Master Thesis

Self-Organizing Data Layouts for Databricks Delta

Author: Adriana Tufă

1st supervisor:	Peter Boncz	
2nd reader:	Alexandru Uță	
daily supervisor:	Adrian Ionescu	(Databricks)

A thesis submitted in fulfillment of the requirements for the VU Master of Science degree in Parallel and Distributed Computer Systems

August 16, 2019

Abstract

Storing large amounts of data has become increasingly accessible over the years and Big Data systems grew popular for easy storage and processing. However, compared to warehousing, in Big Data, information is often diverse and stored without a precise goal. This leads to more varied and complex query workloads, in order to extract meaningful information. To make analytical queries on such large amounts of data possible, it is important to avoid operating on big volumes. Data skipping is one technique that uses file metadata, like minimum and maximum values, in order to prune the data that needs to be read. However, efficient data skipping relies on underlying data being clustered. Due to the varied nature of Big Data, upfront storage optimization is difficult. With this project we automate the data layout optimization, based on patterns extracted from the workload. We implement our solution for Databricks Delta, which uses an in-house version of Apache Spark.

We first explore a physical design mechanism that improves data skipping, based on workload analysis. Then, we implement an automatic optimizer that periodically reorganizes the data according to the characteristics of the workload. The main challenges lie in extracting the relevant features from the workload and implementing this type of optimizer in a distributed Big Data system where monitoring and data access capabilities are restricted by design. Lastly, we build an evaluation framework that outputs representative results over real data and workloads. We consider the optimizer successful if it correctly detects when data re-organizing is beneficial and improves query processing to a degree that outweighs the associated costs. We obtained up to 3.2x speedup compared with no data optimization.

Contents

1
Introduction

1.1 Motivation

Modern storage systems can ingest data from a variety of sources. Often, especially in the case of data lakes, diverse and complex data is stored in its raw format for later analysis. Therefore, it is highly unlikely that its layout is naturally organized in a manner that fits the upcoming workloads. With this project, we explore automatic data layouts that are optimized according to the characteristics of the workload, in order to maximize *data skipping* capabilities. Our target is modern distributed storage and processing systems, specifically Spark (1) and Databricks Delta (2).

In Big Data, automatic data layout is not a very developed topic yet. The high volumes and variety of data makes it impractical to apply data re-organizing or pre-processing techniques used in traditional relational databases systems (e.g. materialized views). Moreover, the purpose of the data is not known beforehand. Thus, data can be viewed from multiple different perspectives and workloads may be changing over time, making it hard to commit to a static physical design structure. The problem of efficiently answering multidimensional queries has been tackled by OLAP (3) systems by performing *multidimensional clustering* (MDC) (4). This places together data with similar values across several dimensions. Then, query execution exploits this locality to read a smaller amount of relevant data, using data skipping techniques. These match filter predicates on data statistics (like minimum and maximum column values), to avoid fetching unnecessary data into the query pipeline, only to be filtered later.

However, MDC or any other physical optimization adopted in existing database systems requires human intervention. Unfortunately, manual supervision is prone to omissions or inaccuracies. DBAs may lack the time and necessary skills to constantly keep the data layout updated. Therefore, relational database systems have tried to automate the data layout optimization but limited themselves to only integrating wizards, such as DB2's Advisor (5) or Microsoft's Autoadmin (6), that provide advice on the physical design

alternatives. This, of course, still needs an administrator behind the wheel who ultimately picks the optimizations which are to be performed.

With the shift towards Big Data platforms, the architecture changed as well. Modern analytics platforms are hosted in the Cloud which needs to take over the role of DBAs. Clients expect to eliminate the need of human intervention and trust that providers can safely manipulate their data in order to increase performance. Indeed, our initial workload analysis on Databricks Delta tables (Section 4) indicates there is potential for automatic improvement by reorganizing data layouts.

