to reside in the same machine as the processor, the query can read the local copy instead of accessing the copy remotely, which can save some network transmission delays.

There has been significant research [54, 4] on leveraging the full copy replication for heterogeneous physical database design. The central idea is to have a different physical design choice on a different replica so that each replica is best suited to serve a different class of workload. Suppose a table has two columns A and B. If we are to build a primary index for the table, we can only build it on either column A or column B. As discussed earlier, a primary index can bring great performance benefits, but each table can only have at most one primary index, as it determines the order of the underlying table. If the table has a 2-way replication, we can build a primary index on column A in replica 1 and a primary index on column B in replica 2. This way, the table can take advantage of both primary indexes for efficient query processing.

2.4 Data Skipping

As discussed in Chapter 1, while partition pruning has been a classic technique in commercial systems (e.g., Oracle [49] and Postgres [50]), data skipping was first proposed in [48]. Specifically, they proposed to maintain small materialized aggregates (SMAs) for each range-partitioned block, such as min, max, count, sum and histograms for each column. Recently, most analytics systems [34, 27, 5, 3, 70, 68, 2, 16] have adopted this idea. As opposed to partitioning pruning, data skipping maintains metadata for all columns, not just the date column, for data *blocks*. Each block is a small horizontal partition of the table, which typically contains 1,000 to 10,000 rows. Data skipping associates each block with some metadata such as min and max values for all columns. Before scanning a block, the query first evaluates its filter against this metadata and then decides if a block can be skipped, i.e., the block does not need to be accessed.

To partition the data into these fine-grained blocks, existing systems have used range partitioning (e.g., [2]), simple splitting (e.g., [14]) or sorting-then-splitting (e.g., [5]). We note that these

simple physical design approaches do not realize the full potential of data skipping. For example, range partitioning is too coarse-grained to generate these small blocks; simple splitting does not re-organizes the data at all, which may miss the opportunity of trading the offline cost for online query performance. Moreover, these systems only consider horizontal partitioning techniques for data skipping and do not incorporate any of the other physical design techniques discussed in Section 2.3. To this end, the focus of this dissertation is to leverage innovative physical design techniques that can fully realize the potential of data skipping. We provide more detailed discussion on specific prior work of data skipping in the following chapters.

2.5 Conclusion

In this chapter, we provide an overview of RDBMS. We also discuss physical design techniques in RDBMS's. We also discuss the pros and cons of row-oriented and column-oriented data layouts. In Chapter 3, we propose innovative horizontal partitioning techniques that can work for row- and column-oriented data layouts. Since analytics systems mostly adopt column-oriented layouts, our proposed techniques in Chapter 4 and Chapter 5 are specifically designed for column-oriented layouts, which are inspired by vertical partitioning, materialized views and full-copy replication.

Chapter 3

Skipping-Oriented Partitioning

3.1 Introduction

In this section, we introduce the skipping-oriented partitioning (SOP) framework. We first explain the design goals of the framework, and then illustrate how such a framework works using an example. Finally, we point out the challenges and our contributions involved in designing the SOP framework.

3.1.1 Overview

The effectiveness of data skipping depends on how the data is partitioned into blocks. Current systems adopt small block sizes for data skipping. For example, IBM DB2 BLU [68] uses $1,000$-tuple blocks; Google's PowerDrill [3] suggests $50,000$ tuples; Shark [70] skips data at the granularity of HDFS blocks, each of which is 128MB by default. These systems rely on range partitioning to generate such blocks. While range partitioning has been useful for many purposes, it may not be ideal for generating fine-grained blocks for skipping. Specifically, range partitioning lacks of a principled way of: (1) setting the fine-grained ranges on each column that matches the data skew and workload skew, (2) allocating the number of partitions for different columns and (3) capturing inter-column data correlation and filter correlation.

In this chapter, we propose a skipping-oriented partitioning (SOP) technique, with a goal of (horizontally) partitioning the data into *fine-grained, balance-sized* blocks in a way that queries can skip a lot of blocks. SOP is an offline process that executes at data loading time. Note that the SOP technique can co-exist with traditional horizontal partitioning techniques, as these techniques may be used for a different purpose, such as roll-in/roll-out operations. Specifically, SOP can be applied to further segment each individual partition.

In SOP, we first extract some filter predicates as features from a past query log using frequent itemset mining [8]. We then generate feature vectors by precomputing these filter predicates on the data and solve an optimization problem to guide the data partitioning.As we describe in Section 3.2, in many real-world workloads, especially the reporting and scheduled workloads, similar queries are repeatedly run when new data comes in. We analyze real-world workloads

	time	id	event	category	publisher	revenue
t_1	08:01:01	102	click	jeans	groupon	0.0
t_2	08:01:01	103	click	shirts	google	-0.5
t_3	08:01:01	104	click	shirts	groupon	0.0
t_4	08:01:02	105	buy	jeans	google	12.0
t_5	08:01:03	106	click	jeans	google	-0.5
t_6	08:01:04	107	buy	shoes	shoedeal	30.0

(a) tuples

	features	weight
F_1	*event='buy'*	50
F_2	*product='jeans'*	20
F_3	*publisher='google'* *revenue < 0*	10

(b) features

	vector (F_1, F_2, F_3)
t_1	(0,1,0)
t_2	(0,0,1)
t_3	(0,0,0)
t_4	(1,1,0)
t_5	(0,1,1)
t_6	(1,0,0)

(c) vectors

	blocking
P_1 (1,1,0)	t_1 (0,1,0) t_4 (1,1,0)
P_2 (0,1,1)	t_2 (0,0,1) t_5 (0,1,1)
P_3 (1,0,0)	t_3 (0,0,0) t_6 (1,0,0)

(d) blocks

Figure 3.1: Example of Skipping-oriented Partitioning

in Section 3.2 and show that (1) a small set of representative filters are commonly used by many queries and (2) many queries use recurring filters. These findings suggest that the workload-driven approach in SOP can be effective for real query workloads.

Some earlier approaches also utilize workloads for physical database design, e.g., [24, 35, 11, 56]. Specifically, our approach is related to materialized view selection (MVS) [17, 56]. Like MVS, we exploit precomputation. However, SOP works at a finer-granularity and is complementary to materialized views. In fact, SOP can be applied to partition large materialized views, e.g., data cubes. As we will show shortly, SOP maintains concise feature-based metadata derived from precomputation. Another proposed data skipping technique involves the use of small materialized aggregates (SMAs) associated with partitions [48, 27]. These SMAs have been shown to improve query performance in range-partitioned systems. In contrast, this chapter is focused on constructing fine-grained partitions that more closely capture the access patterns of complex analytics workloads. Like materialized views, SMAs are also complementary to this work and in fact could be implemented on the SOP partitions as well. We defer the detailed discussion of related work to Section 3.8.

3.1.2 An SOP Example

Suppose we are given a table as shown in Figure 3.1(a), an example log of online events. We first look at the log of queries that were posed on this table and extract a set of features, each of which is a representative filter with possibly multiple conjunctive predicates. Suppose the features extracted are as shown in Figure 3.1(b). Given these features, we then transform the data tuples into feature vectors. This process can be done by scanning the table once and batch-evaluating the features on each tuple. As shown in Figure 3.1(c), each feature vector is (in this case) a 3-dimensional bit vector, whose i-th bit indicates whether this tuple satisfies filter F_i. In practice, the number of features can be kept small, e.g., < 50. We then partition the tuples according to these vectors. Intuitively, tuples that do not satisfy the same features should be placed in the same block such that, when a query uses one of these features as filter, this block of tuples can be skipped altogether. An example of the resulting data blocks is shown in Figure 3.1(d). For each block, we compute a *union vector* by taking a bitwise OR of all the feature vectors in it.

If the i-th bit of the union vector is 0, then we know that no tuple in this block satisfies feature i. In this case, any query whose filter is F_i can skip this block. For example, a query on F_3 can skip the blocks P_1 and P_3. More generally, a query may be able to skip blocks if its filter *is subsumed by* (i.e., is stricter than or equal to) some features. For example, a query with filter *event = 'buy'* $\wedge$ *product = 'jeans'* is subsumed by both features F_1 and F_2, which lead to the skipping of P_2 and P_3 respectively.

3.1.3 Contributions

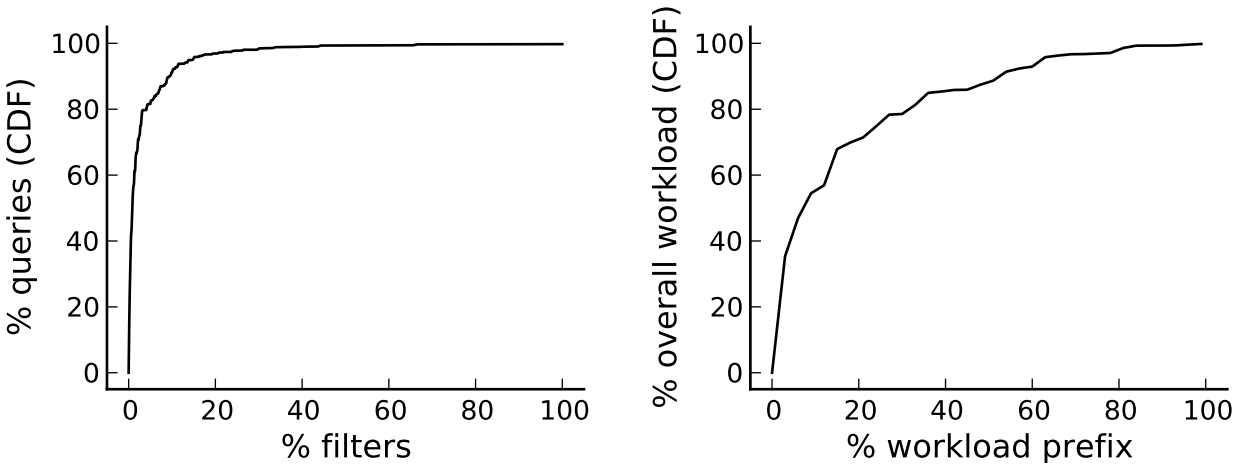
To realize the design of SOP, we address a few technical challenges as outlined below.

Feature Selection. Indeed, selecting the right features to guide partitioning is critical. We develop a *workload analyzer* to identify representative filters as features from a query log. We consider a feature representative if it could be used to help many queries. If some filter predicates are frequently used together, we should combine these predicates as one feature to skip data more effectively, e.g., feature F_3 in Figure 3.1. To capture both *frequency* and *co-occurrence* of filters, we model the feature selection as a frequent itemset mining problem. Due to their subsumption relations, some features can be redundant. For example, $revenue > 0$ could be redundant if there is already a feature $revenue > 100$. We develop a principled way to eliminate redundancy.

Optimal Partitioning. Given a set of features, we compute a feature bit-vector for each tuple. The problem then is to find an optimal partitioning over these vectors. This is clearly a hard problem, as different features may be conflicting, may be correlated, and may have different selectivities. We formulate the *Balanced MaxSkip* partitioning problem: given a desired number of tuples per block, find a partitioning over a collection of tuples (represented as bit vectors) that maximizes the number of tuples that can be skipped. This objective is fundamentally different from other well-known partitioning objectives, such as k-means and distance-based clustering [52]. We prove that *Balanced MaxSkip* is NP-hard, by a reduction from the hypergraph bisection problem [40]. We conjecture that *k-MaxSkip*, a variant without the balance constraint, is also NP-hard. To find an approximate solution efficiently, we adopt the classic bottom-up clustering framework, as it naturally incorporates the objective function of SOP and is a widely-understood framework with scalable implementations, e.g., [75, 31].

Partitioning Cost. SOP adopts a bottom-up clustering algorithm [52] to generate the partitioning map. It is prohibitively expensive to run this algorithm on large datasets. Fortunately, we observe that the input size can be reduced from the number of tuples to the number of distinct feature vectors. The latter mostly depends on the number of features and can be small (e.g., $<10k$) in practice. As shown in Section 3.7, although we run a sophisticated clustering algorithm on the vectors, the actual data movement still takes up most of the time for the entire process of SOP.

We prototype SOP on Shark [70], an open-source data warehouse system. We conduct experiments using TPC-H benchmark and a real-world ad-hoc workload from a video streaming company. The results show that SOP reduces the data access by a factor of 4-7 over existing skipping techniques on top of range partitioning. We also demonstrate that this reduction can directly translate to a reduction in query response time, on both disk and memory resident data.



(a) Filter Commonality (b) Filter Stability

Figure 3.2: Filter Analysis on Real Ad-hoc Workloads

Specifically, we improve the query response time by 5-14x over full table scans without skipping and by 2-5x over existing skipping techniques.

The remainder of this chapter is organized as follows. Section 3.2 gives an overview of the skipping-oriented partitioning framework. Section 3.3 presents the workload analyzer. We discuss the partitioner in Section 3.4. We show how skipping works during query execution in Section 3.5. Section 3.6 discusses the practical issues. We report the experimental results in Section 3.7. Section 3.8 reviews the related work and Section 3.9 concludes the chapter.

3.2 Framework Overview

In this section, we give an overview of the skipping-oriented parititioning (SOP) framework. We first discuss our workload assumptions and then explain the workflow of SOP.

3.2.1 Workload Assumptions

As stated in Section 3.1, SOP extracts representative features from the workload to guide the partitioning. In order for this approach to work well, it requires two properties of the workload:

Filter Commonality. We expect that there is a small set of filters that are commonly used by many queries. If, on the other hand, each query uses a distinct filter, then it would be difficult to find representative filters.

Filter Stability. Since SOP base its partitioning decisions on a past workload, it is important that most of the filters in future queries have occurred before. In other words, we expect that most of filters are *recurring* and only a small portion are entirely new over time.

The commonality and stability of filters can be observed in recurring scheduled or reporting queries, because such queries are usually generated from templates and the same set of filters

would be repeatedly used on the data of different time ranges. Also, the effectiveness of using past queries to guide database design was also evidenced in many previous works, e.g., [56, 35, 24, 11].

We conducted empirical analysis on a real-world production SQL workload from a video streaming company called Conviva. These 8664 ad-hoc queries, spanning the period from 06/2010 to 01/2012, were used for problem diagnosis and data analytics over an access log of video streams. Note that each query uses possibly multiple filter predicates. For each predicate, e.g., *event = 'click'*, we count its frequency, i.e., how many queries use it. In Figure 3.2(a), we show the filters in the descending order of frequency and plot the cumulative percentage of the queries that use the filters. A point (x, y) indicates that the most frequent $x\%$ filters are used by $y\%$ of queries. We can see that the filter usage is highly skewed, e.g., 10% of the unique filters were used by 90% of the queries. This implies that using only 10% of filters as features can benefit 90% of the queries.

We then examine the queries in the order that they arrive. To prevent our analysis from being biased towards a particular starting point, we divided the 8664 queries into 5 disjoint time windows and plotted five curves. Figure 3.2(b) shows an average over these five curves. A point (x, y) means that, as we have seen $x\%$ of the queries (or $x\%$ workload prefix), the filters in these $x\%$ are used by $y\%$ of all the queries in the workload. If every query used completely new filters, i.e., there is no recurring filter, this curve would be a function of $y = x$. The plot, however, shows that many queries use recurring filters. In particular, 80% of the entire workload uses the filters that already occur in the 30% prefix. Since the filters are recurring, we can infer that most of the future filters are predictable based on a past workload.

3.2.2 The SOP Workflow

Having described the workload properties on which SOP depends, we now overview the SOP workflow, as depicted in Figure 3.3. This workflow consists of three standard data marshaling steps (shaded arrows) and two important modules named the workload analyzer and the partitioner.

The input is a table and a workload represented as a collection of queries on this table. The query workload can be obtained by collecting query logs of the system. We now walk through the individual steps in the workflow:

(1) Workload Analyzer. The workload analyzer (Section 3.3) extracts a set of features from the query workload. A feature is a representative filter with possibly multiple predicates.

(2) Featurization. Given the features, i.e., filters, from Step 1, we scan the table once and batch evaluate these filters for each tuple. This step transforms each tuple to a (vector, tuple)-pair.

(3) Reduction. Since the partitioning only depends on the feature vectors, not the actual tuples, we group the (vector, tuple)-pairs into (vector, count)-pairs. This is an important step to reduce the input size for the partitioner.

(4) Partitioner. The partitioner (Section 3.4) runs a partitioning algorithm on the (vector, count)-pairs. This generates a *partitioning map* (from a feature vector to a block id).

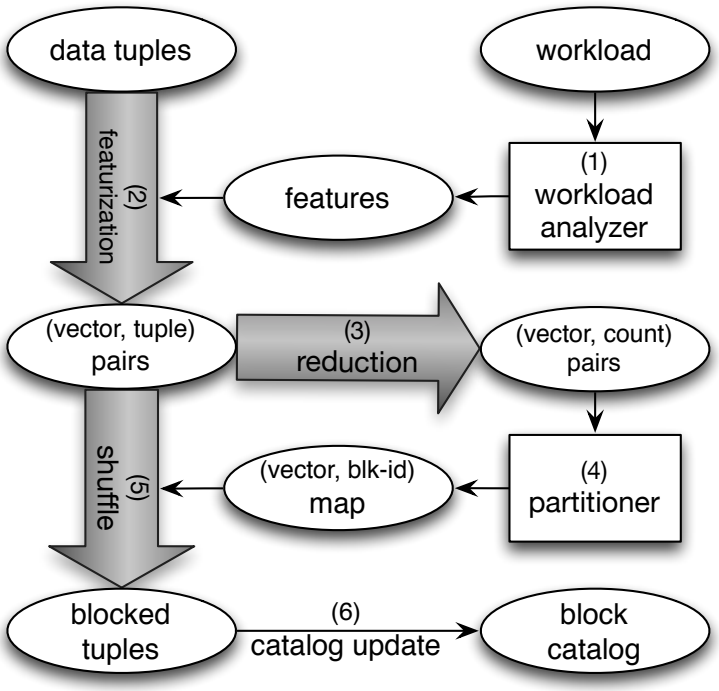


Figure 3.3: The SOP Workflow

(5) Shuffle. In this step, each of the augmented tuples (output of Step 3) finds its destination block by consulting the partitioning map (output of Step 4).

(6) Catalog Update. We add the features and one union vector (e.g., Figure 3.1(d)) for each block to the *block catalog* (Section 3.5). After this step, we can drop the feature vectors, and the partitioned tuples are ready to serve queries.

The above workflow can be executed as an offline process at data loading time and may be re-executed later to account for workload changes. In the event that the data arrival rate is high or that the new data needs to be queried immediately, the partitioning process can be postponed. As stated in Section 3.1, in many data warehouse applications, tables are partitioned by time ranges, and new tuples are typically batch-inserted as a new partition. We can consider the SOP partitioning as a "secondary" partitioning scheme under each such coarse-grained partition. For example, when a new partition `dt='1994-11-03'` is added to the table `logs`:

```
ALTER TABLE logs ADD PARTITION (dt='1994-11-03') ...
```

we can apply SOP on this newly inserted partition without affecting existing data.

Having outlined the workflow, in the following sections, we will discuss the different components in detail.

3.3 Workload Analysis

The goal of workload analysis is to extract features from the query trace. We follow two intuitions. First, we should use representative filters as features to guide the blocking. A feature is representative if it can potentially help a large number of queries skip data. Second, the filter predicates that are frequently used together should be considered as one single feature, e.g., F_3 in Figure 3.1, as these predicates will likely be used together in the future queries and combining them can greatly maximize block skipping. To take into account the counting and co-occurrence of predicates, we model the workload analysis as a *frequent itemset mining* problem [37]. In this section, we first formulate the problem and then present a principled approach for feature selection.

3.3.1 Workload Model

A workload is a collection of queries. Each query is associated with a filter, which evaluates a data tuple and returns a boolean value. Without loss of generality, we assume each query uses a conjunction of *predicates*, where a predicate is a disjunction of filter literals. A filter literal can be an equality or range condition, a string matching operation, or a boolean user defined function. Since we are only concerned with the filter part of the queries, we represent the workload by $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_m\}$, where each Q_i is a set of (conjunctive) predicates. An example workload is given in Example 1.

Example 1 *An example workload, each of which is represented as a set of conjunctive predicates:*
Q_1*:* $product$ = '$shoes$'
Q_2*:* $product$ *in* ('$shoes$', '$shirts$'), $revenue > 32$
Q_3*:* $product$ = '$shirts$' $\wedge$ $revenue > 21$

The workload is thus modeled as a transactional database [8], where a predicate is an item and a query is a transaction. For example, Q_1 can be viewed as a transaction of only one item *product='shoes'*. Let F be the set of all predicates that occurred in $\mathcal{Q}$. We call any set of conjunctive predicates $F_i \subseteq F$ a *predicate set* (analogous to *itemset*). By definition, each Q_i is a predicate set.

Two predicates are equal if all their components are equal, including the columns, the constants and the filtering condition. Given two predicates f_i and f_j, we use $f_i \sqsubseteq f_j$ to denote that f_i *is subsumed by* f_j or f_j *subsumes* f_i, i.e., f_i is equal to or stricter than f_j. For example, the predicate $product =$ '$shoes$' is subsumed by $product$ in ('$shoes$', '$shirts$'). Given two predicate sets F_i and F_j, we say F_i *is subsumed by* F_j, or F_j subsumes F_i, denoted by $F_i \sqsubseteq F_j$, if for any $f_u \in F_j$, there exists f_v in F_i such that $f_v \sqsubseteq f_u$. Clearly, $F_i \sqsubseteq F_j$ if $F_j \subseteq F_i$.

3.3.2 Predicate Augmentation

We want to select representative predicate sets as features. As discussed in Section 3.1 and Section 3.5, a feature can be used to skip data blocks, but only for the queries it subsumes. Let us show this using Example 1. Suppose we use {$product$ in ('$shoes$', '$shirts$')} as a feature. We can

see that this feature subsumes both Q_1 and Q_2. If there is a block that does not contain any tuple that satisfies {$product$ in ('$shoes$', '$shirts$')} (indicated by the union vector, as discussed), we can then infer that no tuple in this block satisfies Q_1 or Q_2, because these queries are subsumed by (i.e., even stricter than) this feature. In this case, Q_1 and Q_2 can safely skip this block.

Since a predicate set, if selected as a feature, is only helpful to the queries it subsumes, we should select the predicate sets that subsume a lot of queries. This problem can be modeled as frequent itemset mining. There is a difference, however, between the number of queries a predicate set subsumes and the number of occurrences this predicate set has. Directly applying a frequent itemset mining algorithm on the queries would miss important features. For example, {$revenue$ > 21} subsumes both Q_2 and Q_3, and {$product$ in ('$shoes$','$shirts$')} subsumes both Q_1 and Q_2; each of these two predicate sets has only 1 occurrence but subsumes 2 queries. To adjust this difference, we perform a filter augmentation step as follows. For each query $Q_i \in \mathcal{Q}$, we augment Q_i with all the predicates in F that subsumes Q_i, using the following procedure.

---

for each $Q_i \in \mathcal{Q}$*:*
  for each $f_j \in F$*:*
    if $\exists f_k \in Q_i$ *s.t.* $f_k \sqsubseteq f_j$*:*
      $Q_i \leftarrow Q_i \cup \{f_j\}$

After this augmentation step, the number of occurrences of a predicate set equals to the number of queries it subsumes. For example, the workload in Example 3 becomes:

Q_1: $prod.$='$shoes$', $prod.$ in ('$shoes$', '$shirts$')
Q_2: $prod.$ in ('$shoes$', '$shirts$'), $revenue$>32, $revenue$> 21
Q_3: $prod.$='$shirts$', $revenue$>21, $prod.$ in ('$shoes$', '$shirts$')

By setting a minimum frequency threshold T, we can now use an off-the-shelf frequent itemset mining algorithm to compute the predicate sets that subsume at least T queries.

3.3.3 Redundant Predicate Set Elimination

We now have obtained a set of predicate sets that subsume at least T queries. Unfortunately, some of these predicate sets may be redundant. For example, if there is a predicate set {$publisher$='$google$', $revenue$<0} in the result, both its subsets {$revenue$<0} and {$publisher$= '$google$'} would also be present, due to the apriori property [8]; but the predicate set {$publisher$='$google$', $revenue$<0} is only useful for the queries that cannot be subsumed by either of the subsets. In addition, the filter augmentation step may also introduce redundant results. For example, if {$revenue$> 32} is present, then {$revenue$>21} is also present due to the augmentation. Again, {$revenue$>21} is only helpful for the queries that cannot be subsumed by {$revenue$>32}. Existing approaches that keep only *maximal* or *closed* frequent itemsets [37] do not capture the subsumption relations in our specific problem.

We develop a principled way to remove redundancy from the frequent predicate sets. Notice that the frequency threshold T used in Section 3.3.2 indicates that we are only interested in the

predicate sets that subsume at least T queries. We will use the same principle for redundancy removal. Specifically, let $\mathcal{F}$ be the frequent predicate sets, we consider a predicate set F_i in $\mathcal{F}$ to be redundant and remove it, if the number of queries F_i *additionally* subsumes, given the predicate sets that are already in $\mathcal{F}$, is less than the threshold T. We denote by $subsume(F_i, \mathcal{Q}) \subseteq \mathcal{Q}$ the set of queries in $\mathcal{Q}$ that are subsumed by F_i. We use the following procedure to eliminate the redundant predicate sets:

---

sort $\mathcal{F}$ by the (partial) order of $\sqsubseteq$, i.e., $F_i \sqsubseteq F_j$ for $i < j$
$\mathcal{Q}_c \leftarrow \emptyset, \mathcal{R} \leftarrow \emptyset$
for each F_i in $\mathcal{F}$:
  $I(F_i) \leftarrow |subsume(F_i, \mathcal{Q}) - \mathcal{Q}_c|$
  if $I(F_i) < T$:
    remove F_i
  else:
    $\mathcal{R} \leftarrow \mathcal{R} \cup \{(I(F_i), F_i)\}$
    $\mathcal{Q}_c \leftarrow subsume(F_i, \mathcal{Q}) \cup \mathcal{Q}_c$

We examine the predicate sets in $\mathcal{F}$ in the (partial) order of increasing generality, such that the eliminating decision of the current predicate set does not affect that of the previous predicate sets. For each F_i, we calculate $I(F_i)$, the number of queries that F_i additionally subsumes. If $I(F_i)$ is smaller than T, F_i is removed; otherwise, we add the pair ($I(F_i)$, F_i) to $\mathcal{R}$ and update $\mathcal{Q}_c$, the cumulative set of queries subsumed by all the predicate sets in $\mathcal{R}$. At the end, we sort $\mathcal{R}$. Given a parameter $numFeat$, the workload analyzer returns $numFeat$ predicate sets from $\mathcal{R}$ with the highest $I(\cdot)$ values.