Another change that comes with the Cloud platforms is the decoupling of storage from processing. While in classical databases the data format can be custom shaped and can exploit locality, in Cloud, stored data has to comply to certain restrictions. For example, by using storage systems like S3 (7), each write generates a new object. In the case of continuous data ingestion, especially when it comes in small amounts, each addition generates a different file, along with its metadata. This can quickly create *many small files*, which are expensive to operate on (e.g. only listing them can become a bottleneck). In such situations, a compaction strategy is needed. Our data science results show potential for improvement in this area as well, as most tables contain a large number of small files.

This project proposes a method of eliminating the human factor from taking physical design decisions in modern Big Data systems. We try to find the optimal balance between data clustering and storage granularity and enforce it automatically, in a way that adapts to both the incoming data and the evolving workload. By doing so, we should improve data skipping and thus result in more efficient scans and filters. In Delta, MDC is already available in the form of Z-ordering (8) but the optimization process is entirely manual - both attribute selection and trigger. We seek improvements along three main directions:

- a better clustering scheme, covering a larger and more diverse set of predicates
- automatic detection of relevant predicates
- automatic triggering of the optimization process

Finally, we assess the overall workload improvement we can bring using an evaluation framework that is based on real-world data and queries.

1.2 Background

1.2.1 Apache Spark

The Spark (1) project was created as an engine for distributed data processing, bringing new alternatives to the MapReduce batch processing paradigm. It was developed in 2009 at Berkeley and later was donated to the Apache foundation. Its initial purpose was to provide in-memory data manipulation, which enables much faster computations, suitable for batch but also iterative and interactive applications. Later, Spark grew considerably, extending its applicability to machine learning (9), graph processing (10), structured streaming (11), SQL (12), while also improving performance with projects like Catalyst (13) and Tungsten (14).

Spark's data model is based on *Resilient Distributed Datasets* (RDDs). These in-memory data structures split data into partitions, which allow parallel processing. Depending on the operations, partitions are either operated on separately (e.g. filter, map), or they need to be combined (e.g. joins, aggregations), which incurs network communication and higher latencies. RDDs are the building blocks of Spark but they provide a low level API. Spark SQL (12) is a module that introduces a higher level data abstraction and lets one express the operations without being aware of implementation details.

Spark SQL models data into DataFrames (15) and implements a query engine that operates similarly to a relational database. It can be used either with SQL queries or through DataFrame API, in multiple languages. In contrast to the RDD API, in Spark SQL operations are passed through a query optimizer, which picks the best execution plan. This is ultimately translated to operations on RDDs and scheduled on the cluster. A query plan consists of a tree structure of operators, such as Scan, Filter, Aggregate, etc. One relevant optimization to our project is *pushed down predicates* - which are filters that appear in some query operators in the query plan, but are pushed down to the Scan level, in order to apply them as soon as possible and reduce the size of the read data from the beginning. We leverage these pushed down filters in order to optimize the data layout.

In Spark, the distributed architecture consists of a Driver and multiple Executors, which run in parallel on a cluster. The Driver is the starting point of the application, which processes the input commands and creates the execution plan. Then, the Driver delegates work to the available Executors. Spark needs an external cluster manager which can schedule and map these processes to cluster nodes.

Another particularity of Spark is that it is responsible only for the computation. The storage layer has to be managed separately and can come from multiple sources (HDFS

(16), Cassandra (17), etc). However, fetching data from an external source generates an important I/O overhead, especially in the case of big datasets. One way to reduce the impact of this is to use efficient file formats, that use compression and, moreover, allow techniques like *predicate pushdown* and *partition pruning*: avoiding to read certain input files, because they are data partitions whose partitioning criterion implies that the predicate cannot select data from them. One such popular file format is Parquet.

1.2.2 Parquet File Format

Apache Parquet (18) is a file format for storing tabular or nested tabular data. Its columnar data layout format helps achieve high performance when only a subset of columns is read. This happens in OLAP (3), where data can be viewed from multiple perspectives and queries may need small, different attribute selections out of the whole dataset. In these situations, reading whole rows produce waste in terms of both memory and I/O. Parquet solves this problem by using a pseudo-columnar format in which data is first partitioned horizontally into row-groups, and inside each row-group, values are stored in column-first order. When reading only a number of columns, Parquet exploits data locality much better than the traditional row-major formats, as the relevant data is contiguous in memory.

Parquet format is structured as a collection of files under one directory. It also allows to partition the data on specific columns. A partition column generates a new directory for each distinct value. Partitioning by multiple columns generates a hierarchy of directories, where their number and distribution depend on the partitioning order. The role of partitioned parquet files is to filter on partitioning columns and read a smaller amount of data. Partition pruning is a form of data skipping and is commonly used in Spark, alongside other techniques discussed in the following sections.

1.2.3 Databricks Delta

Databricks (19) is a unified analytics platform powered by Spark whose aim is to ease the analytic processes for its customers. It uses its own flavour of Spark, referred to as *Databricks Runtime* (DBR), which brings new features and improved performance compared to its open source version. One relevant improvement is that it allows Spark to run in a cloud setting, where the computation happens on clusters started on-demand and is separated from the storage layer. DBR also uses cloud-specific opimizations, like autoscaling and Parquet cache (DBIO (20)) on local temporary compute storage. Running in Cloud also allows Databricks to monetize each customer's usage easier. While certain services run on the Databricks account, the main compute and storage resources are spent

on customer accounts. Our system implementation runs on the customer side as well, so it incurs additional costs.

One significant addition is Databricks Delta (2), which recently has been released in the open source world as Delta Lake (21). Delta is basically a new storage layer that introduces Delta Tables as its file format. Their main purpose is to provide ACID transactions. This, in turn, allows tables to reliably act as either sinks or data sources for batch or streaming jobs. A Delta table is stored in Parquet data files but, in addition, it has a *Delta Log* that provides a consistent view of the table. The log stores metadata about the transactions. Each commit creates a new table snapshot and updates the log with information on the files that were added or removed. For each added file, the log keeps statistics, such as name, size and minimum and maximum values for a number of columns in the table schema. Delta leverages these file statistics for data skipping. In this project, we also make extensive use of file level statistics stored in the log.

Delta has seamless integration with Spark. Below is an example of creating and reading Delta tables using DataFrame API in Scala:

```
val df = spark.range(1000).withColumn("parity", $"id" / 2)
df.write.format("delta").saveAsTable("exampleTable")
val data = spark.read.format("delta").table("exampleTable")

// from here on, "data" is automatically updated when the table
is modified from any other source
```

Listing 1.1: Writing and Reading a Delta Table from Scala DtaFrame API

Every time a query is performed on a Delta table, the log is traversed in order to obtain an up to date snapshot. Thus, any changes to the table contents should be made through Delta API such that they are recorded in the transaction log and the Delta protocol can ensure transactionality.

1.2.4 Data skipping

Data skipping is a technique used at query time with the goal of reducing the volume of data read from persistent storage. This, of course, cuts unnecessary I/O and improves overall running time. Data skipping uses pushed down filters matched against storage level metadata, in order to scan only the portions of the data that are relevant to the query. Data skipping can have multiple granularities. We discuss skipping in relation to the Parquet format and Databricks Delta.

Table partition skipping. Started as a Hive (22) feature, table partitioning means perfectly splitting a dataset based on the values of a number of columns. We mentioned this possibility for the Parquet format, but it is implemented for other types of files as well. When reading a partitioned table, we can mention filters on the partitioning columns, which determines reading only data that has the required values.

The example in 1.1 is a table partitioned by *month* and *region*. The first directory level is based on the values of *month* column. The directories actually embed the column name and the associated values. For each distinct value for *month*, a second layer of directories is created, this time for the distinct values of *region* attribute. Inside each subdirectory there are a number of files, storing data corresponding to the two specified values for *month* and *region*.

```
|---month=0
|   |---region=APAC
|   |       .part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet.crc
|   |       part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet
|   |---region=EMEA
|   |       .part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet.crc
|   |       part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet
|   |---region=US
|           .part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet.crc
|           part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet
|---month=1
|   |---region=APAC
|   |       .part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet.crc
|   |       part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet
|   |---region=EMEA
|   |       .part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet.crc
|   |       part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet
|   |---region=US
|           .part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet.crc
|           part-00000-ce13ffb6-33f1-443e-8d35-f9a7271544e1.c000.snappy.parquet
...
```

Figure 1.1: Directory structure of a partitioned Parquet format table

Partition pruning happens when there is a query with filters on one or both partitioning columns. E.g. `SELECT * from table where month=2` would read only files under *month=2* directory. However, when having only a filter on region, all *month* partitions need to be scanned and skipping happens only at the second level of partitioning. Therefore, the order of partition columns need to be well aligned with the workload for optimal performance.