The cost of workload analysis is dominated by the frequent itemset mining phase. In practice, we expect this process to be very efficient, since most queries would not have many predicates. Next, we discuss the partitioner, which incorporates the data-related aspects such as correlation and selectivity, by solving a optimization problem on the feature vectors.

3.4 The Partitioning Problem

In this section, we first formulate the optimization problem and prove its hardness. We then discuss the reduction step which is critical to scaling. Finally, we present the bottom-up framework.

3.4.1 Problem Definition

Suppose we have a set of m features obtained from the workload analysis, denoted by $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$. We denote by w_j the weight of the features F_j, i.e., the number of queries it subsumes. Based on these features, the data tuples are augmented with binary vectors (Step (1) in Figure 3.3). We now formulate the partitioning problem on the m-dimensional bit vectors.

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a collection of m-dimensional bit vectors, where each vector corresponds to a tuple. We will focus on our discussion on the vectors, finding as a partitioning

on the vectors is equivalent to finding a partitioning on the tuples. The j-th bit of v_i, denoted by v_{ij}, indicates whether v_i satisfies feature F_j. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be a *partitioning* over V, i.e., $\mathcal{P}$ is a set of disjoint subsets of V whose union is V. Let $\overline{v}(P_i)$ be the union vector of all vectors in P_i, i.e., $\overline{v}(P_i) = \bigvee_{v_j \in P_i} v_j$. We say a feature F_j *prunes* a partition P_i if none of the vectors in P_i satisfies F_j, i.e., the j-th bit of $\overline{v}(P_i)$ is 0, denoted by $\overline{v}(P_i)_j = 0$. If F_j prunes P_i, then F_j prunes $|P_i|$ tuples, as each vector corresponds to a tuple. Note that the weight w_j of F_j is the number of queries F_j subsumes in the workload. If F_j prunes P_i, when we run all the queries in the workload, the sum of tuples that can be skipped would be $w_j \cdot |P_i|$. Given a partition P_i, we define the cost function $C(P_i)$ as the sum of tuples in P_i that can be skipped when we execute all the queries in the workload, we have:

$$C(P_i) = |P_i| \sum_{1 \le j \le m} w_j (1 - \overline{v}(P_i)_j) \tag{3.1}$$

Consider the example partitioning scheme P_1 in Figure 3.1(d). The union vector $\overline{v}(P_1)$ is $(1, 1, 0)$, so only feature F_3 prunes P_1. We also know that $|P_1| = 2$ and the weight of F_3, w_3, is 10 as given in Figure 3.1(b). Therefore, we have: $C(P_1) = 2 \times 10 = 20$ using Equation 3.1. We then define the cost function $\mathbb{C}(\mathcal{P})$ over a partitioning, which is the sum of $C(P_i)$ over all P_i in $\mathcal{P}$:

$$\mathbb{C}(\mathcal{P}) = \sum_{P_i \in \mathcal{P}} C(P_i) \tag{3.2}$$

Intuitively, the objective function $\mathbb{C}(\mathcal{P})$ is the sum of tuples that can be skipped if we run all the queries in the workload. From the perspective of a real system, we also have to constrain that the block sizes are almost balanced. If some block is too small, it incurs significant overhead to process the block and maintain block-level metadata; if some block is much bigger than others, it may become a straggler in parallel executions. We now define the *Balanced MaxSkip* partitioning problem:

Problem 1 (Balanced MaxSkip Partitioning) *Given a set V of binary vectors, where $|V|$ is a multiple of p, find a partitioning $\mathcal{P}$ over V such that $\mathbb{C}(\mathcal{P})$ is maximized, i.e.,*

$$\arg\max_{\mathcal{P}} \mathbb{C}(\mathcal{P})$$
$$\textit{subject to} \qquad |P_i| = p \qquad \forall P_i \in \mathcal{P}$$

The balance constraint is important from the perspective of a large-scale system, such as parallel processing of data blocks, but not necessary for data skipping purposes. For theoretical interests, we also formulate a *k-MaxSkip partitioning* problem without the balance constraint:

Problem 2 (k-MaxSkip Partitioning) *Given a set V of binary vectors, find a k-partitioning $\mathcal{P}$ over V such that $\mathbb{C}(\mathcal{P})$ is maximized.*

In the design of SOP, we will focus on Problem 1.

3.4.2 NP-Hardness

We now prove that Problem 1 is NP-hard even in the special case where $p = |V|/2$, by reduction from hypergraph bisection [29]. Given a hypergraph instance $H = (V, E)$, where V is a set of vectices and E is a set of hyper edges. Since each hyper edge connects a set of vertices, we denote by $v_i \in E_j$ if v_i is in E_j. The hypergraph bisection problem finds a balanced 2-partitioning on the vectices such that the number of hyperedges across the two partitions is minimum.

We construct an input to our problem from this instance as follows. We construct a bit vector v_i for each vertex. Each vector has $|E|$ dimensions, where the j-th dimension corresponds to an edge e_j in E. Specifically, the value v_{ij} is 1 if $v_i \in e_j$ and 0 otherwise. By setting the partition size to be $|V|/2$, a solution $\mathcal{P} = \{P_1, P_2\}$ to our problem is a balanced partitioning over V. Let n_1 and n_2 be the number of vectors in P_1 and P_2, respectively, and let m_1, m_2 and m_c be the number of hyperedges in P_1, in P_2 and across both, respectively. The value of our objective function for this solution is:

$$\mathbb{C}(\mathcal{P}) = n_1 m_1 + n_2 m_2 + n m_c \tag{3.3}$$

Now that the partitions are balanced, i.e., $n_1 = n_2$, we have:

$$\mathbb{C}(\mathcal{P}) = \frac{1}{2} n m_1 + \frac{1}{2} n m_2 + n m_c = n m + n m_c \tag{3.4}$$

Since n and m are constant, minimizing Equation 3.4 is equivalent to minimizing m_c. Hence, an optimal solution to our problem solves the hypergraph bisection problem on H.

From Equation 3.3, we can see that our objective function involves the product of number of vertex and number of edges for each vertex partition. We conjecture that Problem 2, the variant that does not have the balance constraint, is also NP-hard.

3.4.3 Reduction Step

The partitioning problem defined in Section 3.4.1 is solely based on the vectors, not the actual tuples. As an optimization step, we *group* the vectors into (vector, count)-pairs. This step does not affect the quality of partitioning, since the same vectors should go to the same partition anyway. Each vector now is associated with a weight, denoting the number of tuples it represents. To compute Equation 3.1, we simply replace $|P_i|$ with the weighted sum of all the vectors in P_i.

Note that this step is the key to scale our partitioning techniques to large datasets. Theoretically, the number of distinct vectors is upper-bounded by $min(2^m, n)$. In practice, however, this number is usually very small, e.g., <$10k$, as many tuples may have the same feature vectors. We will empirically examine this number in Section 3.7.

3.4.4 The Bottom Up Framework

Since Problem 1 is NP-hard, we will turn to Ward's method as a heuristic algorithm to find an approximate solution efficiently. Ward's method [69], originally proposed for minimizing the

error sum of squares, is a general bottom-up clustering framework and has been used for various objective functions [52].

Base on Ward's method, every data point is a partition by itself initially. At each iteration, we select two partitions to merge that maximizes $\mathbb{C}(\mathcal{P})$. Recall that $\mathbb{C}(\mathcal{P})$ is a sum of $C(P_i)$ for all $P_i \in \mathcal{P}$. We denote by $\delta(P_i, P_j)$ the change of $\mathbb{C}(\mathcal{P})$ caused by merging partitions P_i and P_j, i.e.,

$$\delta(P_i, P_j) = \mathbb{C}(\mathcal{P} \cup \{P_i \cup P_j\} - \{P_i, P_j\}) - \mathbb{C}(\mathcal{P}) \tag{3.5}$$

When we merge P_i and P_j, their union vectors are ORed, i.e.,

$$\overline{v}(P_i \cup P_j) = \overline{v}(P_i) \vee \overline{v}(P_j) \tag{3.6}$$

Since the merging of P_i and P_j does not affect the costs of other partitions, we have:

$$\begin{aligned} \delta(P_i, P_j) &= C(P_i \cup P_j) - C(P_i) - C(P_j) \\ &= |P_i| \sum_{1 \le k \le m} w_k(\overline{v}(P_i)_k - \overline{v}(P_i)_k \vee \overline{v}(P_j)_k) \\ &\quad + |P_j| \sum_{1 \le k \le m} w_k(\overline{v}(P_j)_k - \overline{v}(P_i)_k \vee \overline{v}(P_j)_k) \end{aligned}$$

In the bottom up algorithm, we represent each partition as P_i as a $(\overline{v}(P_i), |P_i|)$-pair. Thus, $\delta(P_i, P_j)$ can be evaluated efficiently (constant time to the size of partitions). We also have the following lemma.

Lemma 1 (Monotonicity) *The objective function $\mathbb{C}(\mathcal{P})$ is non-increasing through a partition merge, i.e., $\delta(P_i, P_j) \le 0$ for any $P_i, P_j \in \mathcal{P}$.*

Lemma 1 guarantees that our objective function can fit in Ward's method correctly. The objective $\mathbb{C}(\mathcal{P})$ has the maximal value when every vector is a partition by itself. We iteratively find a pair of points whose merge hurts $\mathbb{C}(\mathcal{P})$ the least.

In practice, we set a parameter $minSize$. A partition is removed from being further merged if its size reaches $minSize$. Thus, a merge of two blocks of size less than $minSize$ would be smaller than $2 \cdot minSize$. We simply accept the blocks of size in $[minSize, 2 \cdot minSize)$. The bottom-up procedure is shown as follows:

$\mathcal{P} \leftarrow \{\{v_1\}, \{v_2\}, \ldots, \{v_n\}\}, \mathcal{R} \leftarrow \emptyset$
while $\mathcal{P}$ is not empty:
· *merge the pair $P_i, P_j \in \mathcal{P}$ with the largest $\delta(P_i, P_j)$*
· *if $|P_i \cup P_j| > minSize$ or $P_i \cup P_j$ is the last one in $\mathcal{P}$:*
  · *remove $P_i \cup P_j$ from $\mathcal{P}$ and add it to $\mathcal{R}$*
return R

A straightforward implementation of this algorithm has a complexity of $O(n^2 \log n)$ [52]. As bottom-up clustering is a classic algorithmic framework, scalable implementations have been

proposed, e.g., [75, 31]. Due to the reduction step, the input size becomes much smaller than the number of tuples. The experimental results in Section 3.7 indicate that the partitioning algorithm can run efficiently even with a naïve implementation, and the cost bottleneck of the entire partitioning process is still on the actual data movement, not on the partitioning algorithm.

Constructing the partitioning map. The output of the algorithm is a partitioning of the vectors. We use this output to construct a *blocking map* which returns a block id for a given feature vector. A data tuple can be routed to the right block by consulting the partitioning map with its corresponding feature vector.

3.5 Feature-based Data Skipping

In order to enhance the data skipping, we now introduce the block catalog, which maintains the metadata for skipping. We also discuss how a query optimizer might use it.

As mentioned earlier, many query engines support data skipping using the value ranges or bloom filters. We could import our block data to these systems and rely on the built-in skipping mechanism. This, however, may miss some data skipping opportunities. For example, the features we used to guide partitioning can involve multiple columns (e.g.,F_3 in Figure 3.1); checking the aggregates of the two columns individually may not be able to prune the block, while maintaining multi-dimensional statistics is expensive. In addition, it is unclear how the aggregates can be used to prune string matching and general user-define functions. To fully exploit our partitioning scheme for data skipping, we propose a feature-based skipping mechanism, which can be used in conjunction with existing data skipping mechanisms based on the aggregates.

Block Catalog. After the partitioning, we obtain a union vector for each block. If the i-th bit of the union vector is 0, then we know that no tuple in this block satisfies feature i. Therefore, the queries that are subsumed by feature i can safely skip this partition. To realize this, we store the features used and one union vector for each block in a *block catalog*, which can be part of the system catalog in a database system. Figure 3.4 shows the partition catalog for the example partitioning scheme in Figure 3.1. Note that the block catalog is very compact: we store the features once and then one bit vector for each block.

Query Execution. Let us see how a query interacts with the block catalog in Figure 3.4. When a query comes, we first extract the filter operator from the query. We check which features in the block catalog can subsume this query. The subsumption check can be implemented as hard-coded rules. We find that F_1 and F_2 can subsume the query. We represent this information by a *query vector* $(0, 0, 1)$, the i-th bit of which is 0 if i-th feature subsumes the query. We then compute, for each block, a bitwise OR between the query vector and the union vector. For any block, if the resulting vector of the OR operation has at least one 0 bit, then this block can be skipped. In Figure 3.4, the ORed vectors for P_1, P_2 and P_3 are $(1, 1, 1)$, $(0, 1, 1)$ and $(1, 0, 1)$ respectively. Thus, we know that we only need to scan P_1. This information is then passed to table scan operator.

The above procedure happens before the table scan operator and is independent of the rest of query execution. As we can see, we only need to maintain minimal metadata and add an simple

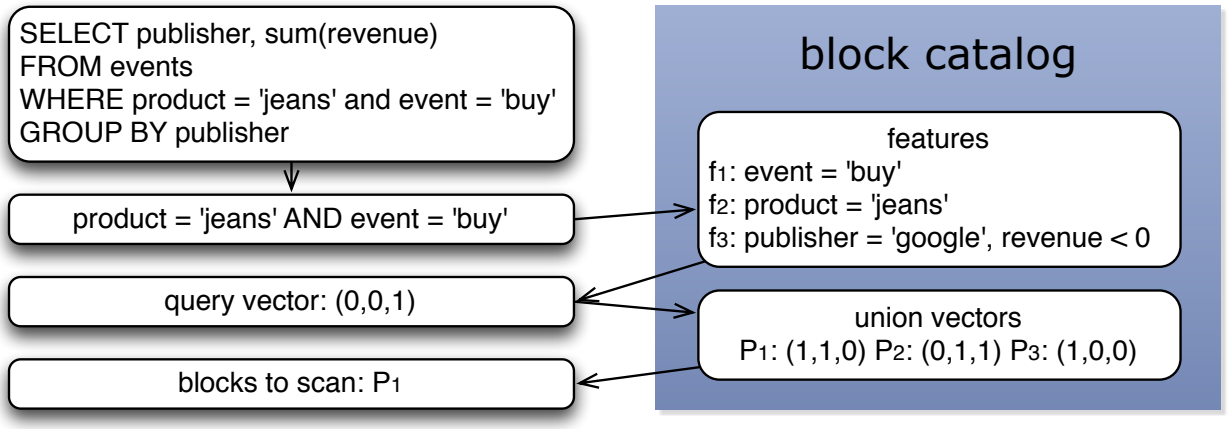


Figure 3.4: Data Skipping in Query Execution

module between query compilation and the start of the query execution. We have implemented this functionality in less than 100 lines of code in Shark [70].

3.6 Discussion

In this section, we discuss the practical issues of using the SOP framework.

Data Update. SOP is designed for a data warehouse setting, where there are infrequent ad-hoc updates and data are batch-inserted and batch-deleted as partitions. We can apply SOP on each partition separately. Therefore, the insertion or deletion of partitions will not affect other parts of data. Nevertheless, we now discuss how ad-hoc updates can be handled for our blocked data.

Since SOP produces small balanced-sized data blocks, ad-hoc insertions should be put into new blocks instead of modifying existing blocks. These new blocks are ready to serve queries. Once we have accumulated enough new data, e.g., 50 blocks' worth of data, we can use SOP to re-organize these data. An ad-hoc deletion can be handled trivially, as a tuple can be removed from a block without modifying the block metadata. When there is an ad-hoc update in a block, we check if we need to update the block metadata, i.e., the union vector. To allow for high-throughput updates, we can simply *invalidate* the union vector by setting all of its bits to 1's to eliminate the checking overhead. After many updates, the partitioning scheme may become ineffective, then we can re-partition the data.

Parameter Selection. Two key parameters were used in the SOP process, namely, $numFeat$, the number of features, and $minSize$, the minimum number of tuples per block. For a real-world workload, we expect $numFeat$ to be small. We can use cross validations to choose an appropriate number of features that prevents under- and over-fitting. The choice of $minSize$ can depend on the underlying system. In the prototype on Shark, we set $minSize$ to be $50,000$, which results in 64-128MB blocks. Each block nicely fits with a Spark/HDFS block. Intuitively, if a block is bigger

than 128MB, the system will break it down into many HDFS blocks anyway. If the blocks are too small, the metadata storage and skip checking overhead may become significant.

Overhead of Skip Checking. In many large-scale query engines, the query optimizer runs in a single node. We can easily perform the skip checking (Section 3.5) in a single-node optimizer. This might seem counter-intuitive, as a large table can have many blocks. To see if this makes sense, we can do some back-of-the-envelope analysis. Assuming an average block size of 128MB, 100TB of data would have less than 1 million blocks. It only takes 13MB to store the bit vectors for all these partitions if each vector is 128-bit, i.e., 128 features used. The block catalog can easily fit in the memory of a single machine. In a modern CPU, a bitwise operation of two 128-bit vectors can be computed using only a single instruction. The block checking incurs minimal overhead even for interactive queries.

Objective Function. The objective function for partitioning (Equation 3.2) is defined as the sum of query costs in the workload, where the cost of a query is quantified as the number of tuples scanned. We chose the current objective function as it is simple, commonly used in database designs (e.g., [33]) and works reasonably well in practice. In some applications, we may want to define the objective differently, e.g., we can maximize the number of queries in the workload that can finish in 30 seconds. In this case, we may need to estimate the query costs and weight the queries based on their costs. Incorporating these in the SOP framework would be an interesting avenue of future work.

3.7 Experimental Evaluation

In this section, we report the experiment results. All experiments were conducted on an Amazon Spark EC2 cluster of 25 m2.4xlarge instances, each with 8×2.66 GHz CPU cores, 64.8 GB of RAM and 2×840 GB disk storage. The datasets are stored in HDFS.

3.7.1 System Prototype

We prototype the skipping-oriented partitioning techniques on Shark [70], a fully Apache Hive [6]-compatible data warehousing system using Apache Spark [44] as runtime. Shark parses and compiles HiveQL (SQL-like) queries to a query plan, which are then translated to Spark tasks. A Shark table is stored as a Spark data abstraction called Resilient Distributed Dataset (RDD), which is physically stored as a list of data blocks, each of which can be either memory- or disk-resident. Each block in Spark is a data processing unit and has a default size of 128MB. Shark supports data skipping over such data blocks. At data import time, Shark collects the data statistics for each block. These statistics are maintained in the system catalog. Before a table scan, the query filter is applied on these statistics, and then the block ids to be scanned are passed to the table scan operator.

We now briefly discuss how our techniques were implemented on Shark.

Workload Analyzer. We can collect a query trace from the query logging system of Shark or from an external source. We used Shark's query parser to convert each query string into a

set of conjunctive filter operators. We implemented a `isSubsume`(f_1, f_2) function using a set of rules to check if filter f_1 subsumes filter f_2. A module was added in Shark to implement the techniques in Section 3.3.

Featurization, Reduce. We wrote a map function for featurization and a reduce function to group by the vectors.

Partition, Shuffle. We implemented a bottom-up clustering algorithm as a module in Shark. Note that this module is independent of the other parts of the SOP workflow, and thus external libraries could be used here. We then constructed a Spark Partitioner, which returns the destination block id for an (vector, tuple) pair. The table (an RDD) are then shuffled using this Partitioner.

Catalog Update. We added the metadata described in Section 3.5 to the Shark system catalog. A table scan now can utilize two tiers of skipping mechanisms: our feature-based skipping (Section 3.5) and the existing min/max skipping.

3.7.2 Datasets

TPC-H

We use the TPC-H benchmark with a scale factor of 100. To focus on the effect of reduced table scan, we denormalize all the tables against the *lineitem* table, which results in a single table of roughly 600 million rows and 700 GB in size. We select eight query templates (q_3, q_5, q_6, q_8, q_{10}, q_{12}, q_{14}, q_{19}) from the TPC-H workload, as these templates involve the *lineitem* table and have selective filters. The FROM clauses in these templates were all changed to be the denormalized table. In a query template, some filters are fixed while the others are parametric. For example, the return item reporting query (q_{10}) has a fixed filter *l_returnflag = 'R'*, which will appear in every query generated from this template, and a parametric filter *o_orderdate >= date '[DATE]'*, where *[DATE]* is replaced with a constant at each run. Using the TPC-H query generator, we generate 800 queries as the training workload, 100 from each template. We then independently generate 80 queries for testing, 10 from each template. TPC-H represents a workload of template-generated queries, which is very common in real-world applications. Note that our approach can greatly take advantage of the fixed filters in a template (e.g., *l_returnflag = 'R'*), as they are perfectly predictable.

TPC-H Skewed

The TPC-H query generator assumes a uniform distribution for the predicate constants. In many real-world workloads, the constant distributions are skewed. For example, *region='North America'* may be queried much more often than the other regions for a U.S. company. To test a skewed distribution, we modified the TPC-H query generator to follow a Zipf distribution. As most parameters in the query templates have a small number of possible values, we chose a high skew factor of 3. For example, REGION only has 5 possible values, and by using a skew factor of 3, the most frequent 20% values occur in 84.3% of the generated queries. Note that we only added

the skewness to the non-date filters, while [DATE] is still uniformly distributed. We similarly generate 800 train queries and 80 test queries under this Zipf distribution.

Conviva

The Conviva data is an anonymized user access log of video streams. The data consists of a single large fact table with 104 columns, such as customer ID, city, media URL, genre, date, time, browser type and request response time. We also obtained an in-production SQL query trace from Conviva, which has 735 queries for problem diagnosis and data analytics issued on the log data. The queries were between 08/01/2012 and 11/30/2012. We split the query trace at the date of 11/24/2012, which results in 674 training queries (before 11/24/2012) and 61 testing queries (on and after 11/24/2012). Based on the training queries, we partition the log data from 11/24/2012 to 11/30/2012. This snapshot of the log has 680 million tuples and is roughly 1 TB in size when stored as text. We evaluate the performance of running the test queries on the partitioned data.

3.7.3 TPC-H Results

Query Performance

We evaluate the effect of our partitioning techniques on query performance. We measure the number of tuples scanned and end-to-end query response time for different partitioning and skipping schemes in Figure 3.5. Specifically, we compare the following alternatives:

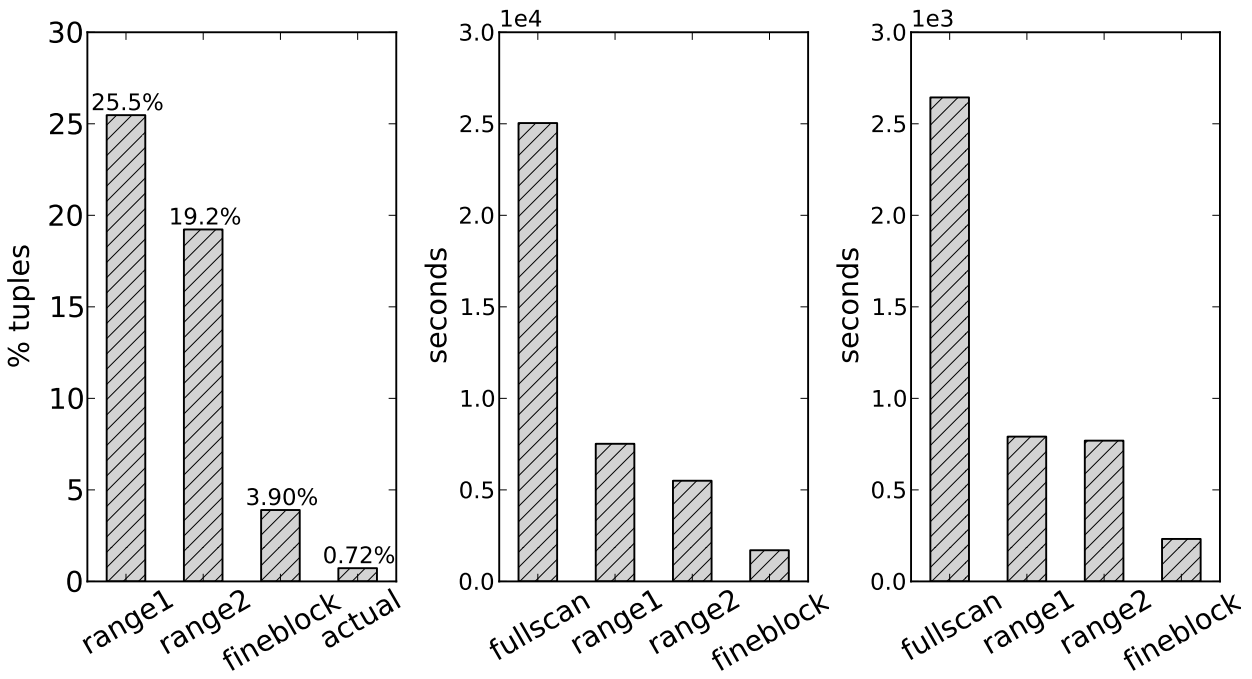
`fullscan`: We disable the data skipping and do a full scan for each query. The partitioning scheme is immaterial here.

`range1`: We perform a workload-oblivious partitioning on *o_orderdate* and each date is a partition. This leads to roughly 2300 partitions. Shark's data skipping is used.

`range2`: We manually devise a composite range partitioning scheme on multiple columns. By identifying the frequently queried columns from the workload, we perform a range partitioning on {*o_orderdate* (by month, 78 partitions), *r_name* (customer region name, 5 partitions), *c_mkt- segment* (5 partitions), *quantity* (5 partitions)}. This results in roughly 9000 partitions. Shark's data skipping is used.

`fineblock`: This is the SOP approach. We first partition by month on *o_orderdate*. We use the workload analyzer to extract 15 features from the 800 training queries (by setting *numFeat*=15). Note that we do not consider any date filters as features and will rely on the month partitions to prune data. Using these features, we run a partitioner instance on each month partition in parallel. An average month partition has 7.7 million tuples. By setting *minSize*= $50k$, a month partition has around 100 blocks. The total number of blocks is roughly 8000. We used both our feature-based skipping (Section 3.7) and Shark's existing min/max skipping.

We evaluate the performance of running 80 test queries (as mentioned in Section 3.7.2) using the above alternatives. Figure 3.5(a) shows the percentage of tuples scanned for the 80 queries relative to `fullscan`. As we can see, the existing data skipping with `range1` and `range2` only scan 25% and 19% of the tuples scanned by `fullscan` respectively. The tuples scanned by



(a) % tuples scanned (b) on-disk query (c) in-memory query

Figure 3.5: Query Performance Results of SOP (TPC-H)

`fineblock` is only 3.9% of `fullscan`, a 5x improvement over the manual range partitioning scheme `range2`. As a reference, the bar `actual` shows the percentage of tuples that must be scanned, i.e., the tuples that actually satisfy the filters, which is 0.7% of `fullscan`.