Data skipping within Parquet files. Parquet files keep metadata about the row groups, which can be leveraged for skipping. Specifically, it can include information like *minimum and maximum* values for the columns. These statistics can be used in what is

called min-max skipping (or range skipping). In 1.2 we exemplify how data skipping works in this situation:

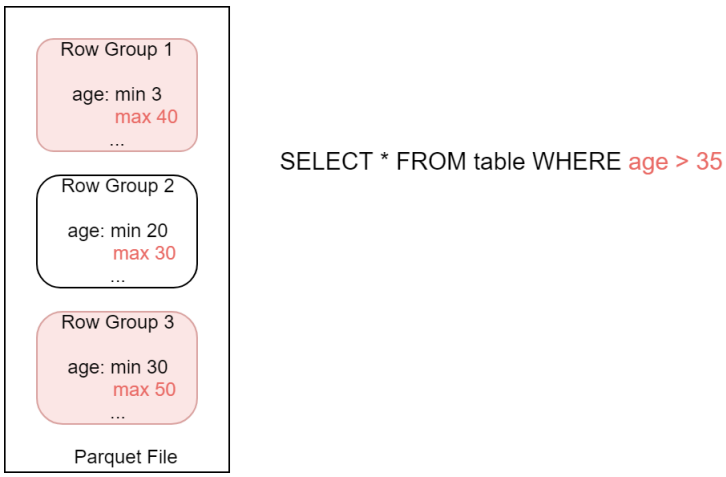


Figure 1.2: Min-max skipping on Parquet row-groups

Considering a table saved as Parquet format that contains, among others, an *age* column, each row group may store statistics on this column. When issuing a query with a filter on age, only row groups that satisfy this filter are read. In the example, row groups with a maximum value for age lower than 35 are skipped from the start.

Skipping in Databricks Delta. Similar to Parquet metadata, the Delta Log maintains statistics on minimum and maximum values for each file. Consequently, they are used for file level *min-max skipping*. Before running a query plan, Delta matches the pushed down filters on the available statistics from the log and eliminates those files that do not qualify. However, not all filters are eligible for this type of filters. Min-max values are effective only in range comparisons. Therefore, the types of predicates that can generate skipping are in the following formats:

```
column <,>,= expression
column startsWith expression
column in (<list of expression>)
```

where expression can be evaluated to constant literal.

1.2.5 Layout Optimization

Data skipping has increased efficiency when the underlying data is sorted or clustered. With table partitioning, data is perfectly split by its partition columns. With range skipping, when the tuples are not sorted, we can end up with the min-max ranges so wide

that no skipping can be actually done. Ideally, each block of data, where a block refers to the data skipping granularity we use, has non-overlapping values with other blocks. The problem of grouping the tuples has a number of solutions, each with its pros and cons.