To test the end-to-end query response time, we consider two scenarios: when the table is entirely on disk and when the data is entirely in memory. Figure 3.5(b) shows the query response times for on-disk data. We run the 80 test queries in a sequence and record the sum of their response time. We cleared the OS cache before running each query. As shown, the query response time for `range1` and `range2` are 30% and 21% of that for `fullscan`. `fineblock` only took 7% of the time for *fullscan*, 23% for `range1` and 30% for `range2`. This is an 3-4x improvement over the range partitioning schemes. Figure 3.5 shows the query response time for memory-resident data. In Shark, we can simply cache a table in memory by executing:

```
create table tpch_cached as select * from tpch;
```

We configured our distributed RAM cache to be large enough to hold the entire table. As we can see, `fineblock` only took 30% and 32% of the time taken for `range1` and `range2`, respectively. This is roughly a 3x improvement. It is interesting to note that the end-to-end improvement for in-memory data is slightly smaller than that for on-disk data. As scanning in-memory data is much faster, the effect of skipping in-memory blocks is less significant.

We also tested the end-to-end performance for TPC-H Skewed. The amount of tuples scanned by `fineblock` is only 8% and 10% of that by `range1` and `range2` respectively. For the on-

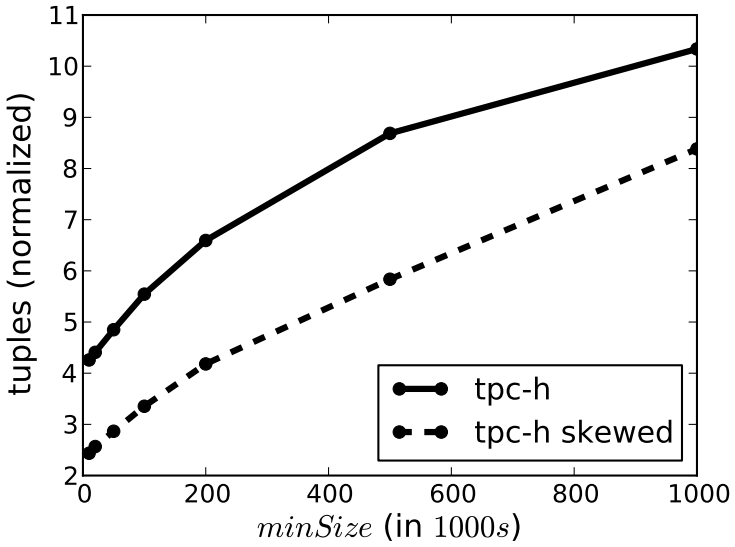


Figure 3.6: Effect of *minSize* in SOP (TPC-H)

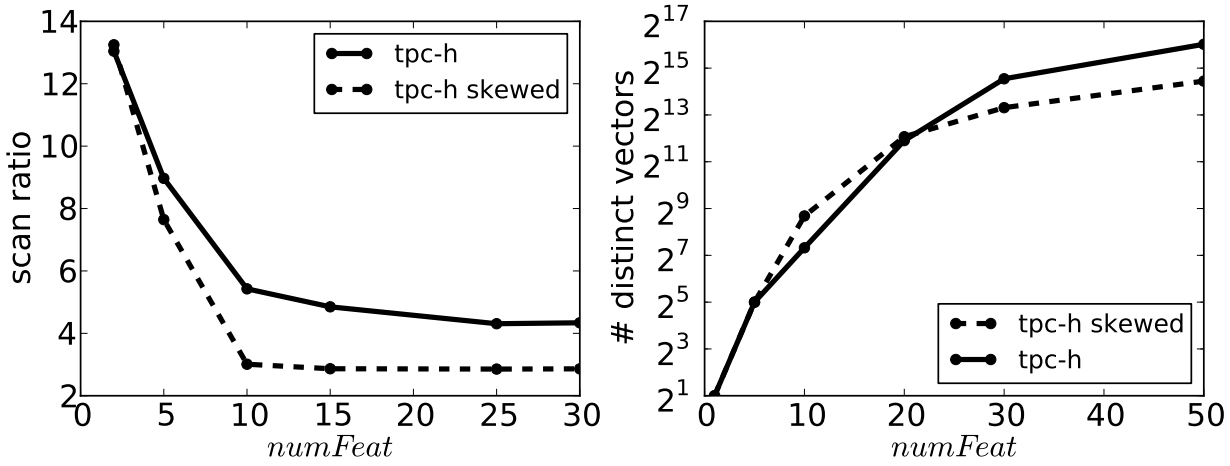
disk data, `fineblock` took 20% and 23% of the time for `range1` and `range2` respectively; for the in-memory data, `fineblock` took 22% and 29% of the time for `range1` and `range2` respectively. Note that our improvement for TPC-H Skewed is better than for TPC-H. The skewness in the filter distribution allows a small number of features to subsume even more queries, and thus makes our techniques even more effective for TPC-H Skewed.

On both TPC-H and TPC-H Skewed, we observe that our approaches significantly reduce the data scanned, which effectively translate to an improvement in end-to-end query response time for both disk- and memory-resident data.

Effect of $minSize$

Intuitively, the smaller the block size is, the more chance we can skip data. In Figure 3.6, we plot the total number of tuples scanned using our approach for answering the 80 test queries in TPC-H and TPC-H Skewed by varing $minSize$, with $numFeat$=15. Since the two curves represent two different workloads, for fair comparison, we plot the ratio of the number of tuples scanned to the number of tuples that have to be scanned. Thus, a y-value of 5 in the curve means we scanned 5 times as many tuples as necessary.

We make several observations. First, for both curves, we scan more data as the block size increases. Second, data skipping is even more effective on TPC-H skewed. Recall our workload assumptions in Section 3.2, many real workloads tend to have skewed predicate distribution. If the filter predicates are skewed, a small set of features can cover more queries. Third, the number of tuples scanned is not sensitive to the block size. In particular, increasing *minSize* from $5k$ to $200k$ only make the scan twice as much for both workloads. This gives us a wide range of



(a) % tuples vs. $numFeat$ (b) # vectors vs. $numFeat$

Figure 3.7: Effect of $numFeat$ in SOP (TPC-H)

choosing $minSize$. For example, in our experiment on Shark, we set *minSize*=$50k$, which make each block nicely fit in a Spark/HDFS block file (128MB by default).

Effect of $numFeat$

Figure 3.7 plots the number of tuples scanned by varying $numFeat$. The numbers are also normalized as in Figure 3.6. As we can see, when using too few features, e.g., < 5, we have to scan a lot of more tuples, as these features are not representative enough for the workload. As we add more features, the effectiveness of skipping quickly stabilizes and TPC-H Skewed consistently benefits more from SOP. We can see that, for both TPC-H and TPC-H Skewed, a small set of features is sufficient. Even though the predicates in TPC-H are uniformly distributed, the fixed filters play an important role as features. For example, the feature *l_returnflag = 'R'* in template q_{10} can subsume 100 out of 800 training queries (and also 10 out of 80 testing queries), since all the queries generated from template q_{10} have this filter. TPC-H Skewed performs even better due to the skewness in the parametric predicates. This curve also suggests that adding more features will not significantly hurt the effectiveness of skipping, though this may hurt the partitioning efficiency. This is because the highly weighted features will dominate and adding more features with small weights will not dramatically change the partitioning solution.

As discussed, one important effect of $numFeat$ is on the number of distinct feature vectors. We cannot afford to run a bottom-up clustering algorithm if the number of distinct vectors is too large, e.g., close to the number of tuples. Theoretically, with m features and n tuples, the number of distinct vectors is upper-bounded $min(2^m, n)$. In Figure 3.7, we plot the actual number of distinct vectors by varying $numFeat$ in base-2 log scale. The plot shows that the number of distinct vectors is much smaller than either 2^m or n. This is because these features have low selectivities and can be correlated.

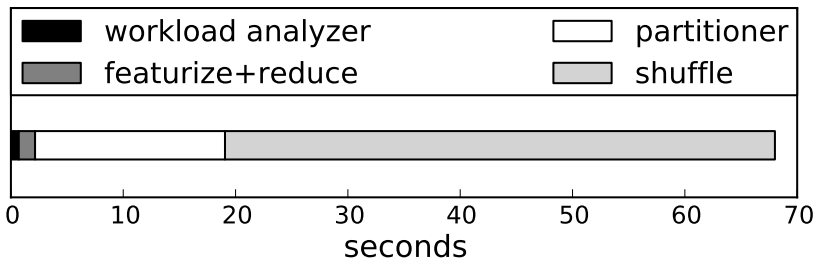


Figure 3.8: Loading Time using SOP (TPC-H)

Loading Time

Figure 3.8 shows the breakdown cost of applying SOP on a month partition in TPC-H. An average month partition has 7.7 million tuples and 8G in size and can be divided into roughly 100 blocks. We set $numFeat = 15$ and $minSize = 50$. It took about 1 minute for the entire workflow. When loading multiple partitions simultaneously, we can run multiple SOP partitioning processes in parallel. As we can see, the shuffling is still the bottleneck, although we run sophisticated algorithms in the workload analyzer and the partitioner. The workload analyzer runs a frequent itemset-based algorithm from 800 queries. The partitioner runs a bottom-up clustering algorithm on 1315 feature vectors. In our experiments, we only used our own vanilla implementations for these algorithms running on a single thread, which were sufficiently efficient. Notice that both components can be further optimized, e.g., using an off-the-shelf library. We combined the cost of featurization and reduce, as they were implemented as Spark map and reduce functions which can pipelined.

3.7.4 Conviva Results

Query Performance

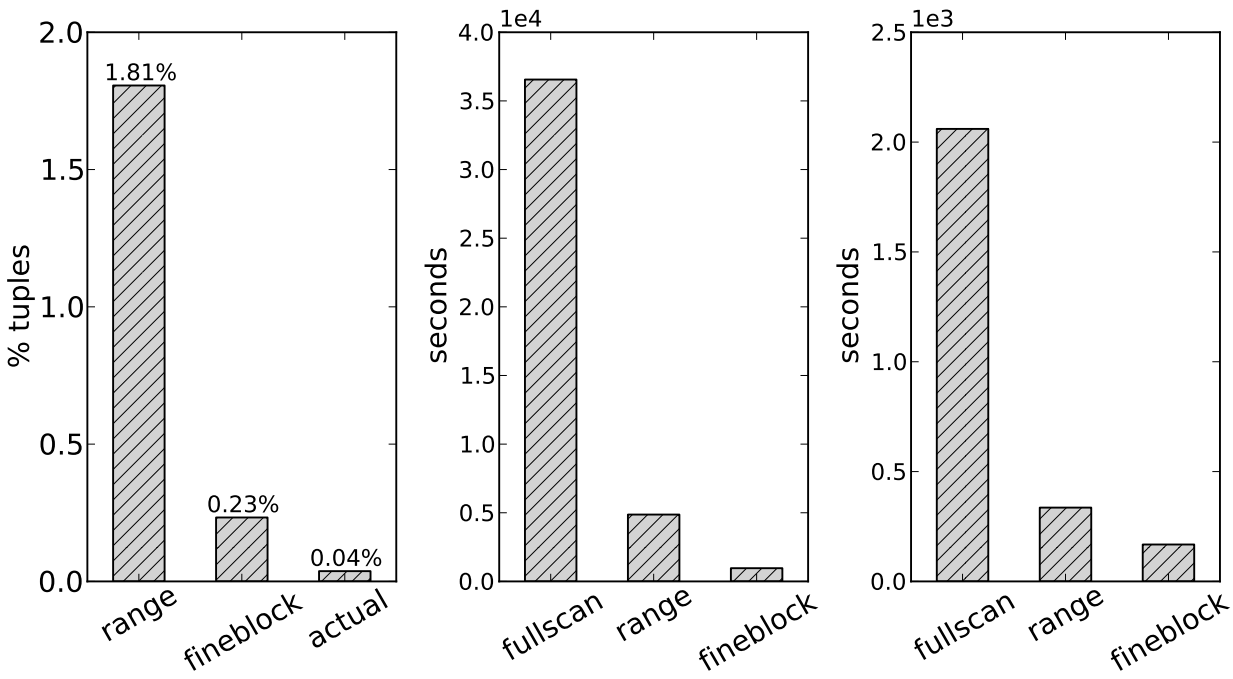
For evaluating the query performance, we compare the following alternatives:

`fullscan`: We perform a full scan for each query.

`range`: We perform a range partitioning on date and a frequently queried column. We use Shark's data skipping.

`fineblock`: We first partition by date. After extracting 40 features from the training queries (*numFeat*=40), we block each date partition with *minSize* =$50k$. We use both our feature-based and Shark's existing skipping mechanisms.

Figure 3.9(a) shows the percentage of tuples scanned for evaluating the test queries, relative to `fullscan`. As we can see, `range` already reduced the scan to be 1.81% of `fullscan`, `fineblock` only scans 0.23% of the data. The average selectivity of these queries (i.e., `actual` is 0.03% of `fullscan`. Figure 3.9(b) shows the query response times for on-disk data. We can find that `range` spent 13% the time as `fullscan`, and `fineblock` further reduced the time



(a) % tuples scanned (b) on-disk query (c) in-memory query

Figure 3.9: Query Performance Results of SOP (Conviva)

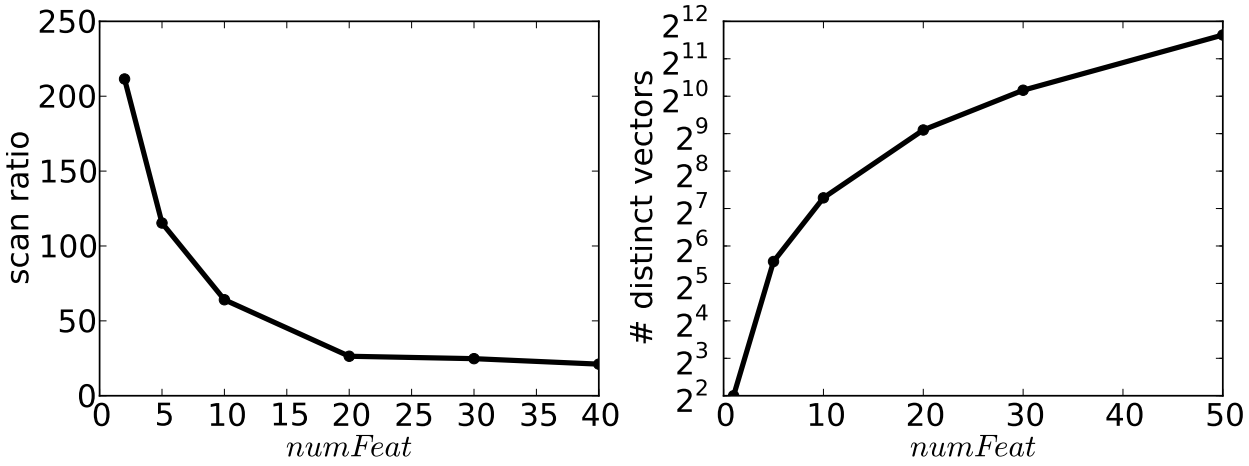
to be 2.6% of `fullscan`. This is a 5x improvement over `range`. Figure 3.9(c) shows the query time for in memory data. We find that `range` and `fineblock` used 16% and 8.1% of the time taken by `fullscan`. The improvement of `fineblock` over the range partitioning schemes is 2x.

As scanning in-memory data is fast, the effect of data scan reduction is diminished by the cost from other parts of the query evaluation such as aggregations. Specifically, we observed that some Conviva queries computed many aggregated values in the SELECT statement, which can be CPU-intensive after the data scan. Incorporating techniques that speed up these aggregations (e.g., materializations) may make our end-to-end improvement more significant.

Effect of $numFeat$

We now study the effect of $numFeat$ on the Conviva workload. Figure 3.10(a) plots the scan ratio by varying $numFeat$. The number of tuples scanned is dramatically reduced as $numFeat$ is increased from 2 to 20. As we continue to add more features, however, the curve is relative flat. This is inline with the results on TPC-H (Figure 3.7), except that this curves starts to stabilize at 20 instead of 10. This result confirms that a small number of features is sufficient for a real workload. In Figure 3.10(b), we can see that the number of distinct vectors is small.

We can conclude that, in a real-world workload, (1) our blocking can effectively help queries skip data and in turn reduce the query response time significantly for both on-disk and in-memory



(a) % tuples vs. $numFeat$ (b) # vectors vs. $numFeat$

Figure 3.10: Effect of $numFeat$ in SOP (Conviva)

data, and (2) the blocking can be done effectively and efficiently with a small number of features.

3.8 Related Work

In this section, we review the related work.

Horizontal Partitioning

Range and hash partitioning are the most widely used horizontal partitioning techniques and serve for many purposes, such as load balancing. Advanced and automated partitioning techniques have also been extensively studied [15, 76, 55, 9], but they were built on top of range or hash partitioning. Although the partitioning problem we study is a form of horizontal partitioning, SOP generates a tuple-level partitioning map by solving an optimization problem instead of using explicit range constraints. SOP can be used to further segment a date-range partition into finer blocks for skipping purposes. While Schism [24] also used fine-grained tuple-level partitioning, they had a different objective, which is reducing cross-machine transactions for OLTP workloads.

Materialized Aggregates and Skipping

Many databases utilize range partitions to enhance query performance. Partition pruning (e.g., Oracle [49] and Postgres [50]) allows queries to skip partitions based on partition key ranges (e.g., date). Extending this idea, other works [48, 27] have proposed maintaining small materialized aggregates (SMAs) for each range-partitioned block, such as min, max, count, sum and histograms

for each column. For a given query, these SMAs can be used to classify the data blocks into three categories: (C1) *irrelevant* blocks, the ones in which no tuple satisfies the query, (C2) *relevant* blocks, the ones in which all the tuples satisfy the query, and (C3) *suspect* blocks, the ones in which some tuples may satisfy the query.

Obviously, the blocks in C1 can be safely skipped. The C3 blocks can also be skipped when the requested aggregates can be answered by SMAs. We note that the opportunity of identifying and skipping C3 blocks can be rather limited in practice, as it requires that 1) all the (conjunctive) filters of the query subsume the block's min and max ranges and 2) all the requested aggregates can be answerable by the chosen SMAs and if the query contains a group-by, the SMAs must be stored for all of the potentially relevant groups. A number of systems, e.g., [5, 70, 3, 68], use a simplified version of SMAs, which only skip C1 blocks and do not distinguish C2 from C3 blocks. Similar to these systems, SOP only consider skipping C1 blocks. While these previous approaches are built on top of range partitioning, our main contribution is to develop a novel fine-grained partitioning technique based on workload analysis, which can turn more blocks into C1 blocks than a range partitioning. Nevertheless, the idea of using SMAs to skip C3 blocks can also be implemented on top of our partitioning scheme.

Materialized View Selection

This chapter is related to the well studied problem of materialized view selection (MVS), since both exploit pre-computations for query performance improvement. These two problems, however, differ fundamentally in several ways. First, SOP is at the file-organization level, while MVS is at the application level; in fact, SOP can be used to partition large materialized views. Second, we utilize pre-computation to guide the tuple re-arrangement and only need to maintain minimal metadata (i.e., a bit vector per block), while MVS does not change the original data but store the precomputed results, which can incur significant space overhead, e.g., data cubes. Third, MVS is an optimization problem constrained on space [33, 56] or maintenance cost [30], while SOP is constrained on the number of partitions.

Similar to SOP, some MVS approaches also exploit workload information. Most of these focus on the group-by columns of the queries (e.g., [17, 13]) for deciding which columns to pre-aggregate. Others (e.g., [56]) also consider which columns are filtered on for selecting indexes and materialized views in an integrated manner. Different from this work, the workload analysis step in SOP aims to identify representative filters, including both filter columns and constants, and their subsumption relations for skipping purposes. We consider all kinds of filters, such as equality/range conditions, string matching and user defined functions.

Workload-driven Physical Design

Many research efforts have been devoted to utilizing workload information for automating database design. For example, the AutoAdmin project [56, 9] integrates many physical design problems, such as selecting indexes and materialized views; Database Cracking [35] reorders the data columns

as a byproduct of query processing to benefit future queries; ARF [11] tunes a range-based filter for skipping cold data; BlinkDB [57] prepares samples offline based on the workload.

Optimization Problems for Partitioning

Finding an optimal partitioning over a set of data points is an important problem in many applications, such as data mining, computer vision [62], gene expression analysis [38] and VLSI design [40]. The partitioning problem is NP-hard for many objective functions, e.g., [12]. To the best of our knowledge, no existing work has formulated the k-MaxSkip problem before, although we found k-MaxSkip and $BalancedMaxSkip$ are closely related to several partitioning problems, such as hypergraph cut [40], discrete basis partitioning problem [47] and row-exclusive biclustering [46].

3.9 Conclusion

In this chapter, we presented skipping-oriented partitioning (SOP), a fine-grained data partitioning framework that partitions the data tuples into blocks in order to help queries skip data. The key components are: (1) a workload analyzer, which generates a set of features from a query log, (2) a partitioner, which computes a partitioning scheme by solving a optimization problem, (3) a feature-based block skipping framework used in query execution. We prototyped the SOP technique on Shark, which showed that SOP can be easily implemented in the context of an existing query engine and the data flow can be executed using standard data marshalling steps, such as map and reduce.

We evaluated the effectiveness of SOP using TPC-H workload and a real-world ad-hoc workload. Our experimental results showed that SOP enabled the queries to scan 5-7x less data than traditional range-based partitioning schemes. The results also indicated that the reduction on data scan can directly translate to a reduction of query response time, for both memory- and disk-resident data.

While SOP is an effective technique, it does have limitations. In particular, SOP only considers horizontal partitioning schemes. In the next chapter, we explore how vertical partitioning can be incorporated to further enhance the effectiveness of data skipping and propose a generalized skipping-oriented partitioning framework.

Chapter 4

Generalizing SOP for Columnar Layouts

It is clear that the opportunity for data skipping highly depends on how the data are organized into blocks. While SOP can significantly outperform traditional horizontal partitioning techniques, such as range partitioning, its performance can be sensitive to the workload and data characteristics. One major constraint in the SOP design is that it only considers horizontal partitioning schemes, i.e., each tuple is an atomic unit of the partitioning. This constraint can be easily eliminated when data is stored in a column-oriented fashion, which, as described in Chapter 2, can be widely seen in analytics databases. In this chapter, we explore how vertical partitioning, or column grouping, can be incorporated into the physical layout design in order to further enhance the effectiveness of data skipping.

4.1 Introduction

Modern analytics applications can involve wide tables and complex workloads with diverse filter predicates and column-access patterns. For these kinds of workloads, SOP suffers from a high degree of *feature conflict*, which refers to the phenomenon that the best partitioning schemes for different features may be highly different. Consider the table in Figure 4.1(a). Suppose SOP extracts two features from the workload:

F_1: `grade='A'`

F_2: `year>2011`$\wedge$`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 4.1(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 a single horizontal partitioning scheme for all columns. 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

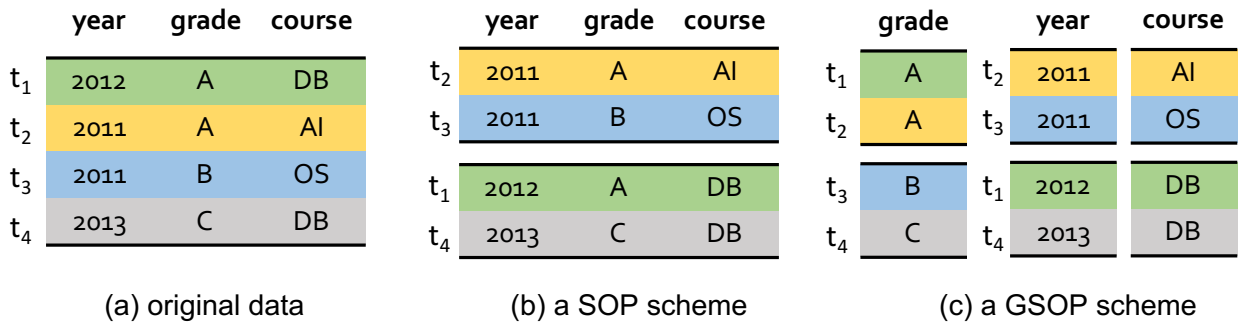


Figure 4.1: GSOP vs. Other Partitioning Schemes.

columnar layouts. Analytics systems have increasingly adopted columnar layouts [26] where each column can be stored separately. Inspired by recent work in column-oriented physical design, such as database cracking [35, 61], 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 4.1(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* [36], 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, in this chapter, we propose a Generalized SOP (GSOP) framework, with a goal of improving the overall query performance by automatically balancing the 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 goals 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., [4, 42, 10, 77]) 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 inter-

relation 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 4.4.3, 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 [44]. 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 [14], 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, in this chapter, 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.

4.2 Review of SOP

4.2.1 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 Chapter 3, 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.

4.2.2 The SOP Framework

Recall that the SOP framework is based on two interesting properties observed from real-world analytical workloads (Section 3.3):

(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 Chapter 3,s 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 4.2 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 [8]. 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 columns, as their filter values tend to change over time. In Figure 4.2, we extract three features, each of which is associated with a weight indicating the importance of the feature. Recall that SOP takes

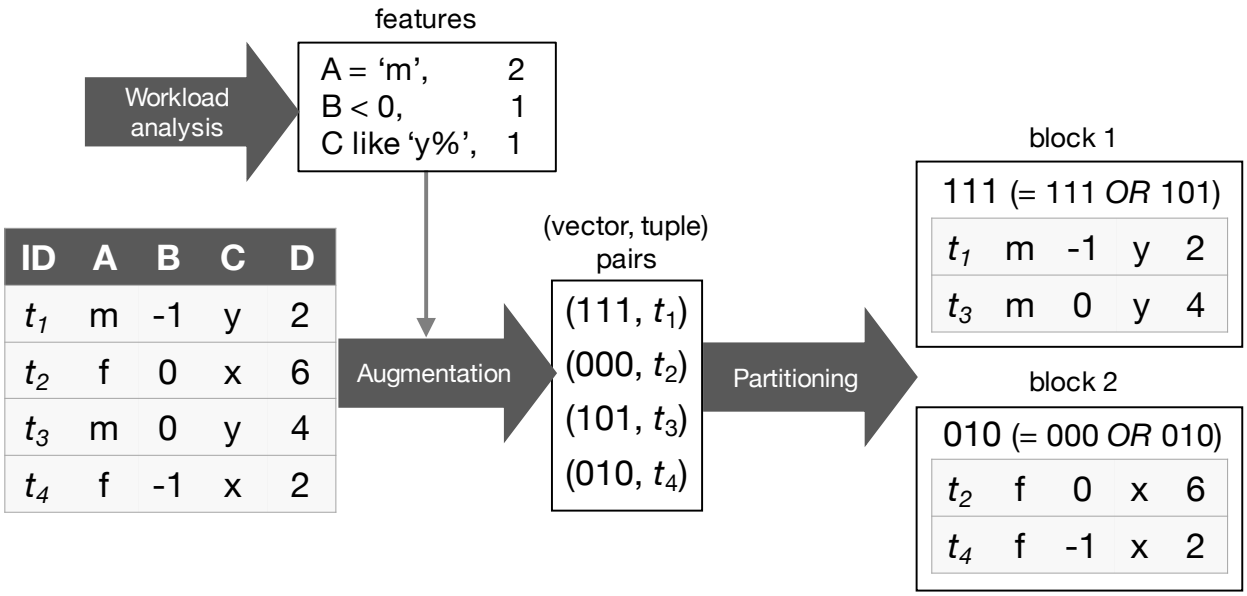


Figure 4.2: Using SOP to Partition Data into 2 Blocks.

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 4.2, 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 4.2, 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 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 4.2, 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.

Recall 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.

4.3 Generalizing SOP

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

4.3.1 A Simple Extension for Columnar Layouts

As described in Chapter 2, modern analytics databases [2, 68, 3] and the Hadoop ecosystem [14, 73] 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.

4.3.2 Spectrum of Partitioning Layouts

The existing SOP framework and the simple extension in Section 4.3.1 represent two extremes of partitioning layouts in a column-oriented storage. Let us now consider the entire spectrum as depicted in Figure 4.3. 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

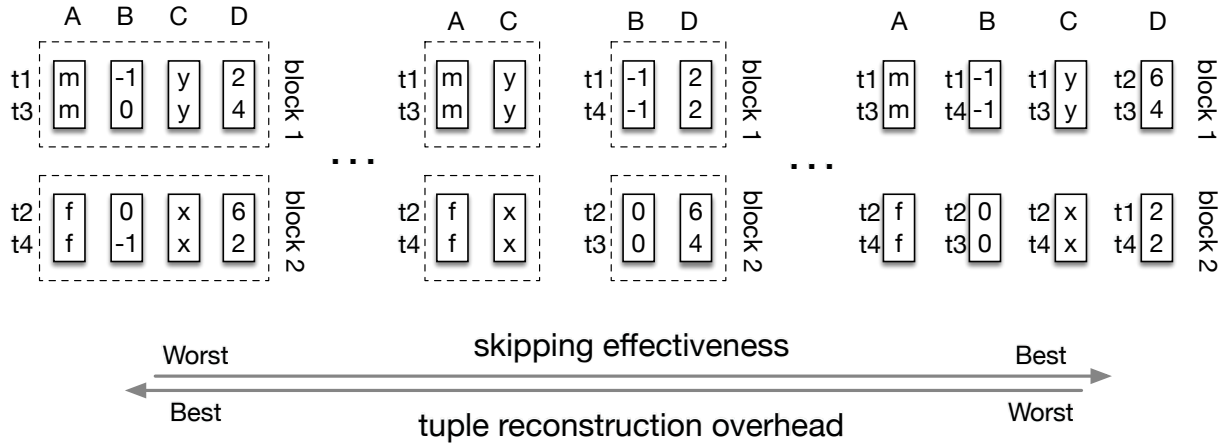


Figure 4.3: The Spectrum of Partitioning Layouts.

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 4.3.1. 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 2 *Figure 4.3 shows three different layouts of the table in Figure 4.2. 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 4.3, 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 predicates (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.*

This partitioned table divides columns into two groups: (A, C), and (B, D), and partitions each group independently. For this partitioned table, we can skip the block 2 because neither t_2 *nor* t_3 *in the block 2 satisfies the query's predicate. Furthermore, we do not need to read any tuple ids because the columns* B *and* D *are in the same group that is guaranteed to be partitioned in the same way. Thus, we only need to process four data cells in total. We summarize these calculations in the following table:*

	values read	*ids read*	*assembly*
left	*8*	*0*	*None*
right	*4*	*4*	*join B&D*
middle	*4*	*0*	*None*

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.

4.3.3 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 4.4, GSOP consists of the following steps:

1). Workload Analysis. This is the same as Step 1 in Section 4.2.2. 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 4.2.2. 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 skip-

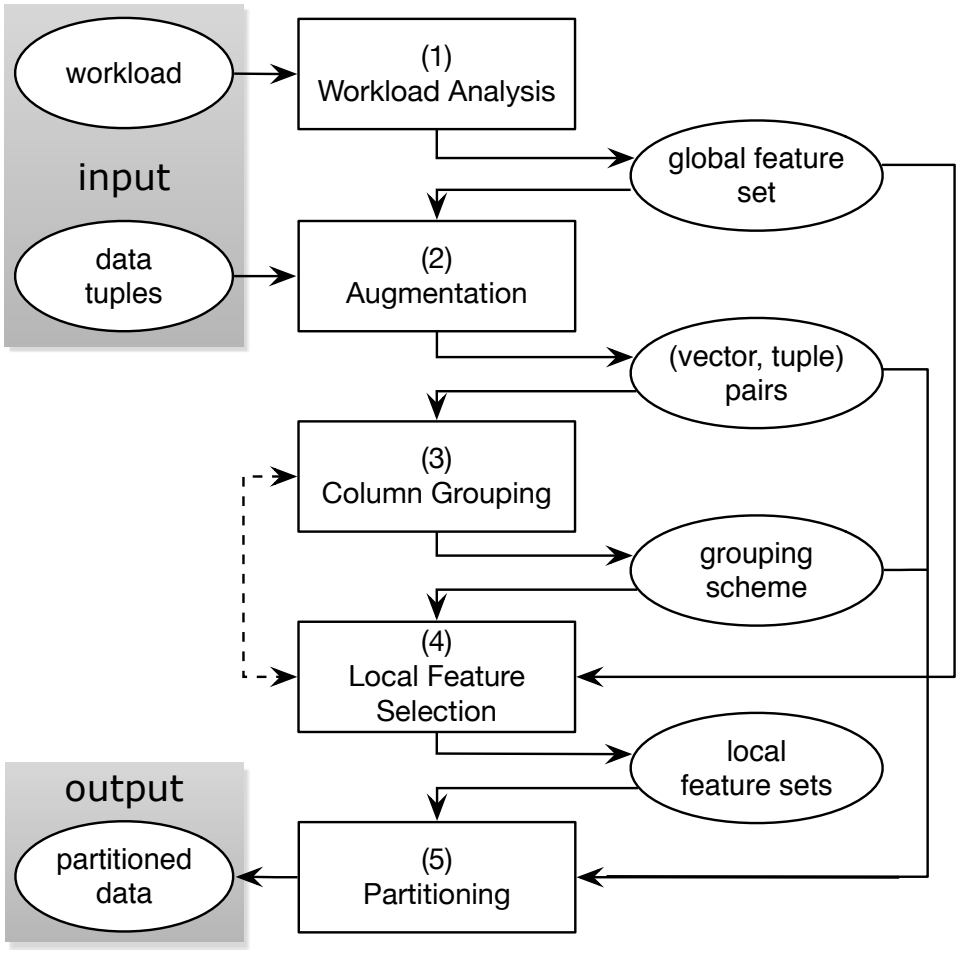


Figure 4.4: The GSOP Framework.

ping 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 4.4). 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 4.5.

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 4.2.2 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 4.4) and local feature selection (Section 4.5) work in detail.

Handling Normalized Schemas. The GSOP framework is motivated by modern analytics scenarios, where data is often stored as single denormalized tables [59, 74]. 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 4.7, 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.

4.4 Column Grouping

4.4.1 Motivation

Column grouping is an important database technique and has been extensively studied (e.g., [60, 58, 39, 10, 77, 45]). 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 [39] 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 3.

Example 3 *Consider the following workload for the table in Figure 4.2:*

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.

4.4.2 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:

$$\sum_{G_i \in \mathbb{G}^q} |G_i \cap C^q| \cdot r_i^q. \tag{4.1}$$

Equation 4.1 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 4.1. We will discuss how to obtain the value of r_i^q in Section 4.4.3.

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 $\mathrm{sort}(x)$ 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:

$$\text{overhead}(q, \mathbb{G}) = \begin{cases} \sum_{G_i \in \mathbb{G}^q} (r_i^q + \text{sort}(r_i^q)) & \text{if } |\mathbb{G}^q| > 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

Objective Function. Based on Equations 4.1 and Equation 4.2, the cost of processing query q w.r.t a column grouping scheme $\mathbb{G}$ is:

$$\text{COST}(q, \mathbb{G}) = \sum_{G_i \in \mathbb{G}^q} |G_i \cap C^q| \cdot r_i^q + \text{overhead}(q, \mathbb{G}) \tag{4.3}$$

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

$$\text{COST}(W, \mathbb{G}) = \sum_{q \in W} \text{COST}(q, \mathbb{G}) \tag{4.4}$$

We are aware that modern column-store systems employ advanced compression techniques and compression-aware execution strategies [25]. For simplicity and generality, however, our cost model does not factor in these advanced techniques. As shown in Section 4.7, 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.

4.4.3 Efficient Cost Estimation

As shown in Equation 4.3, 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 4.7 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 4.3.3), 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. In the process of searching for a good column-grouping scheme (details in Section 4.4.4), 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 sub-routine 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 4.4.3, 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. We illustrate this in Figure 4.5. 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 $\left\lfloor \frac{\mathsf{count}(v)}{b} \right\rfloor$. These blocks all have v as their *union vector*. As discussed in Section 4.2.2, 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 4.7, using our estimation approach can signicantly speed up the column grouping process without sacrificing the quality of results.

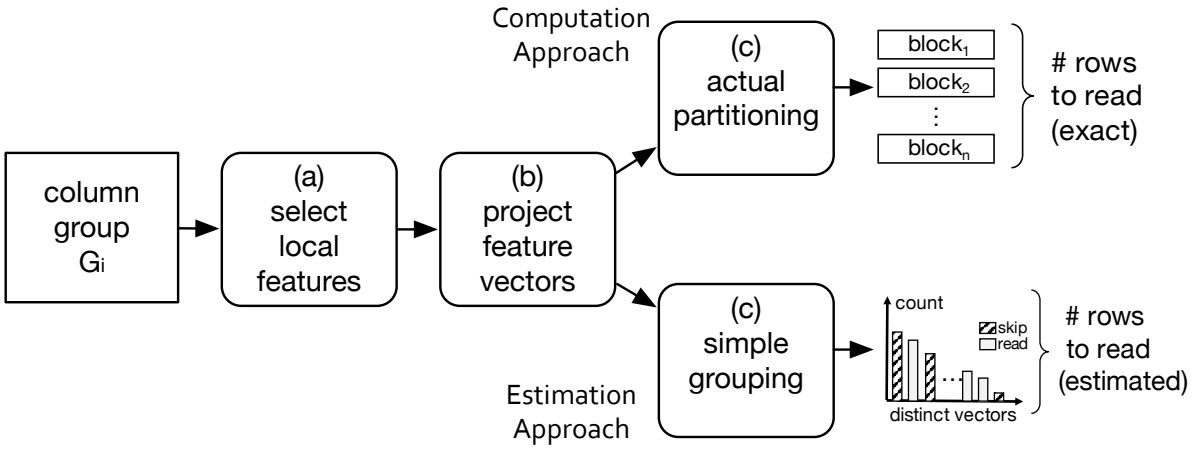


Figure 4.5: Computation Approach vs. Estimation Approach for GSOP

4.4.4 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 [39]. 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 4.4.3 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 4.4.3 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.

4.5 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 4.2 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 3 in Section 4.4.1. 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: $\mathsf{weight}(G, f) = \big|\{q \mid f \in F^q \text{ and } q \in W^G\}\big|$. 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

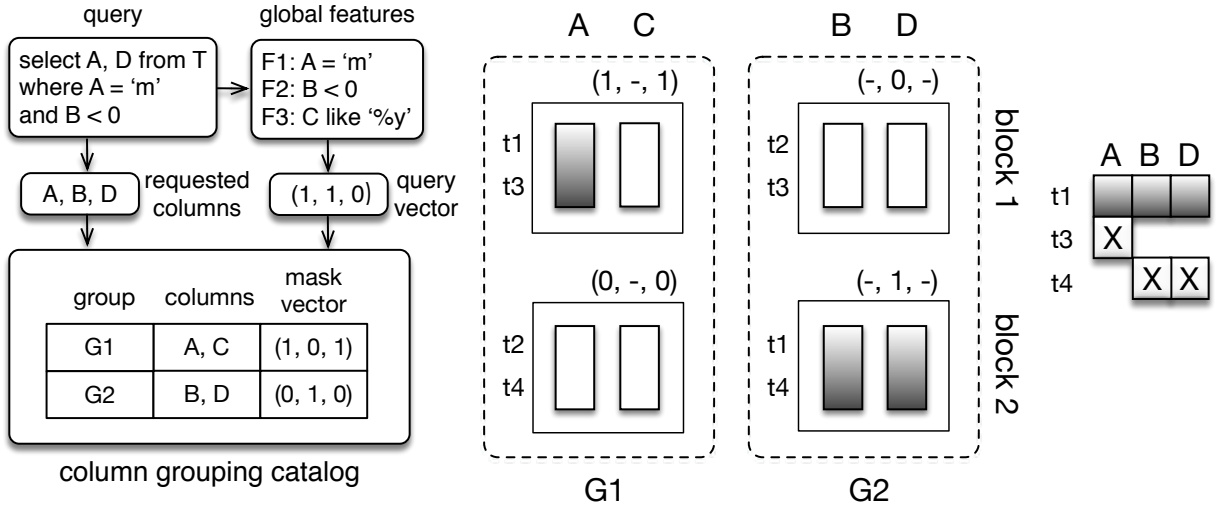


Figure 4.6: Query Processing on GSOP

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.

4.6 Query Processing

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

4.6.1 Reading Data Blocks

We first describe the process of reading data blocks. Figure 4.6 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 4.6, 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 4.6, 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 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 4.6, 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 4.6, we end up reading column A from block 1 of G_1 and columns B and D from block 2 of G_2.

4.6.2 Tuple Reconstruction

If the requested columns of the query span multiple column groups, as shown in Figure 4.6, 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 [14] 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 4.6, 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.

4.7 Experiments

In this section, we evaluate the performance of GSOP. We first explain the prototype of GSOP in Section 4.7.1. In Section 4.7.2, we discuss the three workloads and datasets used in the experiments. We report the experimental settings in 4.7.3, In the first group of experiments (Section 4.7.4), we used the dataset and the simulated scan queries in the Big Data Benchmark. The simulation of queries allows us to understand the performance of our techniques under different characteristics of workloads. In the second group of experiments (Section 4.7.5), we used the dataset as well as the queries provided by the TPC-H benchmark. The goal of this experiment is to understand the end-to-end performance of GSOP. Finally, we conduct experiments on a real-world dataset from Sload Digital Sky Survey in Section 4.7.6.

4.7.1 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 SOP. We then used SparkSQL to batch evaluate these features on a table and generate a global feature vector for each tuple. We implemented the 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.

4.7.2 Workloads

Big Data Benchmark. The Big Data Benchmark [18] 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 [67] 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 GSOP is 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 4.3.3.

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) [63] 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 [20]. 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.

4.7.3 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$\times$ 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.

4.7.4 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 4.7.2, 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 the proposed framework, SOP represents the

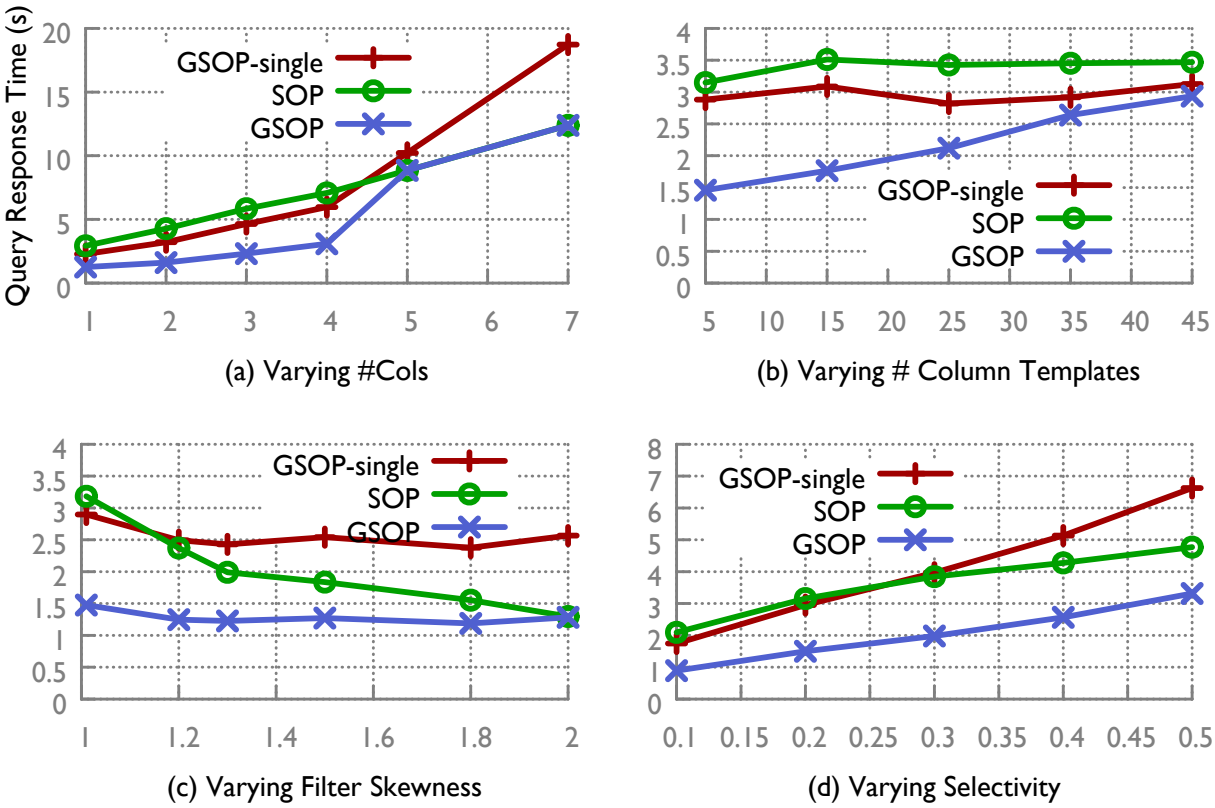


Figure 4.7: Query Performance Results of GSOP (Big Data Benchmark).

SOP framework for data skipping, and GSOP-single represents the proposed extension to SOP, as discussed in Section 4.3.1, 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 4.7(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 4.7(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 4.7(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.

In Figure 4.7(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 the objective function (Section 4.4.2) 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.

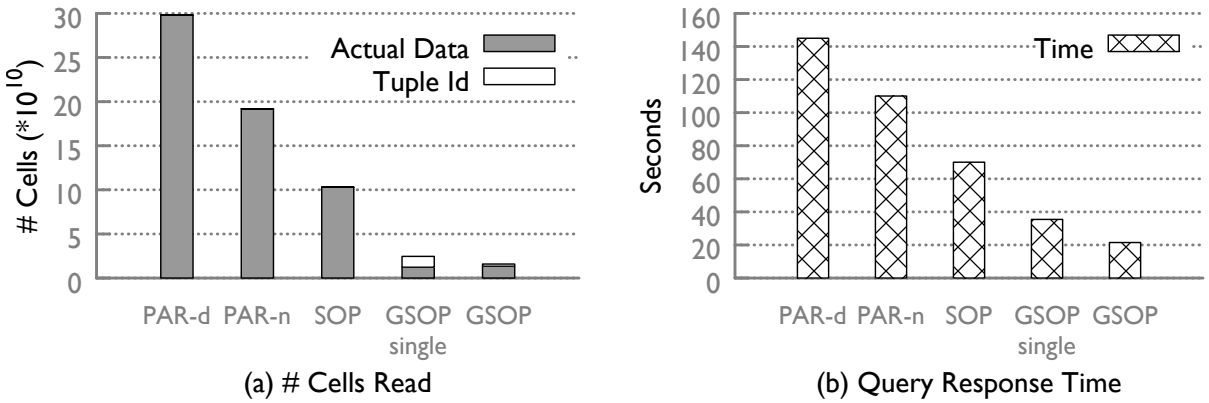
The above results showed how different workload parameters may affect the data layout design. Clearly, no single static layout is the best across different settings. Thus, we need GSOP to help us automatically choose an appropriate layout based on workload and data characteristics.

4.7.5 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 the approaches as described in Section 4.7.4. 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 4.8(a), we measure the average number of actual data cells and tuple ids read by a test query. Overall, our proposed approaches can significantly outperform the baseline approaches which rely on Parquet's built-in data skipping mechanisms. The best approach, i.e., GSOP, can reduce the data read by $20\times$ 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



(a) # Cells Read (b) Query Response Time

Figure 4.8: Query Performance Results of GSOP (TPC-H)

$14\times$ over PAR-n. Of the 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 4.8(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 4.8(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, GSOP outperforms the baselines PAR-d and PAR-n by $6.7\times$ and $5.1\times$, respectively, and outperforms SOP by $3.3\times$.

Column grouping

In Section 4.4, 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 the proposed column grouping technique with the state-of-the-art. We picked two state-of-the-art column grouping techniques, namely HillClimb [51] and Hyrise [42], as they showed superior performance in a recent experimental study [39]. In Figure 4.9, 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,

[1]Please refer to [67] for the details of the workload.

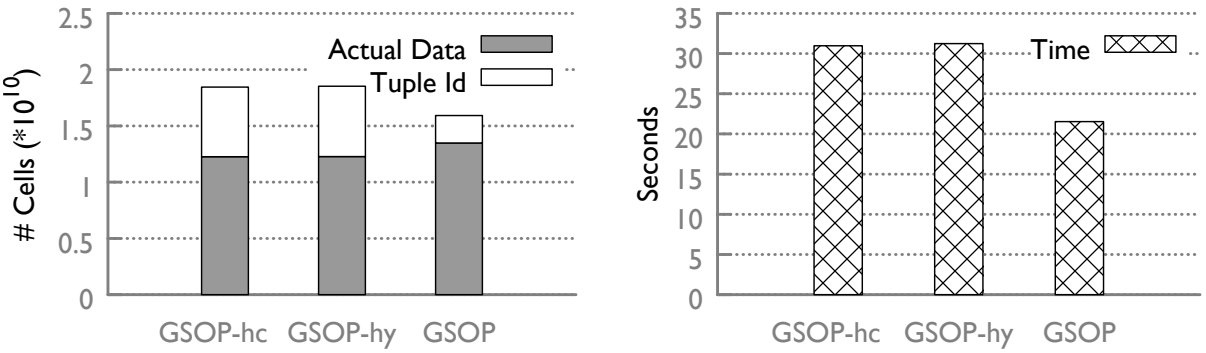


Figure 4.9: Column Grouping Performance Results on GSOP (TPC-H).

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 4.9(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 4.9(a) does not factor in the joining cost. If we look at the end-to-end query response time in Figure 4.9(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

The 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.

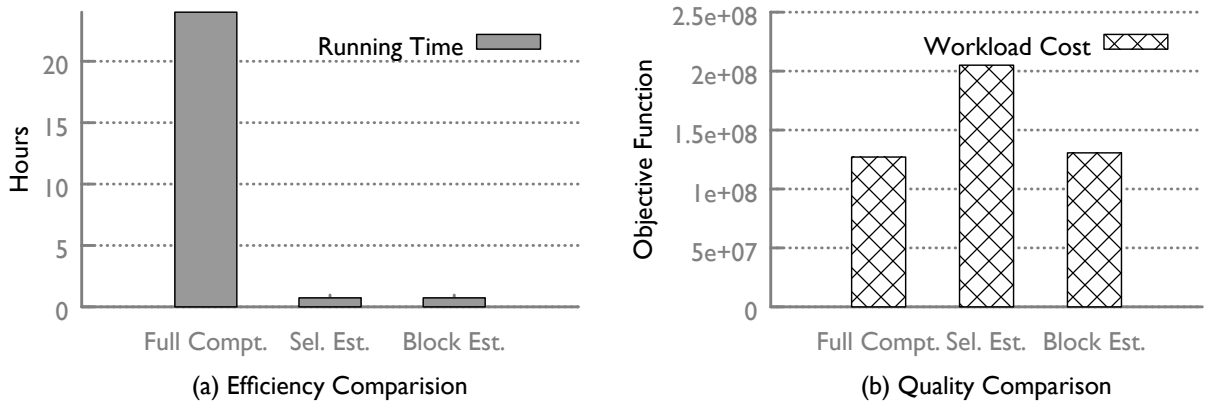


Figure 4.10: Objective Function Evaluation in GSOP (TPC-H).

Objective Function Evaluation

To quantify the goodness of a column grouping scheme, we develop an objective function in Section 4.4.2. 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 4.4.3. 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 4.10(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 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 4.10(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 4.5 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 4.11(a), the input data is a single denormalized table, and in Figure 4.11(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 4.11(a), we compare five alternative approaches 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 4.11(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.

4.7.6 SDSS

We now examine the performance of GSOP on a real-world workload. In Figure 4.12, 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.

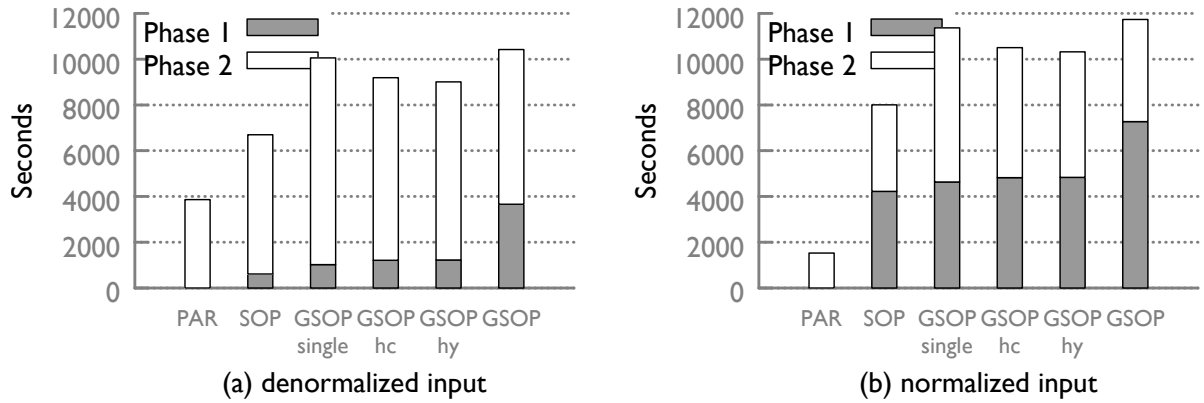


Figure 4.11: Loading Cost of GSOP (TPC-H).

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 SOP by $2.7\times$.