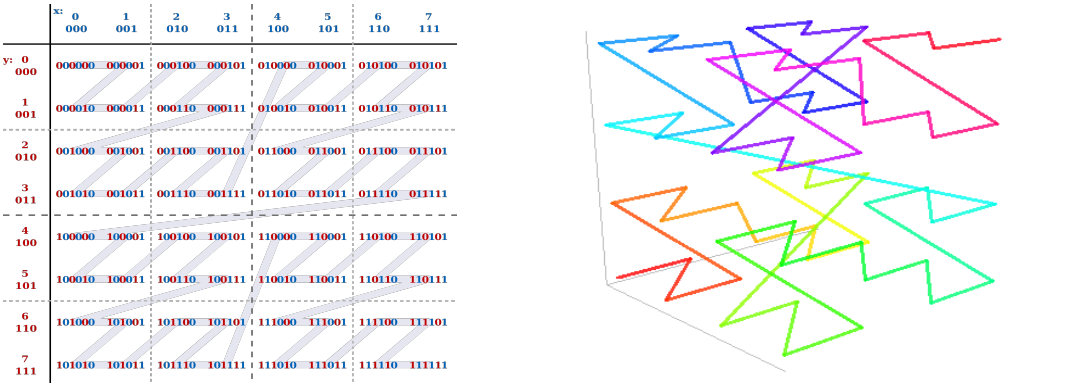
Horizontal partitioning. One approach is horizontal partitioning, which implies splitting the rows in multiple groups. This is usually implemented as range-partitioning or hash-partitioning. Range-partitioning breaks the rows in ordered intervals based on one column. Then each resulting block can maintain statistics useful for data skipping. Hash partitions are obtained by applying a hash function on all specified partitioning columns and grouping by that result.

Table partitioning. Another partitioning approach is the one offered by Hive-style table partitioning. In Spark, this can be applied at table creation time (23) (using partitionBy). This scheme works well for a small number of columns but for many dimensions, it can result in a huge number of files which are too expensive to list and keep metadata on. E.g., partitioning by 4 dimensions, each with 200 unique values, results in 1.6 billion partitions of one or more files each.

Major-minor sorting. A straightforward clustering technique is major-minor sorting by one or more columns. Here, the order of columns is extremely important as the first column gets fully sorted while the following dimensions are ordered only in groups that correspond to the same unique combination of values in the previous columns. This results in more fragmentation for trailing dimensions and decreases the sorting and skipping efficiency.

Multi-dimensional clustering. MDC attempts to eliminate this shortcoming of sorting by giving equal weight to each contributing dimension. It was proven that MDC can bring notable improvements for complex workloads (4, 24, 25). Although logically it implies placing the data in a multidimensional cube where related tuples are close to each other, physically it is common to be reduced as some type of partitioning (8) or indexing (26). One way of performing MDC is by using space filling curves, like Z-Order (27) or Hilbert (28) curves. Currently, Delta uses Z-Order, which we discuss in more detail in the next subsection. The clustering technique proposed in this thesis is integrated with the Z-Order solution, but is expected to work well for any other space-filling curve.

Clustering in Delta. Delta has a dedicated OPTIMIZE command (29) whose purpose is twofold: compaction and clustering. Used by its own, it compacts small files ($<$ 1GB) into bigger files of default size (1 GB). 1GB files help in reducing the "many small files" problem that can be very expensive when listing and accessing files from a remote source,



(a) Two dimensional Z-Order (31) (b) Three dimensional Z-Order (32)

Figure 1.3: Example of Z-Order curve

like S3. The second usage of the command is in OPTIMIZE ZORDER BY (COLS) format which clusters the data across the specified columns.

However, we identified a number of shortcomings of this approach, which this project should solve:

- Customers need to monitor their performance and detect when the data layout is a source of improvement. Moreover, they need to be aware that the dataset can be compacted or clustered by the OPTIMIZE command.
- Once such a situation is identified, customers need to manually trigger the optimization.
- When Z-Ordering, one must know their workload to some degree in order to make the right choice for clustering dimensions.

1.2.6 Z-Order and Skipping Oriented Partitioning

In this section we describe two data clustering approaches that are relevant in this thesis. The first one, *Z-Order*, is the current algorithm used for data layout optimization in Databricks Delta. The second, *Skipping Oriented Partitioning* (SOP) (30) is a different method described in literature, that we are using in this project.

Z-Order is a space filling curve (27) that traverses data across multiple dimensions while preserving data locality. It follows a "Z" pattern which ultimately maps multiple dimensions to a single one. Figure 1.3 depicts the curves for two and three dimensions.