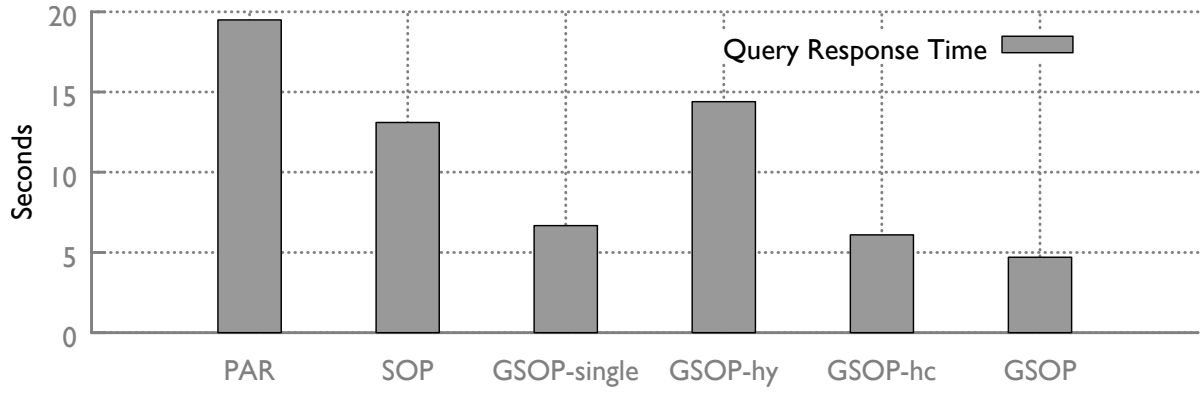


Figure 4.12: Query Performance Results of GSOP (SDSS).

4.8 Related Work

In this section, we review the related work for GSOP.

Horizontal Partitioning

Many research efforts have been devoted to workload-driven physical design (e.g., [56, 58, 60]). Such work aims for automatic physical design, such as indexes and materialized views [56], based on workload information. While workload-driven horizontal partitioning techniques have been studied [58, 15, 76, 55], 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 [24] 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 [60, 4, 58, 42, 39, 10, 77, 45, 51]. 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., [68, 2, 41, 19, 14, 73]). In a column store, columns can form column groups [43, 5, 7]. 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 [1] 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 [43, 5] 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 [35, 61, 36] 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[71] is a PAX-style layout [1] for HDFS, where data is horizontally partitioned into HDFS blocks and each block uses columnar layouts internally. ORC Files [73] and Parquet [14] also adopted the PAX-style layout with performance optimizations over RC Files. Floratou et al. [28] 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 [72] 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.

4.9 Conclusion

The design of GSOP is motivated by the observation that SOP is sensitive to feature conflict. GSOP can effectively mitigate feature conflict by allowing different columns to have different partitioning schemes. As compared to SOP, GSOP employs 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 show 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 improves the query response time by $3.3\times$ over SOP.

GSOP is a good example of boosting query performance by considering flexible data layouts. In the next chapter, we explore how replication can be introduced in the physical layout design in order to even further enhance data skipping.

Chapter 5

Extending GSOP with Replication

In this chapter, we explore how data replication can be leveraged to further reduce feature conflict and improve data skipping. The idea of using data replication to improve query performance is not new. The use of materialized views [56, 17] is a well-known example of this category. By storing the precomputed query results separately, queries can directly access the replicated query results instead of having to navigate the original data. Another line of research in this area is to make full copies of data and design the physical layout differently for each copy so that each copy is best at handling a different subset of workload [54, 4]. A major benefit of such approaches is that they can leverage the data copies that already exist for reliablity and availability purposes, e.g., Trojan [4] exploits the default 3-way replication scheme of HDFS blocks and devises a different column grouping scheme on each of the three replicas. Inspired by these ideas, we propose a framework called generalized skipping-oriented partitioning with replication (GSOP-R), which designs partitioning and replication schemes in an integrated manner.

5.1 Introduction

As described in Chapter 4, GSOP mitigates feature conflict by separating columns into groups. Such an approach works best when the features are evenly distributed across columns. In real-world analytics workloads, however, it is not uncommon that column access is highly skewed, i.e., a small number of columns are accessed by a lot of queries. In such scenarios, its likely that a few columns are *involved* in many features and thus suffer from a high degree of feature conflict. The vertical partitioning process in GSOP simply cannot affect the feature conflict coming from a single column. Suppose we extract another feature f_3: c_2>1 from the workload. In this case, column c_2 is involved in two features, namely f_2 and f_3, which happen to be conflicting. No partitioning scheme of c_2 can work for both features, and vertical partitoning will not help. Note that a column does not have to be in the filter predicate to be considered involved in a feature. For example, we can say that column c_4 is involved in feature c_2>1 if c_4 appears in a query whose predicate is c_2>1 in the workload. Thus, the frequently-accessed columns can easily be involved in a large number of features. As no good horizontal partitioning scheme can be found

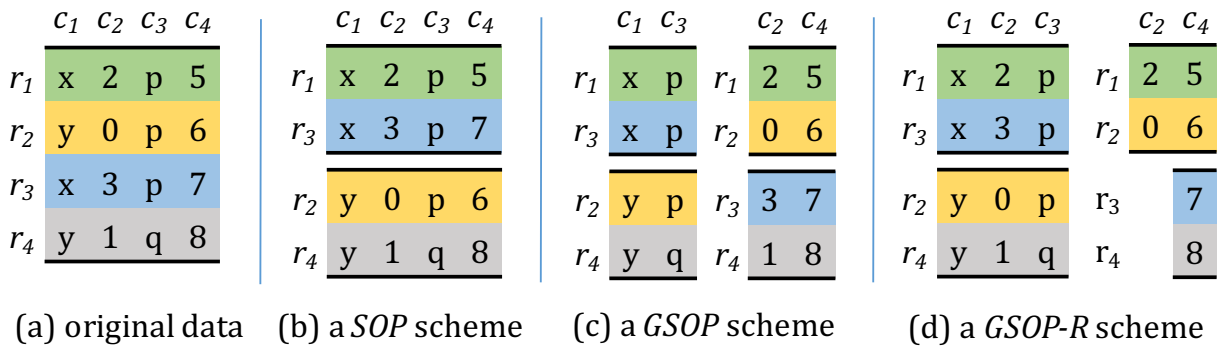


Figure 5.1: GSOP-R vs. Other Layout Schemes.

on such columns, scanning these columns will become the query cost bottleneck due to ineffective data skipping. Even worse, as these columns are frequently accessed, their scanning cost is a considerable part of the overall performance of the workload.

In this chapter, we explore how data replication can be leveraged to reduce feature conflict and improve data skipping. To improve query performance, existing techniques have considered the replication of either columns (e.g., C-Store [43] and database cracking [35]) or rows (e.g., materialized views [56]). Some work also leveraged the full data copies that exist for fault-tolerance and availiability purposes by designing a different physical layout on each data copy (e.g., fractured mirrior [54] and Trojan [4]). To directly extend GSOP with replication, we develop two basic approaches. The first approach is a *selective replication* approach, which first designs a GSOP layout scheme for the data and then picks out a set of features in order to replicate the data that satisfy these features. This approach is similar to the use of materialized views. The second approach is a *full-copy replication* approach, which makes multiple copies of the data and designs a different GSOP scheme on each copy. To do this, we first run a clustering algorithm on features with an objective of minimizing intra-cluster feature conflict. We then assign a cluster of features to each data copy and use GSOP to design the layout of each copy based on its assigned cluster of features. Both approaches, however, treat GSOP as a black box and separate replication from partitioning, which may lead to possible inefficiencies. Also, they are only limited to one form of replication, i.e., either replicating query results or making full copies, which may not make the most efficient use of space budget for the given workload and data characteristics.

To this end, we develop a new framework, termed Generalized Skipping-Oriented Partitioning and Replication (GSOP-R). In contrast to the existing techniques that replicate either rows or columns, GSOP-R supports the replication of subsets of rows and columns, i.e., neither a row nor a column is an atomic unit of replication. The goal of GSOP-R is to organize data into small blocks through replication and partitioning so that the data skipping opportunities can be increased. We define a GSOP-R scheme as a set of *layouts*, each of which contains a piece of data. A layout in GSOP-R is similar to a column group in GSOP but much more flexible. What data goes in a layout is determined by a triplet: *columns*, *local features* and *master columns*. Figure 5.1(d) shows

a GSOP-R scheme of two layouts. In general, a layout is composed of a subset of rows and a subset of columns; columns in a layout may not be of the same cardinality; some subsets of rows and columns can be found in multiple layouts; and columns within a given layout share the same horizontal partitioning scheme. GSOP-R not only generalizes SOP and GSOP, but it also subsumes existing forms of replication based exclusively on either rows or columns.

The flexibility in GSOP-R scheme design enables efficient use of space budget for boosting query performance through effective data skipping. On the other hand, it also poses great challenges in finding a good GSOP-R scheme given the huge search space. To address this challenge, we develop a search framework based on *scheme transformations*. We first construct an initial GSOP-R scheme, where each layout contains a single column and no data is replicated. We define three types of allowed transformation operations on GSOP-R schemes: `replicate`, `merge`, and `merge-replicate`. Each of these operations makes a trade-off between three costs: data scan, row reconstruction, and storage. We then go through an iterative process, where we greedily find the best transformation operation to apply on the current scheme at each iteration. The objective is to minimize the query cost, which (as in Chapter 4) incorporates the data scan cost and row-reconstruction cost, given a storage constraint. The goal is to reach a good GSOP-R scheme through successive locally-optimal scheme transformations.

Clearly, it is critical to consider how queries can effectively leverage GSOP-R schemes. Since some data are replicated across multiple layouts, a query can have different ways of retrieving the requested data. We define *retrieval paths*, which specify how a given query can correctly retrieve data in a GSOP-R scheme. We formulate the problem of finding the optimal retrieval path, i.e., the eligible path that incurs the smallest estimated query cost. We show that this problem is NP-hard by a reduction from the classic set cover problem. Since finding the optimal retrieval path is a online process as part of query optimization, it has to be executed efficiently. We adopt a 2-approximiation greedy algorithm, i.e., the path found incurs a cost at most twice the optimal. In our experiments, however, the greedy algorithm almost always finds the optimal path.

We prototyped the GSOP-R framework using Apache Spark [44] and Apache Parquet [14]. Note that, as with SOP and GSOP, the process of designing GSOP-R layout schemes is offline and happens at data load time. When a new set of records (e.g., a date partition) is being loaded, GSOP-R reorganizes its internal layout and then appends it to the table. This process does not change the layout of previously stored data. When the data is stored using Parquet, for example, GSOP-R works on each Parquet file individually. We conduct experiments on TPC-H and a real-world workload. Our results show that GSOP-R can reduce the query response time by 2x over GSOP with only 20% storage overhead.

5.2 Review of GSOP

Since GSOP-R is based on GSOP, we first review the steps of the GSOP framework proposed in Chapter 4 using Figure 5.2 as an example.

1) Workload Analysis

Given a workload, such as a log of queries or query templates, GSOP extracts as features a set of frequently-occurred filter predicates. Each feature is a filter predicate associated with a weight indicating how many queries it *subsumes* in the workload. The notion of subsumption is important, as a feature can help a query skip data as long as the feature subsumes, or is more general than, the query predicate, even though they are not an exact match. For example, $f_2 : c_2 > 1$ in Figure 5.2 can be used to skip data for queries with predicate $c_2 > 2$. These features provide a succinct summary of the workload and will be used to guide the data partitioning. For each feature, GSOP goes back to the query log, identifies all the queries subsumed by this feature, and collects all the columns that appear in these queries as *relevant columns* for this feature. In Figure 5.2, the feature information is summarized in a table in the upper-left corner.

2) Augmentation

As data is being loaded, we batch evaluate the feature predicates on-the-fly and augment each row with a bit vector. Given m features, each vector has m bits, the i-th bit of which indicates whether this row satisfies the i-th feature. In Figure 5.2, the bit vector for r_2 is 001, which tells us that r_2 only satisfies feature f_3 but not f_1 or f_2. Since the partitioning algorithm later is only concerned with the bit vectors, not the actual data, GSOP then groups the (vector, row-id)-pairs by the vectors and generates a set of (vector, count)-pairs, where *count* is the number of rows augmented with this vector. This *group-by* step is an important optimization, as it significantly reduces the input size of the partitioning algorithm. We omit the illustration of this step in Figure 5.2 for simplicity.

3) Column Grouping

Recall that this step is the main distinction between GSOP and SOP. In SOP, all columns are horizontally partitioned together. GSOP, on the other hand, first vertically partitions the columns into groups, and then allows each column group to have its own horizontal partitioning scheme. This flexibility enables better data skipping, but also introduces overhead for row reconstruction. GSOP searches a column grouping scheme based on an objective function that balances the skipping effectiveness and row-reconstruction overhead. After the column groups are generated, GSOP identifies a set of local features for each column group, which is a subset of the global features extracted from Step 1. A feature can be selected as a local feature for a column group if the column group contains at least one of its relevant columns. For example, the local features of column group $G_1 : \{c_1, c_3\}$ are f_1 and f_2, but not f_3, because G_1 does not contain any relevant column of f_3, namely c_2 or c_4.

4) Partitioning

In this step, GSOP horizontally partitions each column group individually. Based on the local features of a column group, GSOP projects the global vectors from Step 2 onto local vectors by

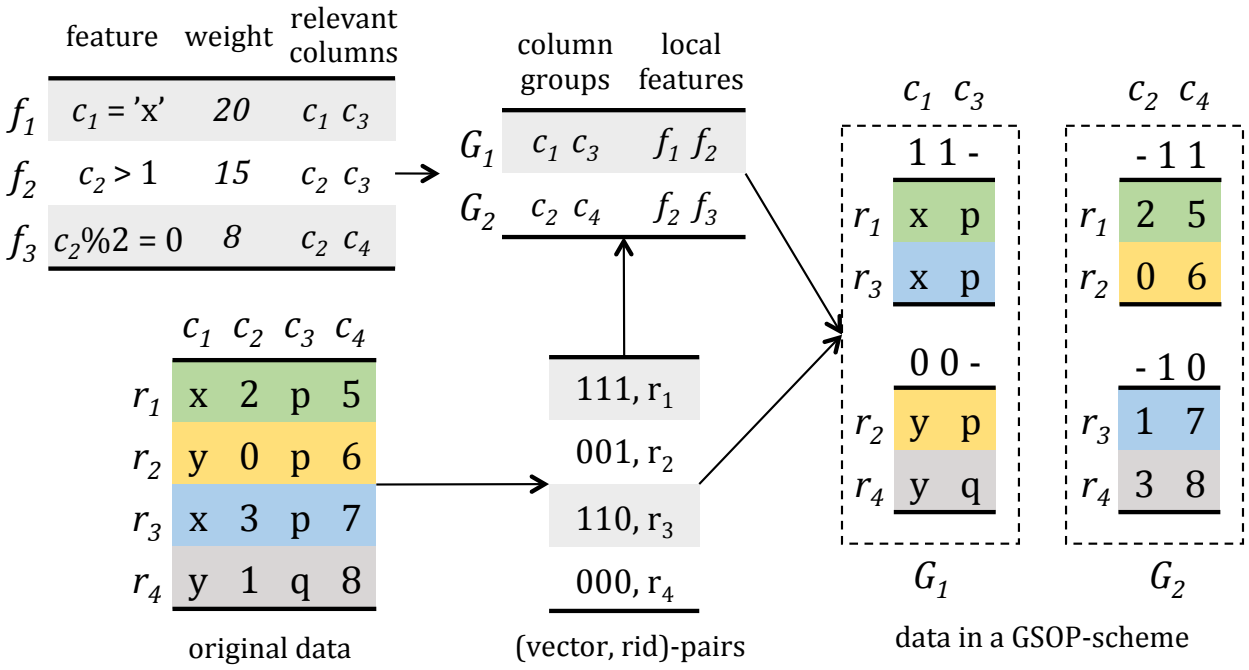


Figure 5.2: A GSOP Example.

keeping only the bits corresponding to the local features, and then performs a clustering algorithm to generate a horizontal partitioning scheme based on these local vectors. For example, since the local features of column group G_1 are f_1 and f_2, GSOP only looks at the first 2 bits of the vectors while running the clustering algorithm; similarly, it ignores the first bit while running the algorithm for G_2. Thus, the two column groups can have different horizontal partitioning schemes. At the end, the data are loaded into the GSOP layout scheme as guided by both the column grouping scheme and the horizontal partitioning scheme of each column group. In Figure 5.2, each column group is horizontally partitioned into two *blocks*. We store a *union vector* as metadata for each block. The union vector of a block is simply a union of the local vectors of all rows in this block. For example, the union vector of the lower block in G_1 is $(-, 1, 0)$. The first bit is an *invalid bit*, which indicates that f_1 is not a local feature and thus should not be used for skipping. The second bit 1 says that the block may contain rows that satisfy feature f_2.The third bit 0 informs us that no row in this block satisfies f_3, and thus any query subsumed by f_3 can safely skip this block. The union vectors will be used by future queries as metadata to skip data blocks. Note that we only need to keep the union vectors and can drop the vectors augmented with individual rows after data is loaded.

Query Processing

Suppose the following query is issued on the data in the GSOP scheme in Figure 5.2:

$$\texttt{SELECT c}_1\texttt{, c}_2\texttt{ FROM T WHERE c}_2\%2 = 0$$

We first check if any feature subsumes this query and find that f_3 does. Given the column grouping scheme, the query needs to read to column c_1 from G_1 and column c_2 from G_2. Since f_3 is not a local feature of G_1, the query cannot skip data based on f_3 in G_1. Note that each block adopts a columnar layout internally. Thus, the query only reads column c_1, not c_3, from both blocks of G_1, 4 values in total. In order to reconstruct the values from column c_1 with that from c_2 later, the query also has to retrieve a row id for each value read, 4 row ids in total. In group G_2, f_3 is a local feature. The query checks the union vectors of both blocks and decides that the lower block can be skipped, as the third bit of its union vector is 0. Then the query reads column c_2 from the upper block, namely, 2 and 0, and their corresponding row ids, i.e., r_1 and r_2. Finally, the query assembles the two columns together based on row ids and only keeps the rows where both column values are present, i.e., rows $r_1 : (p, 2)$ and $r_2 : (p, 0)$.

Having reviewed the steps of GSOP, we next discuss how to incorporate replication into the GSOP framework.

5.3 Basic Replication Approaches

We explore two basic approaches that extend GSOP with data replication.

5.3.1 Selective Replication

The idea of selective replication is inspired by the classic database technique of materialized views, where we can pre-evaluate frequently-issued queries offline and store the results. In the future, queries can directly go to the materialized views instead of having to access the original data.

In selective replication, we first use GSOP to partition the data and see which features can benefit from replication the most. As mentioned in Section 5.2, a feature can represent a set of frequently-occurring queries. Each feature has a predicate, which qualifies a set of rows, and is associated with a set of relevant columns. We can replicate these rows and columns, so that all queries subsumed by this feature can directly scan the (smaller) replicated data instead of the original data. Consider the GSOP scheme in Figure 5.2. The horizontal partitioning scheme of column group G_2 is based on two local features f_2 and f_3, which works out well for f_3 but not f_2. As we can see, the union vectors of the two blocks in G_2 both have a 1 on their second bit, which indicates that queries subsumed by f_2 cannot skip any block. Clearly, feature f_2 can benefit from replication. In Figure 5.3(a), we replicate the data requested by feature f_2 and store them separately as $Repl(f_2)$. The replication $Repl(f_2)$ contains all the rows that satisfy the f_2 predicate $c_2 > 1$, namely, r_1 and r_3, and all its requested columns, namely, c_2 and c_3. This way any query subsumed by f_2 only needs to scan the one block in $Repl(f_2)$ but not G_1 or G_2.

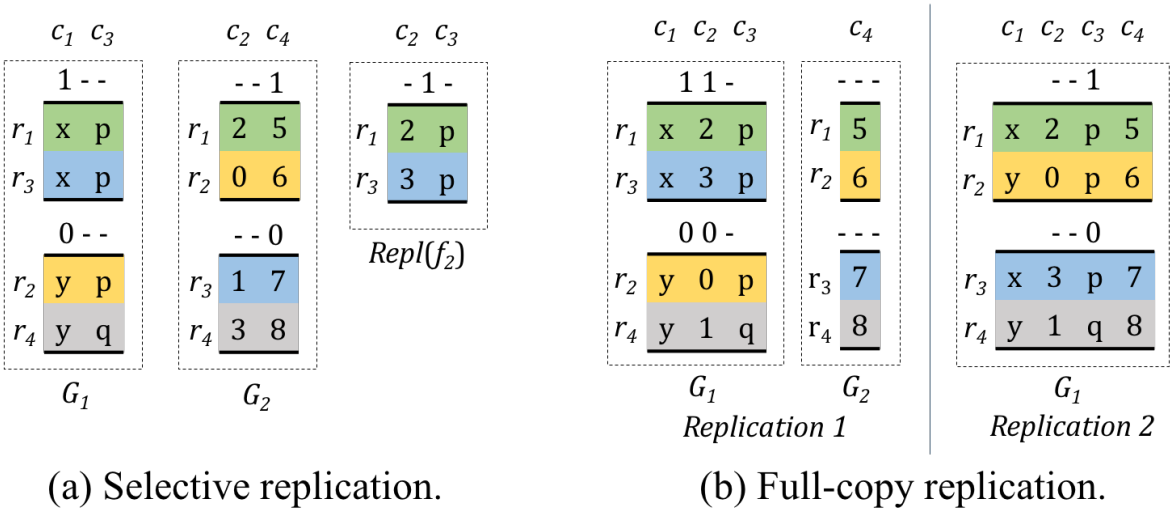


Figure 5.3: Example of Basic Replication Approaches.

Note that the creation of $Repl(f_2)$ can benefit not just f_2 but also the other features. Since the data that satisfy f_2 can be found in $Repl(f_2)$, we no longer need to keep f_2 as a local feature in G_1 or G_2 (as in Figure 5.2). Removing f_2 reduces the feature conflict in G_1 and G_2, because each of them is now concerned with one less feature. After creating $Repl(f_2)$, as shown in Figure 5.3(a), the second bits of the union vectors in G_1 and G_2 are now marked as invalid. Thus, we should re-partition G_1 and G_2 so that their new partitioning schemes are tailored for the remaining features.

Given a space budget, we need to choose a set of features like f_2 for replication that can offer the best query performance boost. We adopt a greedy approach to pick one feature at a time. At each iteration, we pick the best feature for replication and then update the horizontal partitioning schemes of the column groups that had this feature as a local feature. We keep doing this until the space budget runs out. The cost models for query performance and space overhead are defined in Section 5.4.

5.3.2 Full-Copy Replication

Replicating full copies of data has been an important mechanism to improve fault-tolerance, availability and locality in analytics systems. For example, Hadoop file system (HDFS) stores 3 copies of each HDFS block by default. Existing work [54, 4] proposed to leverage these full-copy replications for improving query performance. By designing a different physical layout for different data copies, each copy is best suited to answer a different class of queries. This can easily outperform a monolithic physical layout designed to benefit all types of queries. Such an idea can be borrowed to improve the performance of GSOP. Given m full copies of data, we first cluster the features into m groups and design a GSOP scheme for the i-th copy based on the i-th cluster of

features. As shown in Figure 5.3(b), we assign features f_1, f_2 to the first copy and f_3 to the second copy. Each copy ends up having a different column grouping and horizontal partitioning scheme. Since the GSOP scheme for each copy is based on a cluster of features instead of all features, it can be more effective for data skipping due to reduced feature conflict. To cluster the features, we can simply adopt a bottom-up clustering approach. Each feature starts as a cluster by itself. Iteratively, we find the best pair of clusters to merge based on the query cost model developed in Section 5.4. Each merge reduces the number of clusters by 1 and the iteration stops when we have m clusters.

5.4 GSOP-R Layout Schemes

In this section, we introduce the definition of GSOP-R schemes and explain how they can be applied on data as it is loaded into the database. We also discuss how to evaluate the goodness of a GSOP-R scheme.

5.4.1 Defining a GSOP-R Scheme

A GSOP-R layout scheme L consists of a set of layouts, i.e., $L = \{L_1, L_2, \ldots, L_k\}$. Let C be the set of columns in the table and let F be the set of features extracted from the workload as in GSOP (Section 5.2). We define each layout $L_i \in L$ by a triplet (C_i, F_i, M_i), where $C_i \subseteq C$ is the set of columns in L_i, $F_i \subseteq F$ is the set of local features in L_i, and $M_i \subseteq C_i$ is a set of *master columns*. In a layout L_i, only the master columns in M_i are guaranteed to be *full* columns, which contain all the rows. What rows the non-master columns (i.e., $C_i - M_i$) have are determined by the local features F_i, which also governs the horizontal partitioning scheme of L_i. We will elaborate on this distinction in Section 5.4.2. Since data replication is allowed in GSOP-R, a column may appear in multiple layouts, i.e., there can be overlaps between C_i and C_j for $i \neq j$. To ensure that a GSOP-R scheme is *complete*, i.e., there is no data loss when GSOP-R is applied, we require that every column in C must be a master column in at least one layout in L. Figure 5.4(a) shows an example GSOP-R layout scheme with two layouts. for the example table shown in Figure 5.1(a).

5.4.2 Applying a GSOP-R scheme

When a new set of rows, e.g., a date partition, is submitted to the system, we apply a GSOP-R scheme to reorganize its layout and append it to the table. We now show how to apply a GSOP-R scheme on the incoming data.

Let R be the set of rows in the incoming partition. As before, we refer to the intersection of a row and a column as a *data cell*. Given a GSOP-R scheme and the incoming data, we first need to determine what data cells each layout contains. While the column set in a layout L_i is explicitly specified by C_i, the row set in L_i, denoted by R_i, needs to be inferred from the local features F_i. Since each feature $f_j \in F$ has a filter-predicate, it qualifies a subset of rows from R. We let R_i be set of rows that satisfy at least one feature in F_i. Suppose we are now applying

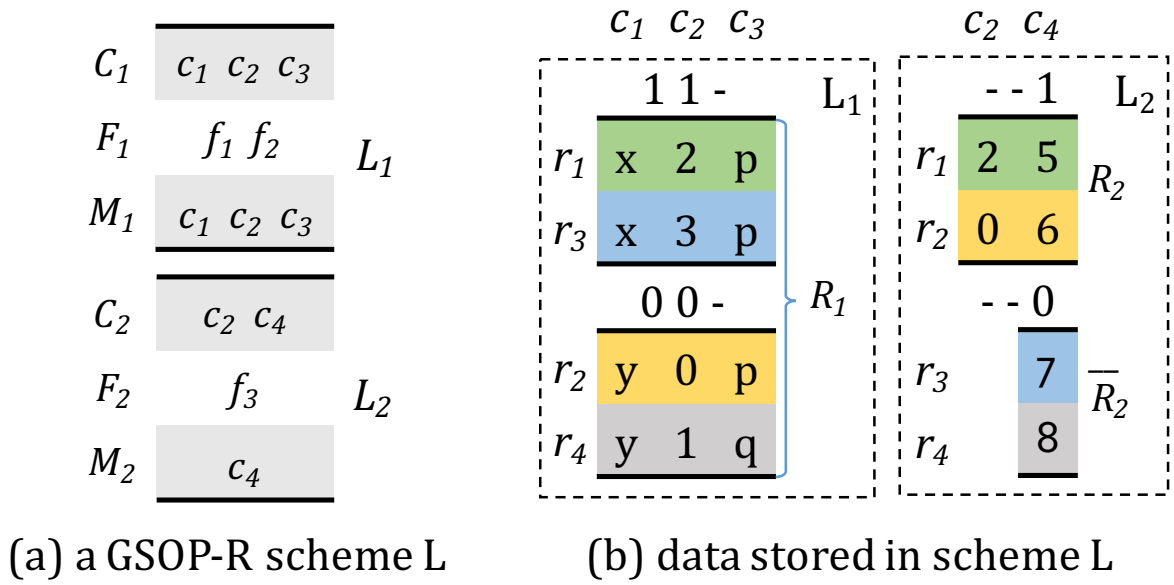


Figure 5.4: Example of a GSOP-R Layout Scheme.

the GSOP-R scheme in Figure 5.4(a) on the data in Figure 5.1(a) with the features f_1, f_2 and f_3 in Figure 5.2. The result is shown is in Figure 5.4(b). Given the local features $F_1 : \{f_1, f_2\}$ in layout L_1, we can infer that $R_1 = \{r_1, r_2, r_3, r_4\}$, as rows r_1, r_3 satisfy f_1: c_1='x' and r_2, r_4 satisfy $f_2 : c_2$>1. Similarly, $R_2 = \{r_1, r_2\}$, as $f_3 : c$%2=0 qualifies r_1, r_2 but not r_3, r_4. By definition, the master columns in each M_i are required to have all the rows from R. Thus, apart from the rows in R_i, each master column in a layout L_i also needs to include the data cells from the remaining rows, denoted by $\overline{R_i} = R - R_i$. In Figure 5.4(b), $\overline{R_1} = \emptyset$ since R_1 already contains all 4 rows; in contrast, $\overline{R_2} = \{r_3, r_4\}$, and thus we need to add these rows to the only master column in M_2, namely c_4. This way the data in Figure 5.4(b) is guaranteed to have every cell of the original data in Figure 5.1(a).

A GSOP-R layout scheme not only describes what data cells each layout contains, but also governs how the data cells in each layout are physically stored. The data cells in each layout L_i are logically stored as two separate parts. The first part contains the data cells from the rows R_i and the columns in C_i. This part is treated in the same way as a column group in GSOP, which adopts a column-oriented storage scheme and is horizontally partitioned using the clustering algorithm in GSOP based on the features in F_i. The second part contains only the cells from the master columns M_i and the remaining rows $\overline{R_i}$. This part also adopts a column-oriented scheme but is not horizontally partitioned based on the features, as the rows $\overline{R_i}$ do not satisfy any feature in F_i. For example, in Figure 5.4(b), the first part of L_1, composed of rows R_1 and columns C_1, is horizontally partitioned into two blocks. Each block is annotated with a union vector, as described in Section 5.2. The second part of L_1 is empty, and thus is not shown. There is no distinction

in the physical storage of these two parts; both parts are physically stored as a set of blocks. In layout L_2, the first part, composed of rows $R_2 = \{r_1, r_2\}$ and columns $C_2 = \{c_2, c_4\}$, is stored as a single block, and the second part is a single block of two rows in $\overline{R_2} = \{r_3, r_4\}$ and a master column c_4. A query does not need to treat the blocks in $\overline{R_i}$ differently from those in R_i, as it only needs to read the union vector and decide whether a block is to be scanned or skipped. We will discuss how queries are processed on the data stored in a GSOP-R scheme in Section 5.6.

We summarize our notation in Table 5.1.

Table 5.1: Summary of Notation

Notation	Meaning
C	set of columns in the table
R	set of rows to apply GSOP-R on, e.g., a date partition
F	set of all features extracted from the workload
L	a GSOP-R layout scheme, $L = \{L_1, L_2, \ldots, L_k\}$
L_i	a layout in a GSOP-R scheme L, defined as (C_i, F_i, M_i)
C_i	set of columns in layout L_i, $C_i \subseteq C$
F_i	set of features in layout L_i, $F_i \subseteq F$
M_i	set of master columns in layout L_i ($M_i \subseteq C_i$)
R_i	set of rows in layout L_i (inferred from F_i), $R_i \subseteq R$
$\overline{R_i}$	R - R_i
C^q	set of columns requested by query q, $C^q \subseteq C$

5.4.3 Evaluating a GSOP-R Scheme

We evaluate the goodness of a GSOP-R layout scheme based on both query performance and storage overhead.

Given a query q issued on the data stored in a GSOP-R layout scheme L, as in SOP and GSOP, we define the cost of query q as the number of data cells it scans. We now discuss how to estimate the number of data cells a query q needs to scan from the data stored in L. Since GSOP-R schemes allow data replication, a query can go to multiple places to retrieve the data it needs. Thus, a query can have multiple *retrieval paths*, each of which specifies what columns the query reads from each layout. Suppose the set of columns query q needs to read is $C^q \subseteq C$. We represent the retrival path using a binary function $\mathtt{p}(c_i, L_j)$, where $\mathtt{p}(c_i, L_j)$ evaluates to 1 if the query will read column c_i from layout L_j and 0 otherwise. For each column c_i in C^q, there is one and only one layout L_j in L such that $\mathtt{p}(c_i, L_j) = 1$. We defer the discussion of finding the optimal retrieval path to Section 5.6. For now, we assume that a retrieval path $\mathtt{p}$ is given. For each layout L_j in L, we denote by $\mathtt{rr}(q, L_j)$ the number of rows to be read by query q in L_j. The value of $\mathtt{rr}(q, L_j)$ is determined by how many blocks can be skipped based on the horizontal partitioning scheme of L_j and the query predicate. Thus, $\mathtt{rr}(q, L_j)$ is the same for all columns read from L_j. We describe how to derive $\mathtt{rr}(q, L_j)$ using metadata in Section 5.6. When query q

reads at least one column from a layout, it also needs to read row ids so that the columns from this layout can be reconstructed with those from other layouts. The number of row ids that needs to be read from L_j equals the number of rows read from it, i.e., $\texttt{rr}(q, L_j)$. To sum up, given a query q, a layout scheme L and a retrieval path p, the total number of data cells and row ids scanned can be calculated as:

$$\texttt{q-cost}(q, L, R, \texttt{p}) = \sum_{L_j \in L} \texttt{rr}(q, L_j)(\bigvee_{c_i \in C^q} \texttt{p}(c_i, L_j) + \sum_{c_i \in C^q} \texttt{p}(c_i, L_j)) \tag{5.1}$$

To evaluate the query performance of a GSOP-R scheme L with respect to the entire workload, we simply take the sum of the cost (Equation 5.1) for all queries in the workload .

We quantify the storage cost of a GSOP-R layout scheme L as the ratio of the number of data cells stored in L to the number of data cells in the original data. For each layout $L_i \in L$, the number of data cells L_i contains is $|R_i| \times |C_i| + |\overline{R_i}| \times |M_i|$. The number of data cells in the original data is $|R| \times |C|$. Thus, we have:

$$\texttt{s-cost}(L, R) = \sum_{L_i \in L} (|R_i| \times |C_i| + |\overline{R_i}| \times |M_i|)/(|R| \times |C|) \tag{5.2}$$

For example, the GSOP-R scheme in Figure 5.4(b) contains 18 data cells, i.e., 12 from L_1 and 6 from L_2, while the original data in Figure 5.1(a) has 16 data cells. Hence the storage cost for this layout scheme is $18/16 = 1.125$.

We realize that our cost model mainly uses the number of data cells as an estimation of the query cost and storage cost, which omits many of the implementation details involved in a real-world setting. For example, the data cells can be in different types and also can be compressed in different ways, which affects the actual scanning and storage costs. The goal of our cost model, however, is not to model these costs as accurately as possible, but rather to serve as a guideline to search for GSOP-R layout schemes for a given dataset and workload. Our experimental results (Section 5.7) show that this cost model works well in terms of guiding the search for GSOP-R layout schemes.

Given the models for query and storage cost, we set our goal as finding a GSOP-R scheme whose query cost is the smallest w.r.t. the entire workload given a storage cost budget.

5.5 Searching GSOP-R Schemes

In the search of GSOP-R schemes, we first construct an initial layout scheme and then go through an iterative process to *transform* the layout scheme using our defined transformation operations. At each iteration, we examine all allowed transformation operations on the current scheme and greedily pick the best operation, i.e., the one that can result in a GSOP-R scheme with the best query performance under the space budget. The goal is to reach a good GSOP-R scheme through successive locally-optimal transformations.

During the search process, we evaluate a GSOP-R scheme using the cost models discussed in Section 5.4.3 based on a sample of data. Once a GSOP-R scheme is chosen, it can be used

repeatedly for the incoming data partitions and only needs to be refreshed periodically or when there is a dramatic change to data characteristics or workload patterns.

5.5.1 Scheme Transformation Operations

Initial GSOP-R Scheme. In the initial layout scheme L, we create a layout for each column in the table. Every column is a master column in its own layout. Thus, no data is replicated in the initial scheme. To determine the local features F_i of each layout L_i, we refer to the feature information, e.g., the upper left table in Figure 5.2, and obtain the set of relevant columns for each feature f_i in F, denoted by $\texttt{rel-cols}(f_i)$. We consider feature f_i as a local feature of L_j if and only if at least one column in L_j is a relevant column of f_i, i.e., $f_i \in F_j$ if and only if $C_j \cap \texttt{rel-cols}(f_i) \neq \emptyset$. In Figure 5.5(a), we illustrate the initial layout scheme L for the data in Figure 5.1(a), where the local features of each layout are listed at the bottom. For instance, the local features of L_2 are f_2, f_3, as column c_2 is a relevant column of both f_2 and f_3.

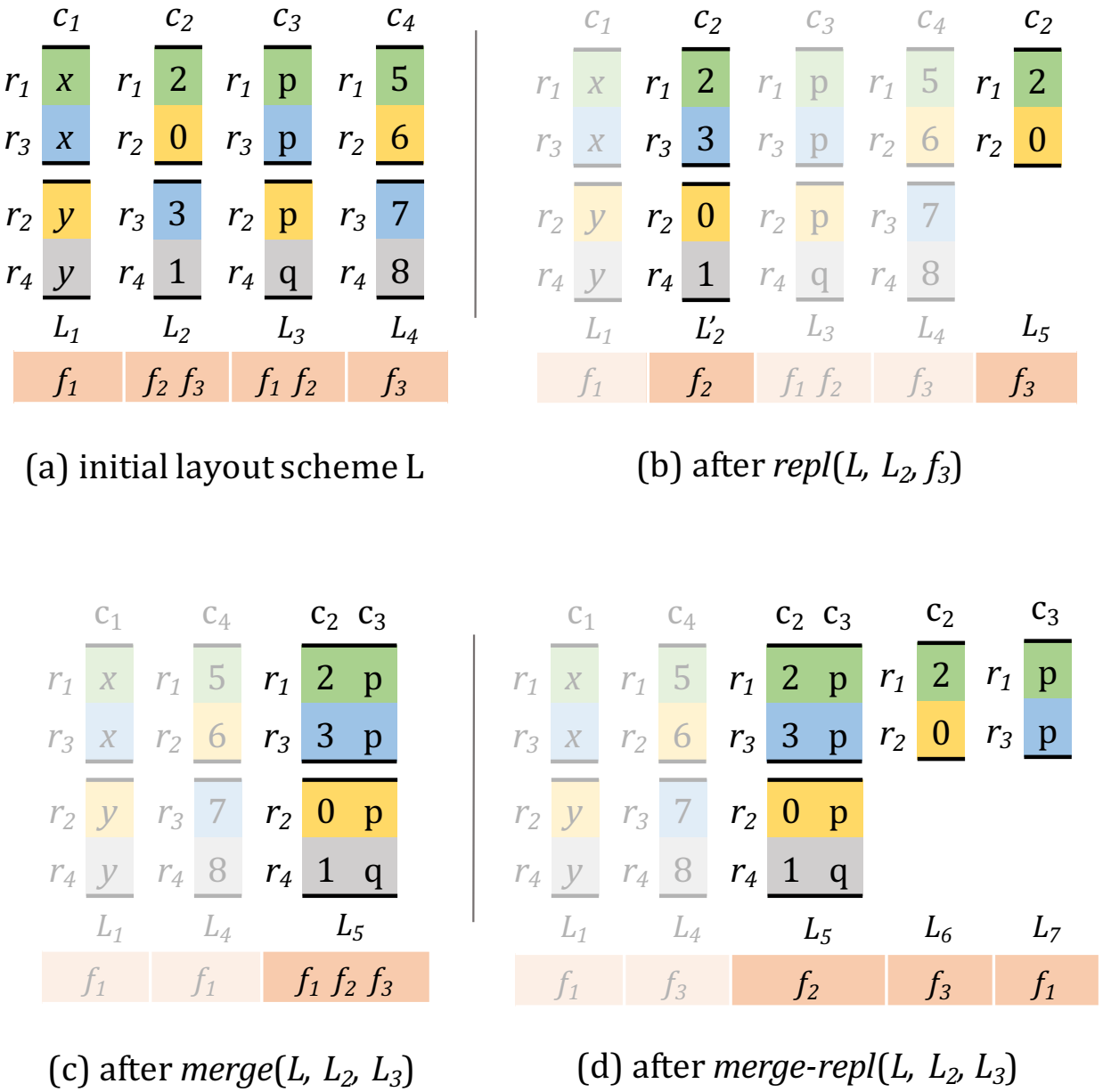
We define three types of transformation operations on a GSOP-R scheme, each of which makes a trade-off between three costs: *data scan*, *row-reconstruction*, and *storage*. An operation is considered the best for the current GSOP-R scheme if it can result in a scheme with the lowest query cost (using Equation 5.1 in Section 5.4.3), which factors in both data scan and row-reconstruction costs, under a given storage cost budget (using Equation 5.2 in Section 5.4.3). Note that these operations are only focused on the columns and local features of the layouts. Given the columns C_i and local features F_i of each layout L_i, we discuss an approach to specify the master columns M_i automatically in Section 5.5.3.

1) $\texttt{repl}(\texttt{L}, \texttt{L}_\texttt{i}, \texttt{f}_\texttt{j})$: This `replicate` operation creats a new layout by replicating the data cells requested by feature f_j from layout L_i. It creates a new layout L_{k+1}, where $C_{k+1} = \texttt{rel-cols}(f_j) \cap C_i$ and $F_{k+1} = \{f_j\}$. This operation also removes feature f_j from the local feature set of L_i, and thus updates L_i to L_i', where $F_i' = F_i - \{f_j\}$ and $C_i' = C_i$. To summarize, we have:

$$\texttt{repl}(L, L_i, f_j) = L - L_i + L_i' + L_{k+1}$$

Figure 5.5(b) shows the resulting scheme after applying $\texttt{repl}(\texttt{L}, \texttt{L}_2, \texttt{f}_3)$ to the initial scheme L in Figure 5.5(a), where the unchanged layouts are shadowed. This operation adds a new layout L_5, where $C_5 = \{c_2\}$ and $F_5 = \{f_3\}$. In the updated layout L_2', f_3 is no longer a local feature and $F_2' = \{f_2\}$. Note that this also causes L_2' to have a different horizontal partitioning scheme from L_2.

The goal of this operation is to mitigate feature conflict by replicating the data cells requested by a feature. In the initial layout L in Figure 5.5(a), the local features of L_2, i.e., f_2 and f_3, are conflicting. No horizontal partitoning scheme of L_2 can work for both features. Layout L_2 ends up with a partitioning scheme $\{\{r_1, r_2\}, \{r_3, r_4\}\}$ that only benefits f_3 but not f_2. The operation $\texttt{repl}(\texttt{L}, \texttt{L}_2, \texttt{f}_3)$ effectively eliminates such a conflict. The queries subsumed by f_3 can now retrieve column c_2 from the new layout L_5. In the updated layout L_2', f_3 is not a local feature, and thus the horizontal partitioning scheme of L_2' is centered around the only local feature, namely, f_2. In summary, the `replicate` operation can reduce the scan cost because of more effective data skipping while increasing storage cost due to replication.



(a) initial layout scheme L

(b) after *repl*(L, L_2, f_3)

(c) after *merge*(L, L_2, L_3)

(d) after *merge-repl*(L, L_2, L_3)

Figure 5.5: Examples of Scheme Transformation Operations.

2) $\texttt{merge}(\texttt{L}, \texttt{L}_\texttt{i}, \texttt{L}_\texttt{j})$: This `merge` operation merges two layouts L_i and L_j by taking the *union* of their column sets and feature sets. It creates a new layout L_{k+1}, where $C_{k+1} = C_i \cup C_j$ and $F_{k+1} = F_i \cup F_j$. Thus, we have:

$$\texttt{merge}(L, L_i, L_j) = L - L_i - L_j + L_{k+1}$$

As shown in Figure 5.5(c), applying $\texttt{merge}(\texttt{L}, \texttt{L}_2, \texttt{L}_3)$ on the initial layout L creates a new layout L_5, where $C_5 = \{c_2, c_3\}$ and $F_5 = \{f_1, f_2, f_3\}$.

The operation has the same effect as a column grouping operation in GSOP. Merging two layouts will align the columns from both layouts. For example, column c_2 and c_3 become aligned in L_5 and we only need to store one set of row ids for both columns. Queries also can read fewer row ids and avoid joins when reconstructing rows from these two columns. On the other hand, this operation may hurt the effectiveness of data skipping due to increased feature conflict. This is because the two columns in the newly merged layout L_5 now have to be partitioned in the same way, i.e., based on the union of two feature sets. In this case, columns c_2 and c_3 are horizontally partitioned based on three features f_1, f_2, f_3, which may result in less effective data skipping than partitioning the two columns separately.

3) $\texttt{merge-repl}(\texttt{L}, \texttt{L}_\texttt{i}, \texttt{L}_\texttt{j})$: This `merge-replicate` operation merges two layouts L_i and L_j by taking the *union* of their column sets and taking the *intersection* of their feature sets. It creates a new layout L_{k+1}, where $C_{k+1} = C_i \cup C_j$ and $F_{k+1} = F_i \cap F_j$. To retain the features that do not fall in the intersection F_{k+1}, this operation additionally creates two new layouts L_{k+2} and L_{k+3}. In layout L_{k+2}, we replicate the data cells requested by $F_i - F_j$, i.e., $F_{k+2} = F_i - F_j$ and $C_{k+2} = C_i \cap \bigcup_{f_k \in F_{k+2}} \texttt{rel-cols}(f_k)$. Similarly, for L_{k+3}, we have $F_{k+3} = F_j - F_i$ and $C_{k+3} = C_j \cap \bigcup_{f_p \in F_{k+3}} \texttt{rel-cols}(f_p)$. Putting it together, we have:

$$\texttt{merge-repl}(L, L_i, L_j) = L - L_i - L_j + L_{k+1} + L_{k+2} + L_{k+3}$$

Figure 5.5(d) illustrates the result of applying $\texttt{merge}(\texttt{L}, \texttt{L}_2, \texttt{L}_3)$ on the initial layout L in Figure 5.5(a). This creates a new layout L_5, where $C_5 = \{c_2, c_3\}$ and $F_5 = f_1$, and two new layouts L_6 and L_7 by replicating data cells from column c_2 in L_2 and column c_3 in L_3, respectively, whose local features are the features that do not fall in F_5, i.e., $F_6 = \{f_3\}$ and $F_7 = \{f_2\}$. As compared to a `merge` operation, a `merge-replicate` operation keeps the benefit of merging columns from two layouts, i..e., reduced cost of row-reconstruction, but avoids its downside, i.e., increased feature conflict, by taking the intersection of the two feature sets, instead of their union. This way both the scan cost and row-reconstruction cost can be reduced, at the expense of the increased storage cost.

As we can see, no single type of operation is expected to outperform the other types at all times. Our goal is to always select the operation for the current scheme that minimizes the query cost while not exceeding the storage budget. We summarize the characteristics of three types of operations in the following table.

operations	data scan	reconstruction	storage
`repl`	↘	–	↗
`merge`	↗	↘	–
`merge-repl`	↘	↘	↗

5.5.2 Automatic Layout Merge

When the local features of two layouts L_i and L_j are identical, $\texttt{merge-replicate}(\texttt{L}, \texttt{L}_\texttt{i}, \texttt{L}_\texttt{j})$ and $\texttt{merge}(\texttt{L}, \texttt{L}_\texttt{i}, \texttt{L}_\texttt{j})$ lead to the same result. More importantly, since their local features are identical, merging these two layouts does not cause increased feature conflict or replication, and we can enjoy the upside of the merge, i.e., reduced reconstruction cost, for free. Therefore, we always merge the layouts with the same local features for the initial scheme and for newly transformated schemes. For example, after applying the $\texttt{repl}(L, L_2, f_3)$ operation in Figure 5.5(b), the new layout L_5 has the same local features as L_4, so we merge them, which results in the scheme shown in Figure 5.4(b). When choosing the best transformation operation for the current scheme, we will evaluate the resulting scheme *after* these automatic merges take place.

To efficiently merge the layouts that share the same local features, we maintain the layouts in L using a (feature-set, layout) map structure during the search of GSOP-R schemes. For each layout L_i in L, we insert an entry $(F_i \rightarrow L_i)$ to the map. Upon insertion, if a feature set that is identical to F_i already exists as a key in the map, we merge L_i with that existing layout in the map by taking a union of their column sets. For example, after applying $\texttt{repl}(L, L_2, f_3)$ operation in Figure 5.5(b), we insert the new layout L_5 to the map, which automatically merges it with L_4.

5.5.3 Specifying Master Columns

The above transformation operations are only focused on the columns and local features. As mentioned in Section 5.4, we also need to specify master columns in a GSOP-R scheme in order to to make sure there is no data loss after the GSOP-R scheme is applied. We now discuss a uniform way of specifying master columns on every newly transformed scheme.

Since each layout does not necessarily contain all the rows, we need the notion of master columns to ensure the completeness of the scheme, i.e., every data cell of the original data can be found in at least one layout. Specifying a master column in a layout enforces this layout to have all the data cells from that column. As defined in Section 5.4, the row set of layout L_i is R_i, i.e., the rows that satisfy at least one feature in F_i. When a column is specified as a master colum, L_i has to include the additional data cells from $\overline{R_i}$ for this column. It is important to note that the specification of master columns does not affect the query cost of a GSOP-R scheme. This is because the rows in $\overline{R_i}$ do not satisfy any feature and their presence in L_i does not affect the partitioning schemes of the rows in R_i. Thus, our goal is to specify the master columns in a way that minimizes the storage cost. This can be achieved by following two simple rules. First, a column does not need to be a master column in more than one layout. Second, for every column c_i in C, we set c_i as a master column in layout L_j if L_j has the largest row set, i.e., $|R_j|$ is the

largest, of all the layouts that contain c_i; this way the extra storage cost of turning c_i into a master column, i.e., $|\overline{R_j}|$, is the smallest.

Algorithm 1: `find-GSOPR-scheme`

Input: data sample $S(R, C)$; workload W; storage budget b
Output: a GSOP-R scheme $bestScheme$

```
1  F ← workload-analysis(W)
2  V ← featurize(S, F)
3  L ← construct-initial-scheme(C, F)
4  op ← Null
5  qCost ← Null
6  bestQCost ← MaxValue
7  bestScheme ← L
8  do
9      (op, qCost) ← pick-next-operation(L, W, V, b)
10     if nextOp ≠ Null then
11         L ← apply(op, L)
12         if qCost < bestQCost then
13             bestQCost ← qCost
14             bestScheme ← L
15 while op ≠ Null
16 return bestScheme
```

5.5.4 The Search Framework

Given a data sample $S(R, C)$, a workload W, and a storage budget b, the framework of searching a GSOP-R layout scheme is presented in Algorithm 1. We first perform a workload analysis step to extract features F from the workload W (line 1). This step is the same as in GSOP, as reviewed in Section 5.2. We then apply a featurization step in line 2 to batch evaluate the features on the data sample, which turns each row into a feature vector (line 2). This step is also the same as in GSOP. The subsequent search of layout schemes will be solely based on the vectors V, instead of the actual rows R. We then construct an initial layout scheme L based on the column set C and the feature set F (line 3), as described in Section 5.5.1. We go through an iterative process (line 8-15). At each iteration, we greedily find the best transformation operation using the `pick-next-operation` subroutine and apply it on the current scheme. As discussed in the next paragraph, scheme transformation using the three types of operations defined in Section 5.5.1 does not generate *cycles*. The iteration will stop when no transformation is possible. Finally, we return the best scheme we have seen during the process.

The procedure of `pick-next-operation` is presented in Algorithm 2. For each layout that has more than one local feature, we try a `replicate` operation on every local feature (line 3-10). For each resulting scheme, we estimate the query cost and storage cost. We exclude an operation

Algorithm 2: `pick-next-operation`

Input: layout scheme L; workload W; vectors V; storage budget b
Output: best operation $bestOp$; best query cost $bestQCost$

```
1  bestQCost ← MaxValue
2  bestOp ← Null
3  for each L_i: (C_i, F_i, M_i) ∈ L do
4      if |F_i| > 1 then
5          for each f_j ∈ F_i do
6              L' ← repl(L, L_i, f_j)
7              qCost ← q-cost(W, L', V)
8              sCost ← s-cost(L', V)
9              if sCost ≤ b ∧ qCost < bestQCost then
10                 bestOp ← (repl, L_i, f_j)
11                 bestQCost ← qCost
12 for each L_i: (C_i, F_i, M_i) ∈ L do
13     for each L_j: (C_j, F_j, M_j) ∈ L do
14         if i ≠ j ∧ F_i ∩ F_j ≠ ∅ then
15             L' ← merge(L, L_i, L_j)
16             qCost = q-cost(W, L', V)
17             if qCost < bestQCost then
18                 bestOp ← (merge, L_i, L_j)
19                 bestQCost ← qCost
20             L' ← merge-repl(L, L_i, L_j)
21             qcost ← q-cost(W, L', V)
22             sCost ← s-cost(L', V)
23             if sCost ≤ b ∧ qCost < bestQCost then
24                 bestOp ← (merge-repl, L_i, L_j)
25                 bestQCost ← qCost
26 return (bestOp, bestQCost)
```

from consideration if the resulting storage cost exceeds the budget b. The query cost (line 7) and the storage cost (line 8) can be estimated using Equation 5.1 and Equation 5.2 in Section 5.4.3, respectively. Similarly, we examine all possible `merge` and `merge-replicate` operations (line 11-24). One important restriction here is that we only try to merge the layouts that have at least one local feature in common. When two layouts do not share any feature, then we know that there is no query in the workload that requests columns from both layouts simultaneously. Thus, merging the two layouts would not improve the query cost based on our model. Also, this restriction makes sure the newly replicated layouts (resulting from `replicate` and `merge-replicate`) will never be merged back to their originating layouts and no cycles will be generated through successive scheme transformations.

As we can see, the transformations will end on a scheme where each layout only has one local feature and no pair of layouts shares a local feature. In practice, the space budget is usually

small, e.g., twice the data size, so the process can stop in just a few iterations. The bottleneck in this process is the estimation of query cost and storage cost. In particular, estimating the query cost needs the information of the union vectors, as discussed in Section 5.6.2, and thus involves the horizontal partitioning of the vectors V. Since each transformation operation only makes changes to a few layouts in a scheme, in our implementation, we reuse the cost estimations as much as possible. For example, we cache union vectors of different partitioning schemes so we do not have to actually perform the partitioning every time. Cost estimation reuse of this kind can greatly speed up the search process.

5.6 Query Processing

In this section, we discuss how a query retrieves data from the data stored in a GSOP-R scheme.

5.6.1 Eligible Layouts for Columns

Since a column can have multiple replications, which reside in different layouts, a query has multiple options for where to retrieve a certain column. However, not all of these column replicas can be used to answer the query, as a column replica might not contain all the values unless it is a master column. To query the data stored in a GSOP-R scheme, the first step is to identify which layouts are *eligible* to serve a certain column to the query.

Metadata. We maintain three catalog tables, as shown in Figure 5.6: a feature catalog, a layout catalog and a block catalog. The feature catalog lists the features used in the design of the GSOP-R scheme. The layout catalog maintains the information of layouts, where each layout is described by three fields: columns, local features, and master columns. The local feature field is represented as a *mask vector*, where the i-th bit indicates whether feature i is a local feature of the layout. In Figure 5.6, the mask vector $(1, 1, 0)$ of layout L_1 means that L_1 has features f_1 and f_2 but not f_3. We defer the discussion of the block catalog to Section 5.6.2.

Given a query, we first check the feature catalog and see which features subsume this query. The result is encoded in a *query vector*. Since the query in Figure 5.6 is subsumed by feature f_3, but not f_1 or f_2, the query vector is $(0, 0, 1)$. We also extract the columns requested by the query, i.e., c_2, c_3 and c_4. Although both layouts have column c_2, they may not both be eligible to provide column c_2 to the query, as they may not have all the rows that the query needs. As defined in Section 5.4, a layout contains all the rows that satisfy at least one feature of the layout. Thus, to see if a layout contains all the rows requested by the query, we can check if any of its local features subsumes (or is more general than) the query predicate. If so, we are guaranteed that the row set of this layout is a superset of the rows needed by the query. To perform this check, we take the intersection of the query vector and a mask vector in the layout catalog and see if the resulting vector is empty. In Figure 5.6, by taking the intersection of the query vector $(0, 0, 1)$ and the mask vector of L_2 $(0, 1, 1)$, we obtain the result $(0, 0, 1)$, which is not empty. This tells us that L_2 contains all the rows that the query needs and thus is an eligible layout for column c_2.

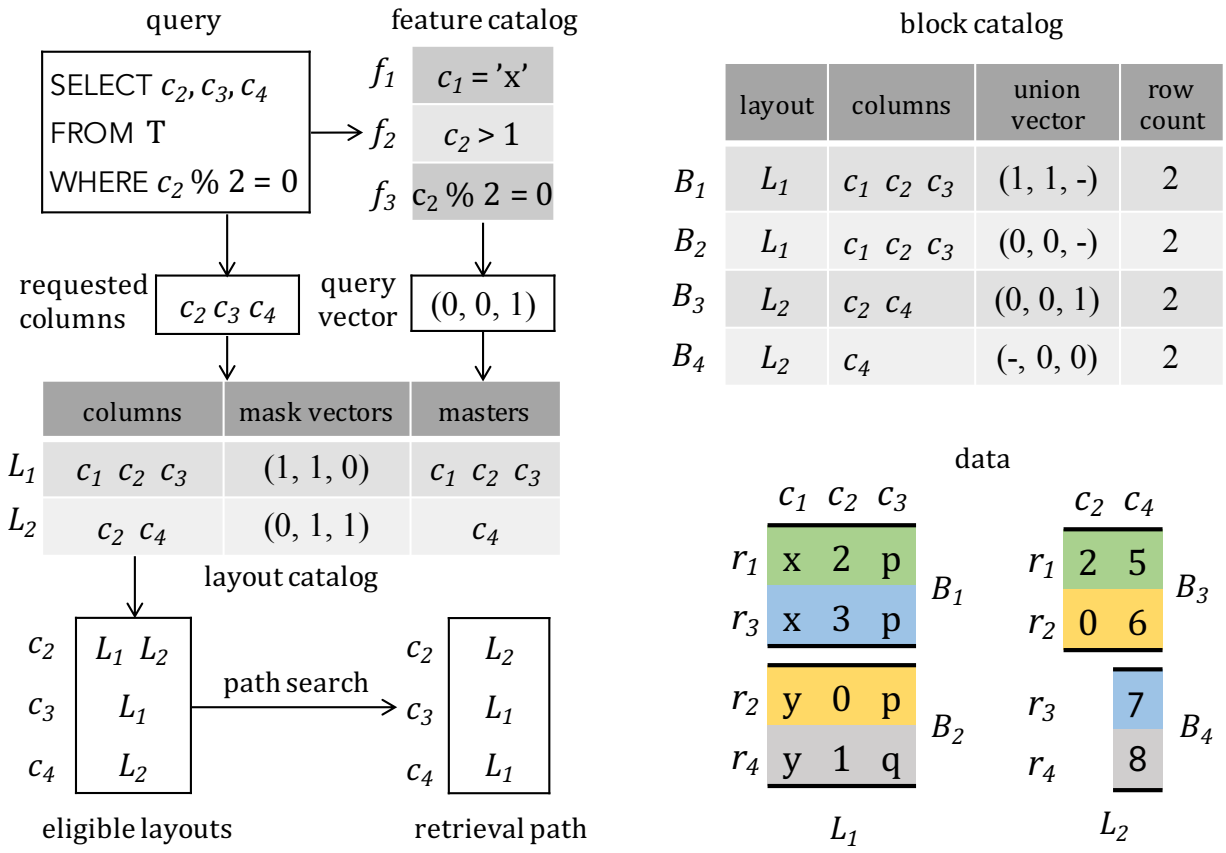


Figure 5.6: Example of Query Processing in GSOP-R.

To summarize, for a given column c_i requested by a query, a layout L_j is *eligible* if 1) c_i is a master column in L_j or 2) c_i is a column in L_j and at least one feature in L_j subsumes the query predicate. In Figure 5.6, the set of eligible layouts of c_2 is $\{L_1, L_2\}$ and that for c_3 and c_4 are both $\{L_1\}$.

5.6.2 Evaluating Retrieval Paths

Given the eligible layouts for each column requested by the query, we need to find a combination of eligible layouts that specifies how the query will retrieve all the columns. As mentioned in Section 5.4.3, a *retrieval path* p of a query is an assignment from each requested column to one of its eligible layouts. The query in Figure 5.6 has two retrieval paths: 1) $\{c_2 \rightarrow L_1, c_3 \rightarrow L_1, c_4 \rightarrow L_2\}$ and 2) $\{c_2 \rightarrow L_2, c_3 \rightarrow L_1, c_4 \rightarrow L_2\}$.

Different eligible retrieval paths for a query can incur dramatically different costs for both data scan and row reconstruction. To evaluate the cost of a retrieval path using Equation 5.1 in Section 5.4.3, we need to estimate the value of $\texttt{rr}(q, L_j)$, the number of rows query q needs to

read in layout L_j. We can consult the *block catalog* to estimate the value of $\texttt{rr}(q, L_j)$. The block catalog table consists of the information for the blocks from all layouts in a GSOP-R scheme, as shown in Figure 5.6. For each block, a *layout* field says which layout this block belongs to; a *columns* field indicates which columns are present in this block; a *union vector* field stores the union vector (Section 5.2); and a row count field states the number of rows in this block. Suppose we need to evaluate the retrieval path p: $\{c_2 \rightarrow L_2, c_3 \rightarrow L_1, c_4 \rightarrow L_2\}$. Let us start with L_2, where we need to read column c_2 and c_4. According to the block catalog, two blocks B_3 and B_4 are from layout L_2. We then use the union vector field (Section 5.2) to determine if any of the two blocks can be skipped. We compare the query vector with the union vectors and find that block B_4 can be safely skipped, as the third bit 0 of its union vector indicates that this block does not contain any row that satisfies feature f_3, which subsumes the query predicate. Thus, the query only needs to read block B_3 from L_2, which has 2 rows according to the row count field. This way we have estimated that the query needs to read 2 rows from L_2, i.e., $\texttt{rr}(q, L_2) = 2$. The total number of data cells read from L_2 is 4. For layout L_1, the query reads one column, i.e., c_3. Since f_3 is not a local feature, the query cannot skip any block and has to read c_3 from both blocks B_1 and B_2, which have 4 rows total, i.e., $\texttt{rr}(q, L_1) = 4$. The number of row ids read in a layout equals to the number of rows read from that layout. Thus, 4 row ids from L_1 and 2 row ids from L_2. Using Equation 5.1, $\texttt{q-cost}(q, L, \texttt{p})$ is 14.

5.6.3 Finding Optimal Retrieval Paths

We now consider the problem of finding the optimal retrieval path, i.e., the one with the smallest cost. We denote by $\texttt{eligibleL}(c_i, q, L) \subseteq L$ the set of eligible layouts for column c_i, i.e., $\texttt{eligibleL}(c_i, q, L) \subseteq L$. An eligible path p is an assignment from every column c_i in C^q to a layout in $\texttt{eligibleL}(c_i, q, L)$, where $\texttt{p}(c_i, L_j)$ evaluates 1 if c_i is assigned to L_j and 0 otherwise. We can also estimate the number of rows read from each layout, i.e., $\texttt{rr}(q, L_j)$, using the block catalog as described in Section 5.6.2. We formulate the problem of finding the optimal retrieval path as follows:

Problem 3 *Given a set of columns requested by the query C^q, a set of layouts L, a cost value $\texttt{rr}(q, L_j)$ for every L_j in L, and a set of eligible layouts $\texttt{eligibleL}(c_i, q, L)$ for every column c_i in C^q, find an eligible retrieval path p, i.e., an assignment from every column c_i in C^q to a layout in $\texttt{eligibleL}(c_i, q, L)$, such that $\texttt{q-cost}(q, L, \texttt{p})$ (Equation 5.1) is the smallest.*

We can prove that Problem 3 is NP-hard by reduction from the set cover problem. From an input of set cover, we can construct an input to a special case of our problem, where the value of $\texttt{rr}(q, L_j)$ is 1 for every L_j in L. Thus, if we could solve Problem 1 in polynomial time, we could also solve set cover in polynomial time. We next discuss two approaches for solving this problem.

Brute-force. A brute-force approach is to materialize all eligible retrieval paths and pick the one with the smallest cost. This approach can find the optimal path but can be prohibitively expensive, as the total number of eligible retrieval paths can be combinatorially large. In the worst case, where every layout in L is eligible for every column in C^q, the number of eligible

retrieval paths is $|L|^{|C^q|}$. In practice, both $|L|$ and C^q can be in the order of tens. The process of finding a retrieval path is executed online as part of query optimization. Thus, its efficiency is critical.

Greedy. Instead of materializing all eligible retrieval paths, we can pick an eligible layout for each column in a greedy manner. Since we can estimate the number of rows query q needs to read from each layout L_j in L, i.e., $\texttt{rr}(q, L_j)$, we simply pick, for each column in C^q, the eligible layout L_j with the smallest value of $\texttt{rr}(q, L_j)$. This way, we can guarantee that the path generated using this approach reads the smallest number of data cells. It may, however, read more row ids than the optimal path, as it does not consider which columns are from the same layout. Note that the number of row ids scanned by a query can never be larger than the number of data cells it scans. Since the greedy algorithm makes sure that the query reads the optimal number of data cells, we can infer that the total number of data cells and row ids read (Equation 5.1) by using the greedy algorithm is at most twice the optimal. Thus, this greedy approach is a 2-approximation solution of Problem 3, i.e., the cost of the path found is at most twice the optimal. Even so, our experiments in Section 5.7 show that the greedy approach almost always finds the optimal path.

After choosing the retrieval path, the query can read the columns as specified by the path, assemble the columns into rows [66], and pass the assembled rows to the subsequent stages of query processing.

5.7 Experiments

We implemented the GSOP-R framework using Apache Spark [44]. Given a GSOP-R scheme, we loaded the data into a GSOP-R-aware Parquet [14] files. Each Parquet file contains several millions of rows stored in a GSOP-R scheme and maintains the layout catalog and the block catalog (Figure 5.6) as file metadata.

All the expeirments were conducted on a Spark cluster of 9 Amazon Ec2 i3.2xlarge instances, with 1 master node and 8 slave nodes.

5.7.1 Workloads

TPC-H (Denormalized). TPC-H [67] is a well-known benchmark for ad-hoc analytics workloads, which provides data and query generators. We use the TPC-H benchmark in the same way as in Chapter 3, except that we use a different set of query templates, as described below. We generate the data using a scale factor of 100. The schema of the TPC-H data is normalized. Since we focus on using GSOP-R to design the layout scheme of a single table, by following the evaluation approach in Chapter 4, we denormalize all the TPC-H tables, which results in a single table with 600 million rows and 70 columns. The results from the GSOP work [66] suggested that effective data skipping can lead to significant improvement of query performance for both normalized and denormalized schemes. In this chapter, we only focus on the denormalized case, as the performance gains of GSOP-R over GSOP can be directly translated to the normalized case as well.

We first horizontally partition the data based on the month of the `l_orderdate` column. Each month partition consists of roughly 8 millions rows and is stored in a standalone Parquet file. We then apply GSOP-R on each Parquet file individually. TPC-H also provides query templates. We construct the TPC-H workload using 14 query templates with selective filters, namely, q_2, q_3, q_5, q_6, q_7, q_8, q_{11}, q_{12}, q_{16}, q_{17}, q_{19}, q_{20}, q_{21} and q_{22}. The number of columns accessed by these queries are: 11, 7,7,4, 5, 6, 4, 5, 4, 3, 8, 3, 5 and 2, respectively. We generate a training workload of $1,400$ queries, 100 queries from each template. We then independently generate 140 queries as the test workload, 10 queries from each template.

SDSS. Sloan Digital Sky Survey (SDSS) [63] provides public datasets on the photometric observations from the sky. These data can be accessed via a SQL interface. The access logs is also publicly available at [20]. In our experiment, we use the same dataset and workload as in Chapter 4. Specifically, we consider the Star table, which records the photometric parameters of all primary point-like objects in the sky. This table consists of over 260 millions rows and 453 columns. We use GSOP-R to design the layout of 4 millions rows at a time. We collect 2340 queries from the SQL access log between 01/2011 and 06/2011. After sorting these queries based on their arrival time, we use the first 3/4 as the training workload and the rest 1/4 as the testing workload. The numbers of columns accessed by the queries have a mean of 13.6 and a standard deviation of 5.13.

5.7.2 TPC-H Results

Query Performance

In Figure 5.7, we compare the query performance on the data stored in 4 alternative layout schemes. GSOP is the state-of-the-art layout design framework from [66], where no replication is used. GSOPR1.1, GSOPR1.2 and GSOPR1.5 are based on our proposed GSOP-R framework with a storage cost 1.1, 1.2, and 1.5, respectively. The storage cost is as defined in Section 5.4.3, i.e., a storage cost 1.1 means using 10% extra storage. Figure 5.7(a) shows the number of data cells and row ids an average test query scanned for the data stored in different schemes. On average, a query scanned 32×10^8 data cells on the original data without any layout design. As TPC-H is a complex worload with diverse filter predicates and skewed column access patterns, there is a high degree of conflict in the features extracted from this workload. To mitigate the feature conflict, GSOP generates as many as 17 column groups. However, vertically partitioning the columns into a lot of groups still cannot eliminate the feature conflict coming from individual columns. On the other hand, with only 20% extra storage, GSOPR1.2 can find a layout scheme that effectively mitigate the feature conflict and thereby reduce the number of data cells and row ids by a factor of 2 as compared to GSOP; with 50% extra storage, GSOPR1.5 can improve `gsop` by a factor of 3.

Figure 5.7(b) shows the end-to-end query response time for these approaches. We can see that the reduction in data scan can directly translate to the improvement in end-to-end query response time. In fact, as less data is scanned, the CPU cost of row reconstruction is significantly improved, which is not reflected in 5.7(a). Although GSOP-R allows for much more flexible schemes than GSOP, querying the data stored in a GSOP-R scheme, as discussed in Section 5.6, involves little

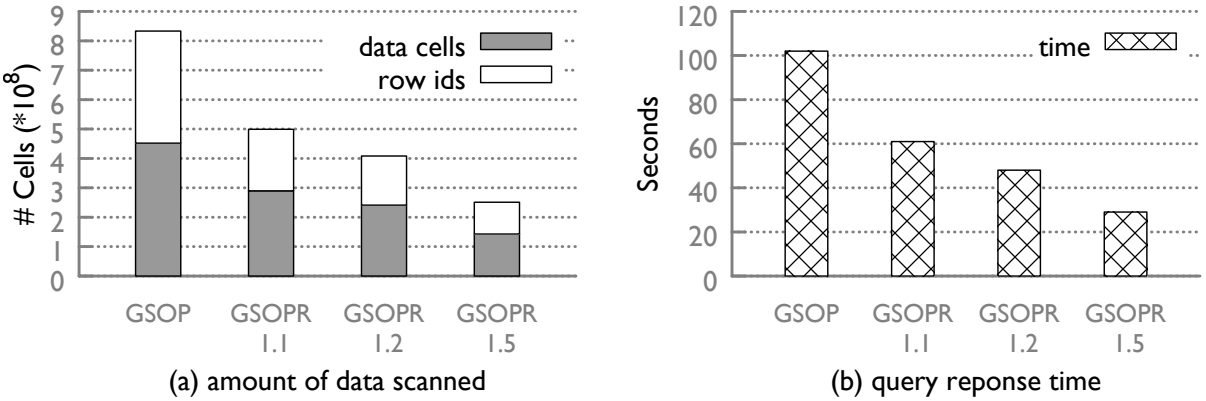


Figure 5.7: Query Performance Results of GSOP-R (TPC-H)

overhead. Specifically, GSOPR1.1 GSOPR1.2 and GSOPR1.5 can improve the query response time over gsop by a factor of 1.6, 2, and 3 respectively. For reference, the average query response time on the original data in Parquet without any layout design is 161 seconds.

Basic Replication Approaches

In Figure 5.8, we compare the effectiveness of GSOP-R against the basic approaches proposed in Section 5.3. GSOP-FC is the full-copy approach, which makes full copies of data and applies GSOP on each copy based on a different set of features. GSOP-SE is the selective approach, which first applies GSOP on the data and then greedily picks the most cost-effective features and replicates the data they request. Both approaches borrow ideas from classic database techniques that trade space for query performance and use these ideas to extend GSOP with replication. Clearly, for all these approaches, the more extra storage they use, the better the queries will perform. In this experiment, we focus on the *efficiency* of storage use. In Figure 5.8, we plot the number of data cells plus row ids scanned by an average query vs. the storage cost, where 2 means 100% overhead. As we can see, what GSOP-R can achieve with less than 50% overhead takes GSOP-SE a storage cost of 5, i.e., a 400%, storage overhead. The drawback of GSOP-SE is that it replicates the data for each feature separately and does not consider the correlation between features. For example, when two features request similar sets of data cells, they can be grouped together for replication in order to save the cost of storing these data cells twice. Such kind of correlations are naturally incorporated in the design of GSOP-R schemes. GSOP-FC has the least efficient use of storage, as it only supports full-copy replication. The promise of GSOP-FC, however, is that it can leverage the existing full copies for reliability or availability purposes. For example, when the data is stored in a 3-way replication, we can use GSOP-FC to design a GSOP layout for each replica, which improves GSOP by almost 2x with zero storage overhead.

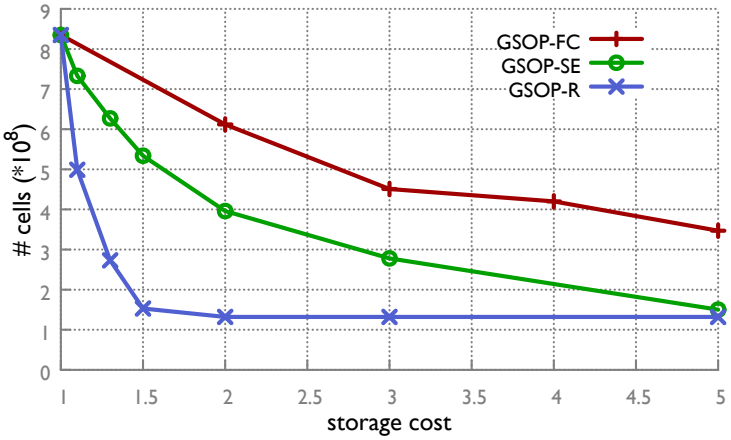


Figure 5.8: Query Performance Results of GSOP-R vs Basic Replication Approaches (TPC-H)

Loading Cost

The data loading costs are shown in Figure 5.9. PAR represents the cost of directly loading data into Parquet. The cost of GSOP and GSOP-R can be broken down into 2 phases. Phase 1 is the layout scheme design. In Phase 1, GSOP finds a column grouping scheme, and GSOP-R performs the layout scheme search based on scheme transformations as described in Section 5.5. As we can see, GSOP-R needs more time to find the layout scheme when there is a larger storage budget. This is because the search space of GSOP-R schemes is proportional to the storage budget. Recall that the search of GSOP-R schemes is an iterative process. If the storage budget runs out at a certain iteration, we only need to consider `merge` transformations in the subsequent iterations and rule out the `replicate` and `merge-replicate` as they incur storage overhead. Thus, with a small storage budget, GSOP-R does not take much longer in Phase 1 than GSOP. The Phase 2 is a data loading phase, where GSOP-R does not differ much from GSOP. First, as the data is being scanned, GSOP-R batch-evaluates the features on each row. Then, each layout will be horizontally partitioned based on the feature vectors. Then, the data cells will go to their destination blocks based on both the GSOP-R layout scheme and horizontal partitioning scheme. As we can see in Figure 5.9, the cost of Phase 2 is similar for different alternatives. In summary, GSOP-R does not take much more time than GSOP for data loading, but can greatly improve the query performance, as shown previously. In particular, GSOPR1.5 can improve GSOP by 3x with less than 20% overhead in data loading cost.

Retrieval Path Search

As GSOP-R schemes allow data replication, queries can find the requested data via different retrieval paths. In Section 5.6.3, we propose two approaches to find retrieval paths, namely, brute-force and greedy. The brute-force approach can always find the optimal retrieval path, but may take a long time to run when there are a lot of column replications. The greedy approach can

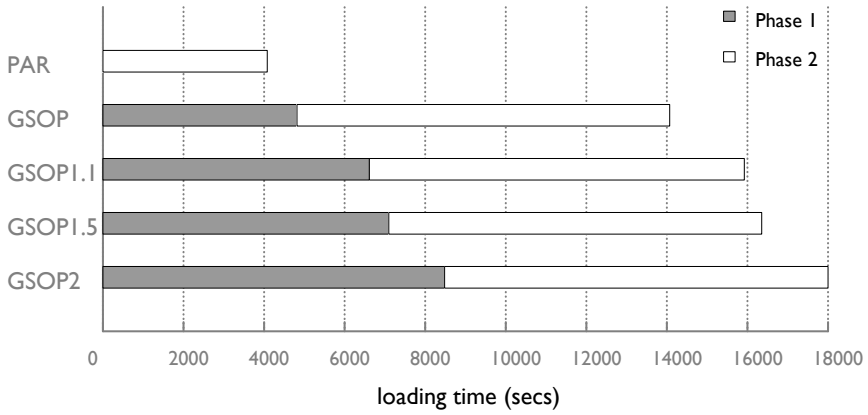


Figure 5.9: Loading Cost of GSOP-R (TPC-H)

quickly find a path that has the smallest data scan cost but may need to read more row ids than the optimal path. In our experiments with different GSOP-R schemes on TPC-H, namely, GSOPR1.1, GSOPR1.2, GSOPR1.5, and GSOPR2, we found that the greedy approach almost always returned the same path as the brute-force approach. We observed the cost of scanning a column from different eligible layouts in these schemes can vary dramatically. This is a favorable situation for the greedy approach. Since the number of row ids read cannot be larger than the number of data cells read, by focusing on picking a path that scans far fewer data cells than other paths, the greedy approach can almost always find the optimal path.

5.7.3 SDSS Results

Figure 5.10 shows the query performance of the SDSS workload. We compare three alternantives: the GSOP scheme, GSOPR1.5 and GSOPR2. As we can see, GSOP-R can improve GSOP by 1.4x and 1.8x with 50% and 100% storage overhead, respectively. Note that SDSS is a scientific workload, whose queries tend to read many more columns than in TPC-H. The average number of columns accessed by a query in SDSS and in TPC-H are 13.1 and 5.3, respectively. Thus, the columns in SDSS tend to be involved in more features than in TPC-H and suffer more from feature conflict. To achieve the similar improvement of query performance, GSOP-R needs to replicate more columns in SDSS than in TPC-H.

5.8 Related Work

Existing techniques have used many forms of replication for improving query performance. Traditional indexes such as B+-trees are a form of data replication. Although a table can only have one primary index, we can create many secondary indexes, each of which replicates one or more columns of the table and sorts them in a different way from the original table [21] to facilitate a

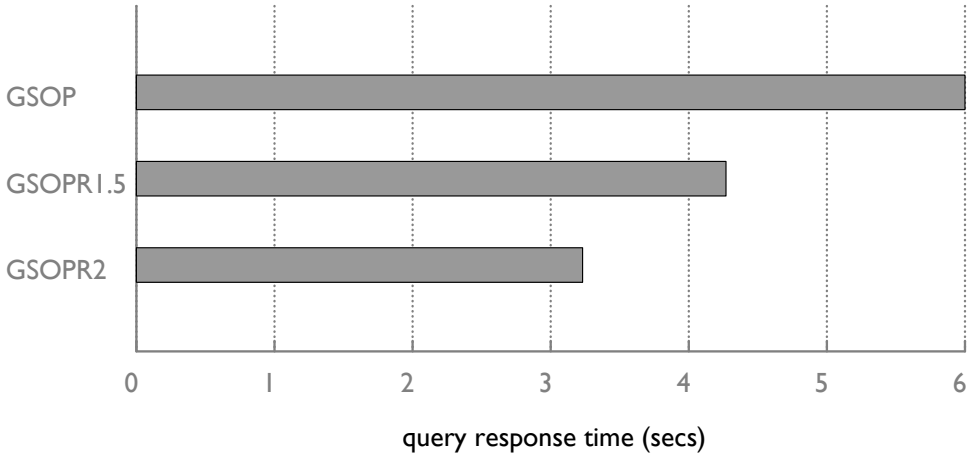


Figure 5.10: Query Performance Results of GSOP-R (SDSS)

different set of queries. Similarly, in column stores, columns can be replicated, and these column replications reside in different column groups, each of which can have a different sort orders [43, 5]. Database cracking [35, 61, 36] creates a replication for a column upon being accessed and adaptively changes its layout based on the workload. These forms of replications are *column-oriented*. In contrast, materialized views [56, 17] can be considered as a form of *row-oriented* replication. In a materialized view, the rows that satisfy the query are replicated and clustered together instead of being scattered throughout the original table. Schism [24] also replicates rows to minimize the number of distributed transactions.

Different from these existing forms of replication, the design of GSOP-R schemes is driven by workload features and can express both row-oriented and column-oriented replications. In a GSOP-R scheme, both rows and columns can be replicated in mulitple layouts. Each column replication can have a different horizontal partitioning scheme. Each row replication is surrounded by different sets of rows and will be used to answer a different set of queries. Neither a row or a column is atomic unit of replication in a GSOP-R scheme. GSOP-R also has a different goal from the existing techniques, which is to pack data cells into fine-grained blocks so that the queries can skip as many blocks as possible.

There is also research that exploits the full-copy replication schemes for fault-tolerance and availability purposes. Fractured mirrors [54] leverages a mirroring scheme (RAID 1), and stores one data copy in a row-major layout and the other in a column-major layout. The Trojan layout approach [4] proposed to devise a different column grouping scheme for each of the 3 replicas of a HDFS block so that each replica is best at handling a different subset of queries. GSOP-R is designed for making the most efficient use of extra storage rather than utilizing full-copy replications. Nevertheless, in this chapter, we also develop an approach that enables GSOP to take advantage of full-copy replications. If some extra storage is allowed for each full copy, we can apply GSOP-R to design the layout of each copy.

5.9 Conclusion

We developed GSOP-R, a layout design framework for data skipping. GSOP-R extends GSOP by considering fine-grained partitioning and replication schemes in an integrated manner. We evaluated GSOP-R using TPC-H and a real-world workload. The results show that, by paying a little extra storage, GSOP-R can significantly improve query performance over the state-of-the-art GSOP framework. In our experiments, GSOP-R improved the query performance over GSOP by 2x and 3x with only 20% and 50% storage overhead, respectively. In practice, we recommend using a moderate amount of extra storage and leveraging GSOP-R for the significant query performance improvement.

Chapter 6

Conclusions

In this chapter, we conclude the dissertation. We first summarize the physical design frameworks proposed in the dissertation. We then highlight the innovations in designing these frameworks. Finally, we discuss future work.

6.1 Summary of Proposed Frameworks

In this dissertation, we develop the following three frameworks for skipping-oriented physical database design:

SOP: Skipping-oriented Partitioning

We describe the Skipping-oriented Partitioning Framework (SOP) in Chapter 3. The SOP framework is focused on finding fine-grained horizontal partitioning schemes that can maximally help queries skip data. In this chapter, we analyze real-world analytics workloads and make two important observations that motivate the design of the workload analysis techniques. The first observation is that the filter usage is highly skewed, which suggests that we can represent most queries in the workload with a small number of workload features. The second observation is that filter usage remains stable over time, which assures us that we can utilize a past workload to guide the physical design in order to help future queries.

Given the workload features, we show how to use them for data partitioning. This step involves first generating a feature vector for each row of the table and then running a clustering algorithm on the feature vectors to generate a partitioning map. The partitioning map eventually guides the rows to their destination blocks. We further explain that the union of feature vectors, or union vectors, can be stored as block-level metadata, which can work in conjunction with traditional metadata such as min/max values. We discuss how queries can make use of such metadata for data skipping. At the end of this chapter, our experimental results show that SOP can significantly outperform existing techniques, such as range partitioning. Specifically, SOP scans 5-7x less data, which directly translates to the reduction on query response times for both memory- and disk-resident data.

GSOP: Generalized Skipping-oriented Partitioning for Columnar Layouts

We present the Generalized Skipping-oriented Partitioning Framework (GSOP) in Chapter 4. The GSOP framework generalizes the SOP framework by allowing different columns to have different partitioning schemes. Thus, the GSOP framework incorporates horizontal partitioning and vertical partitioning schemes in an integrated manner. In Chapter 4, we first bring up the notion of feature conflict, which expresses that the best partitioning schemes for different features can be dramatically different. We then point out that the effectiveness of SOP is highly sensitive to feature conflict, because SOP forces all columns to be partitioned in the same way. This constraint can be easily removed in a columnar layout. We define the spectrum of partitioning layouts for columnar layouts. While SOP is only focused on one end of the spectrum, we propose GSOP in order to cover the entire spectrum. To realize this, we introduce two new components in the GSOP framework as compared to SOP.

The first new component in GSOP is column grouping. In contrast to the existing work on column grouping, we develop a cost model that aims to balance the trade-off between the effectiveness of skipping horizontal blocks and the overhead of row-reconstruction. Based on this cost model, we propose efficient algorithms to search for column grouping schemes. The second new component is local feature selection. For each column group, the goal of local feature selection is to find a subset of features that are specific to this column group. These local features will be then used to guide the horizontal partitioning of each column group. Finally, we illustrate how a query can be processed on the data partitioned by GSOP, which involves efficiently reconstructing rows using the data read from multiple column groups. At the end of this chapter, we conduct experiments using two public benchmarks and a real-world dataset from Sloan Digital Sky Survey. Our experiments that compare GSOP with SOP and other baseline approaches show that GSOP significantly outperforms the competitors due to the balance between skipping effectiveness and row-reconstruction overhead. In particular, in TPC-H, GSOP improves SOP by 3.3x in terms of query response time.

GSOP-R: Generalized Skipping-oriented Partitioning with Replication

We propose the Generalized Skipping-oriented Partitioning with Replication Framework (GSOP-R) in Chapter 5. The GSOP-R framework complements the GSOP framework with data replication. In Chapter 5, we first discuss why GSOP cannot fully eliminate feature conflict and illustrate the scenarios where the effectiveness of GSOP is limited. We then explore how the classic ideas of replications can be borrowed in GSOP in order to reduce feature conflict. These approaches, however, treat GSOP as a black box and may not make the most efficient use of storage. This motivates us to design the GSOP-R framework, which natively supports fine-grained replication with partitioning.

We first formally define GSOP-R schemes and show the flexibility and fine-granularity of GSOP-R replication. A GSOP-R scheme is composed of a set of layouts. Each layout is specified by a triplet, namely, columns, local features, and master columns. We propose objective functions to evaluate the goodness of a GSOP-R scheme through three aspects: scan cost, row-reconstruction

cost and storage cost. Given the flexibility in the definition of GSOP-R schemes, finding a good GSOP-R scheme involves a huge search space. We propose to search for GSOP-R schemes through scheme transformations.

We define three types of transformation operations on GSOP-R schemes, namely, `replicate`, `merge` and `merge-replicate`. Each of the three transformations makes a different trade-off between scan cost, row-reconstruction cost and storage cost. Given an initial GSOP-R scheme, we search for GSOP-R schemes by successively applying locally-optimal transformation operations. At the end, we return the best GSOP-R scheme we have seen during the search. Finally, we discuss how to query the data stored in a GSOP-R scheme. Since some data is replicated in a GSOP-R scheme, a query could have different ways of retrieving the requested data. We define retrieval paths to specify how a query can correctly retrieve the data in a GSOP-R scheme. We also formulate the problem of finding optimal retrieval path. At the end of the chapter, Our experiments using TPC-H and Sloan Digitial Sky Survey show that GSOP-R improves the query response time by 2x over GSOP with only 20% storage overhead.

6.2 Innovation Highlights

In the design of the skipping-oriented physical design frameworks, we make several key innovations. These innovations are not tied to any particular framework that we developed. In this chapter, we highlight the innovations we make and hope that these high-level ideas can be helpful for designing physical design frameworks in the future.

Feature-driven Workload Analysis

The query workload provides invaluable information of database access patterns. Existing work has exploited workload information to guide the physical database design. In this dissertation, we propose to extract features from workload. Simply put, features are representative filter predicates. While most existing techniques have only looked at the column names involved in filter predicates, a feature can capture filter predicates in their entirety, including column names, operation types and constants. We simply view a feature as a boolean function that evaluates a row to be either 1 or 0. This abstraction allows us to incorporate as features any kind of filtering predicates occurred in the workload, including user-defined functions (UDF's).

The task of feature-driven workload analysis is to identify a set of features that can subsume the most queries. The notion of subsumption is essential in the context of data skipping. The subsumption relation guarantees us that if we can find a physical database design that helps these features skip data effectively, then this design will also benefit all the queries subsumed by these features. To extract features, we perform frequent pattern analysis on the query log. The counting mechanism has been changed so that the frequency of a feature is the number of queries it subsumes instead of the number of queries it occurs in. We also develop principled ways of eliminating redundancies in the resulting patterns and selecting a set of frequent patterns as final features.

The idea of feature-driven workload analysis is inspired by several observations we make in real-world analytics workloads. We find that the use of filter predicates in the real-world workloads is highly skewed and stable. The skewness suggests that the workload can be succinctly summarized by only a small number of features. The stability ensures that the features generated from a past workload are relevant for future workloads. These properties are essential for the effectiveness of feature-driven workload analysis.

Fine-grained and Flexible Layouts

The layout schemes generated by our proposed frameworks organize data into fine-grained blocks. Typically, each block only contains 1,000's to 100,000's of data cells, where each data cell is an intersection of a row and a column. Existing physical design frameworks are not suitable to generate layout schemes at this granularity. When the data is organized as blocks, a query can check the block-level metadata and decide if it needs to look into the block or simply skips it. Thus, the size of a block only needs to be large enough to make this metadata check negligible. Consider the extreme case, where each block contains one data cell, then skipping a block does not bring in any performance benefit because the query in effect has to go through every data cell anyway. For skipping purposes, a small number like a 1'000 data cells per block can usually makes block skipping worthwhile. In practice, however, the block size is usually lower-bounded by the architecture of the system. For example, for disk resident data, it is desirable to let each block to be large enough to avoid random disk access; for column stores, each block should be large enough to hold enough values for compression purposes. Our proposed techniques are designed to scale well with the block granularity of layout schemes.

To strive for the best chances of skipping blocks, we realize that the layout schemes have to be highly flexible. This means we should break the boundaries of rows and columns when packing data cells into blocks. Different parts of a row or a column can reside in different blocks. Some parts of a row or a column can be replicated while the other parts are not. We explain why such flexibility can effectively boost the skipping effectiveness. However, the skipping effectiveness is not everything. Such flexibility can introduce significant overhead to query processing and hurt the overall query performance. When the data is organized as a set of rows or columns, a query can easily find the data it needs. When the data cells are scattered in different blocks, it is a challenge to make sure the query can correctly find all the data it needs and efficiently assembled them back into rows for further processing. We address this challenge by clearly defining what data each block has and describing how the data in such flexible layouts can be queried.

Optimization Problem Formulation and Heuristic Solutions

Given the fine-granularity and flexibility in the layout design, it would be daunting to manually specify and search for the layout schemes. To automate the search of layout schemes, the challenges include 1) how to evaluate the goodness of a layout scheme and 2) how to efficiently search for them. In our proposed frameworks, we adopt a simple and generic approach to define the objective functions. Specifically, the objective functions are centered around the number of

cells. The data scan cost is defined as the number of data cells read; the row-reconstruction cost is defined as the number of row ids read; and the storage cost for replication is defined as the number of replicated data cells created. These cost models omit many of the implementation details, but serve as an good approximation for evaluating the goodness of layout schemes. Even with such simple cost models, directly evaluating them turned out to be prohibitively expensive. We propose approaches to efficiently and accurately estimate them. Based on these cost models, we adopt greedy algorithms to explore the search space by taking local-optimal steps iteratively. At the end of the iterations, the search algorithm recommends the best layout scheme seen, i.e., the one with the smallest cost according to the objective function.

Predicate-based Metadata and Skipping

Existing systems use data statistics as metadata for data skipping. Most commonly used are the min and max values for each column. Such value-based statistics can skip data blocks for queries with simple range predicates, such as *price > 3*. However, modern analytics workloads involve complex types of filter predicates, such as approximate string matching operations and general user defined functions (UDF's). These filter predicates cannot make use of the statistics-based metadata for data skipping. In our proposed frameworks, each data block in the resulting layout scheme is augmented with a union vector, which encodes the evaluation results of the features, or representative predicates. An incoming query can first check whether a feature can subsume this query's predicate. If there is a such feature, the query can look up the corresponding bit of the union vector on each block and decide whether this block can be skipped. Such a predicate-based skipping mechanism can complement the value-based mechanism by allowing any kind of predicates, including UDF's, to skip data blocks. In our proposed frameworks, this mechanism works in synergy with the feature-based workload analysis and layout scheme design, as the features can collectively subsume most of the queries. Outside of these frameworks, predicate-based metadata in general can also be considered as an approach to complement value-based metadata for data skipping.

6.3 Future Work

While our proposed frameworks are built on several key innovations and can significantly outperform the state-of-the-art techniques, we outline several directions where our work can be further extended.

The first line of future work is on how to incrementally update the layout schemes in the face of changes in both workloads and data. Our proposed frameworks are based on the assumptions that the data are batch loaded and not updated frequently thereafter. When there is a update to the data, we can accordingly update the block-level metadata (i.e., union vectors as well as statistics) to ensure the correctness of skipping. When there is a change in the workload, we can choose not to update the layout schemes without affecting the correctness. However, in both cases, the effectiveness of the layout schemes may be compromised. In the proposed frameworks, the only

way to embrace the changes in the data and/or workloads is to rebuild the layout schemes from scratch. It would be useful to have the ability to incrementally update the layout schemes.

A second line of future work is to extend the proposed frameworks beyond a single table. To use the proposed frameworks on normalized tables, our approach was to denormalize them into a single table first. While the great benefit of using our proposed frameworks can justify the additional denormalization step, we expect that the ability of working with normalized tables natively can make the proposed frameworks even more attractive. This would require fundamental redesign of key aspects of these frameworks, including workload analysis and objective function. For normalized tables, joins can easily become cost bottleneck. It would be interesting and challenging to consider data skipping and joins in the same context.

6.4 Conclusion

As data volumes continue to grow, the efficiency of large-scale query processing is crucial for unlocking insights from enormous data in a timely manner. However, with the advent of Big Data systems in the recent years, many of the traditional techniques in data management may no longer be suitable for these modern architectures. In this dissertation, we propose innovative data layout design frameworks for modern analytics systems with a goal of improving the efficiency of large-scale query processing. Our focus is on data skipping, i.e., reducing the amount of unnecessary data access. As modern analytics systems have increasingly considered data skipping as an important mechanism, our techniques can have a broad impact for improving query efficiency in current and future analytics systems.

Bibliography

[1] A. Ailamaki *et al.* "Data Page Layouts for Relational Databases on Deep Memory Hierarchies". In: *VLDB Journal* 11.3 (Nov. 2002), pp. 198–215.

[2] A. Gupta *et al.* "Amazon Redshift and the Case for Simpler Data Warehouses". In: *SIGMOD*. 2015, pp. 1917–1923.

[3] A. Hall *et al.* "Processing a trillion cells per mouse click". In: *PVLDB* 5.11 (2012), pp. 1436–1446.

[4] A. Jindal *et al.* "Trojan Data Layouts: Right Shoes for a Running Elephant". In: *SOCC*. Cascais, Portugal, 2011, 21:1–21:14.

[5] A. Lamb *et al.* "The Vertica Analytic Database: C-Store 7 years later". In: *VLDB* 5.12 (2012), pp. 1790–1801.

[6] A. Thusoo *et al.* "Hive: a warehousing solution over a map-reduce framework". In: *PVLDB* 2.2 (2009), pp. 1626–1629.

[7] D.J. Abadi et al. "Materialization Strategies in a Column-Oriented DBMS". In: *ICDE*. 2007, pp. 466–475.

[8] Rakesh Agrawal and Ramakrishnan Srikant. "Fast Algorithms for Mining Association Rules in Large Databases". In: *VLDB*. 1994, pp. 487–499.

[9] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. "Integrating vertical and horizontal partitioning into automated physical database design". In: *SIGMOD*. 2004, pp. 359–370.

[10] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. "H2O: A Hands-free Adaptive Store". In: *SIGMOD*. Snowbird, Utah, USA: ACM, 2014, pp. 1103–1114.

[11] Karolina Alexiou, Donald Kossmann, and Per-Ake Larson. "Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia". In: *PVLDB* 6.14 (2013), pp. 1714–1725.

[12] Daniel Aloise et al. "NP-hardness of Euclidean sum-of-squares clustering". In: *Machine Learning* 75.2 (2009), pp. 245–248. ISSN: 0885-6125.

[13] Kamel Aouiche, Pierre-Emmanuel Jouve, and Jérôme Darmont. "Clustering-based materialized view selection in data warehouses". In: *Advances in Databases and Information Systems*. 2006, pp. 81–95.

[14] *Apache Parquet*. http://parquet.apache.org. URL.

[15] B. Bhattacharjee *et al.* "Efficient query processing for multi-dimensionally clustered tables in DB2". In: *VLDB*. 2003, pp. 963–974.

[16] B. Dageville *et al.* "The Snowflake Elastic Data Warehouse". In: *SIGMOD*. 2016, pp. 215–226.

[17] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. "Materialized Views Selection in a Multidimensional Database." In: *VLDB*. Vol. 97. 1997, pp. 156–165.

[18] *Big Data Benchmark*. https://amplab.cs.berkeley.edu/benchmark. URL.

[19] Peter A. Boncz, Marcin Zukowski, and Niels Nes. "MonetDB/X100: Hyper-Pipelining Query Execution". In: *CIDR*. 2005, pp. 225–237.

[20] *CasJobs*. http://skyserver.sdss.org/casjobs/. URL.

[21] Surajit Chaudhuri and Vivek R. Narasayya. "An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server". In: *VLDB*. 1997, pp. 146–155.

[22] Edgar F. Codd. "A Relational Model of Data for Large Shared Data Banks". In: *Communications of the ACM* 13.6 (June 1970), pp. 377–387.

[23] *Conviva*. https://www.conviva.com. URL.

[24] Carlo Curino et al. "Schism: a workload-driven approach to database replication and partitioning". In: *PVLDB* 3 (2010), pp. 48–57.

[25] D. Abadi *et al.* "Integrating Compression and Execution in Column-oriented Database Systems". In: *SIGMOD*. SIGMOD. 2006, pp. 671–682.

[26] D. Abadi *et al.* "The Design and Implementation of Modern Column-Oriented Database Systems". In: *Foundations and Trends in Databases* 5.3 (2013).

[27] D. Ślęzak *et al.* "Brighthouse: An Analytic Data Warehouse for Ad-hoc Queries". In: *PVLDB* 1.2 (2008), pp. 1337–1345.

[28] Avrilia Floratou et al. "Column-oriented Storage Techniques for MapReduce". In: *PVLDB* 4.7 (2011).

[29] Michael R. Garey and David S. Johnson. *Computers and Intractability*. 1990. ISBN: 0716710455.

[30] Himanshu Gupta and Inderpal Mumick. "Selection of views to materialize under a maintenance cost constraint". In: *ICDT*. 1999, pp. 453–470.

[31] H. Koga *et al.* "Fast agglomerative hierarchical clustering algorithm using Locality-Sensitive Hashing". In: *Knowledge and Information Systems* 12.1 (2007).

[32] *Hadoop*. http://hadoop.apache.org. URL.

[33] Venky Harinarayan, Anand Rajaraman, and Jeffrey D Ullman. "Implementing data cubes efficiently". In: *SIGMOD Record*. Vol. 25. 2. 1996, pp. 205–216.

[34] *IBM Netezza*. http://www.netezza.com.

[35] Stratos Idreos, Martin L Kersten, and Stefan Manegold. "Database Cracking". In: *CIDR*. 2007, pp. 68–78.

[36] Stratos Idreos, Martin L. Kersten, and Stefan Manegold. "Self-organizing Tuple Reconstruction in Column-stores". In: *SIGMOD*. 2009, pp. 297–308.

[37] J. Han *et al.* "Frequent pattern mining: current status and future directions". In: *DMKD* 15.1 (2007), pp. 55–86.

[38] Daxin Jiang, Chun Tang, and Aidong Zhang. "Cluster analysis for gene expression data: a survey". In: *IEEE TKDE* 16.11 (2004).

[39] Alekh Jindal et al. "A Comparison of Knives for Bread Slicing". In: *PVLDB* 6.6 (2013), pp. 361–372.

[40] G. Karypis et al. "Multilevel hypergraph partitioning: applications in VLSI domain". In: *IEEE Trans. on VLSI* 7.1 (1999), pp. 69–79.

[41] M. Armbrust *et al.* "Spark SQL: Relational Data Processing in Spark". In: *SIGMOD*. 2015, pp. 1383–1394.

[42] M. Grund*et al.* "HYRISE: A Main Memory Hybrid Storage Engine". In: *PVLDB* 4.2 (Nov. 2010), pp. 105–116.

[43] M. Stonebraker *et al.* "C-Store: A Column-oriented DBMS". In: *VLDB*. 2005, pp. 553–564.

[44] M. Zaharia *et al.* "Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing". In: *NSDI*. San Jose, CA, 2012, pp. 2–2.

[45] M. Zukowski *el al.* "DSM vs. NSM: CPU Performance Tradeoffs in Block-oriented Query Processing". In: *DaMoN*. 2008, pp. 47–54.

[46] Sara C. Madeira and Arlindo L. Oliveira. "Biclustering Algorithms for Biological Data Analysis: A Survey". In: *IEEE Trans. on Computational Biology and Bioinformatics* 1.1 (2004), pp. 24–45.

[47] P. Miettinen et al. "The Discrete Basis Problem". In: *IEEE TKDE* 20.10 (2008), pp. 1348–1362.

[48] Guido Moerkotte. "Small Materialized Aggregates: A Light Weight Index for Data Warehousing". In: *VLDB*. 1998, pp. 476–487.

[49] *Oracle*. http://docs.oracle.com/.

[50] *Postgres*. http://www.postgresql.org/docs/.

[51] R. Hankins*et al.* "Data Morphing: An Adaptive, Cache-conscious Storage Technique". In: *VLDB*. Berlin, Germany, 2003, pp. 417–428.

[52] Anand Rajaraman and Jeffrey David Ullman. *Mining of massive datasets*. Cambridge, 2012.

[53] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. 3rd ed. 2003.

[54] Ravishankar Ramamurthy, David J. DeWitt, and Qi Su. "A Case for Fractured Mirrors". In: *VLDB*. 2002, pp. 430–441.

[55] Jun Rao et al. "Automating physical database design in a parallel database". In: *SIGMOD*. 2002, pp. 558–569.

[56] S. Agarwal *et al.* "Automated Selection of Materialized Views and Indexes in SQL Databases". In: *VLDB*. 2000, pp. 496–505.

[57] S. Agarwal *et al.* "BlinkDB: queries with bounded errors and bounded response times on very large data". In: *EuroSys*. 2013, pp. 29–42.

[58] S. Agrawal *et al.* "Integrating Vertical and Horizontal Partitioning into Automated Physical Database Design". In: *SIGMOD*. 2004, pp. 359–370.

[59] S. Melnik *et al.* "Dremel: interactive analysis of webale datasets". In: *PVLDB* 3.1-2 (2010), pp. 330–339.

[60] S. Papadomanolakis *el al.* "AutoPart: Automating Schema Design for Large Scientific Databases Using Data Partitioning". In: *SSDBM*. 2004, pp. 383–392.

[61] Felix Martin Schuhknecht, Alekh Jindal, and Jens Dittrich. "The Uncracked Pieces in Database Cracking". In: *PVLDB* 7.2 (Oct. 2013), pp. 97–108.

[62] Jianbo Shi and Jitendra Malik. "Normalized Cuts and Image Segmentation". In: *IEEE TPAMI* 22 (1997), pp. 888–905.

[63] *Sloan Digital Sky Surveys*. http://www.sdss.org. URL.

[64] Liwen Sun et al. "A Partitioning Framework for Aggressive Data Skipping". In: *PVLDB* 7.13 (Aug. 2014), pp. 1617–1620. ISSN: 2150-8097.

[65] Liwen Sun et al. "Fine-grained partitioning for aggressive data skipping". In: *SIGMOD*. 2014, pp. 1115–1126.

[66] Liwen Sun et al. "Skipping-oriented Partitioning for Columnar Layouts". In: *PVLDB* 10.4 (2016), pp. 421–432.

[67] *TPC-H*. http://www.tpc.org/tpch. URL.

[68] V. Raman *et al.* "DB2 with BLU Acceleration: So Much More than Just a Column Store". In: *PVLDB* 6.11 (2013), pp. 1080–1091.

[69] J.H. Ward. "Hierarchical Grouping to Optimize an Objective Function". In: *J. American statistical association* 301 (), pp. 236–244.

[70] Reynold S. Xin et al. "Shark: SQL and Rich Analytics at Scale". In: *SIGMOD*. 2013, pp. 13–24.

[71] Y. He *et al.* "RCFile: A fast and space-efficient data placement structure in MapReduce-based warehouse systems". In: *ICDE*. 2011, pp. 1199–1208.

[72] Y. Huai *et al.* "Understanding Insights into the Basic Structure and Essential Issues of Table Placement Methods in Clusters". In: *PVLDB* 6.14 (2013).

[73] Yin Huai *et al.* "Major technical advancements in Apache Hive". In: *SIGMOD*. 2014, pp. 1235–1246.

[74] Z Liu *et al.* "JSON Data Management: Supporting Schema-less Development in RDBMS". In: *SIGMOD*. New York, NY, USA, 2014, pp. 1247–1258.

[75] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. "BIRCH: An Efficient Data Clustering Method for Very Large Databases". In: *SIGMOD*. 1996, pp. 103–114.

[76] Jingren Zhou, Nicolas Bruno, and Wei Lin. "Advanced partitioning techniques for massively distributed computation". In: *SIGMOD*. 2012, pp. 13–24.

[77] Jingren Zhou and K.A. Ross. "A multi-resolution block storage model for database design". In: *IDEAS*. 2003, pp. 22–31.