The order is determined by interleaving the bits of the specified dimensions and sorting the resulting values (see Fig.1.3a). Z-ordering a dataset in DBR means reordering data based on the interleaved values of a number of attributes. Then, the ordered tuples are split

evenly into files and written back to disk. This new order preserves, to some degree, data locality across all dimensions. Combined with file level statistics, this results in smaller min-max ranges, which are more useful for data skipping. In Fig.1.3a, each rectangle would represent a file. As such, the min-max values for x and y are perfectly arranged and non-overlapping.

Skipping Oriented Partitioning is a clustering approach designed specifically for increasing data skipping on a given workload. The clustering input consists of features, which are predicates or conjunctions of predicates. A few examples of SOP features can be the following:

```
'comment' like "% cluster %"
length(stringReplace('message', "-", "")) < 10
datediff(fromunixtime('timestamp'), 3) = 4
udf('type')
```

Compared with Z-Order, predicates in SOP can take any format, as long as they are evaluated to a boolean values. This also implies that min-max skipping is not possible in this case.

The SOP paper (30) describes both the clustering algorithm, and a method for choosing the features that are most suitable for the algorithm. We focus only on the clustering approach.

Given a set of features and a workload, SOP clusters data such that it maximizes skipping potential for the features on that workload. That is, it takes into account the weight and selectivity of each feature when creating the clusters. In contrast to Delta's min-max file statistics, in SOP, skipping is done based on binary values, where a 1 represents that there are tuples that satisfy a specific feature, and 0 that there are not, so the file can be skipped. Therefore, a file has an associated bit vector of n elements, each bit corresponding to one input feature. This is achieved by first augmenting the dataset with bit vectors, which store the evaluation of the chosen features for each tuple, and then clustering the vectors with a bottom-up clustering approach which takes the workload into account. Finally, each cluster is written to a file and its corresponding bit vector is the result of OR'ing all the vectors in that cluster. Below we briefly describe the steps used in the clustering algorithm:

Let $F = \{F_1, F_2, \ldots, F_m\}$ be the set of chosen clustering features. For each F_i we compute w_i - the weight of F_i in the workload. For simplicity, it is enough to consider the weight as the frequency of F_i among the filter predicates.

Each tuple is augmented by a m-dimensional bit vector representing the evaluation of the features in F. So, in the beginning, there are n m-dimensional bit vectors, where n is the size of the dataset.

Initially, each tuple is its own partition. Here a *partition* refers to one SOP cluster, that will be written to a separte file in the end. Although it is still a way of grouping data, it is different from table/Parquet partitions which we have discussed earlier. To avoid ambiguity, we will use the terms *SOP partitions* to denote SOP clusters and *partitions* as general table partitions. At each step, the two SOP partitions that improve skipping the most are merged. Considering that at some point in time, the partitioning scheme is $P = \{P_1, \ldots, P_k\}$, we define the following terms. Each P_i has an associated bit vector:
$\underline{v}(P_i)$ = the bit vector for P_i - the result of OR-ing all the v_j in P_i
$\underline{v}(P_i)_j = 0$ means F_j prunes P_i , thus no tuple in P_i satisfies F_j, so P_i can be skipped. If $\underline{v}(P_i)_j = 0$, it means that for F_j we can skip $|P_i| * w_j$ tuples.

Based on this, the cost of one SOP partition is defined as:

$$C(P_i) = |P_i| * \sum_{j=1}^{m} (w_j * (1 - \underline{v}(P_i)_j))$$

The cost is basically the sum of tuples that can be skipped for each feature in F.

The total cost of the partitioning scheme at some point is is the sum of costs over all partitions:

$$C(P) = \sum_{i=1}^{k} (C(P_i))$$

Therefore, each merge step should maximize the total cost. This is basically achieved by obtaining as many zeroes as possible for features with the highest weights.

1.3 Related Work

Physical Database Design. Several physical design techniques have been employed over the years in order to improve query execution times. The most popular ones are indexes, partitioning and materialized views.

We consider classic indexes (B-trees (33), bitmaps (34) and other formats (35)) as being unidimensional physical clustering, thus not able to provide improvement for multiple dimensions simultaneously. However, they can be redesigned and used in MDC as described in IBM's DB2 approach (26). Here each unique combination of values is stored in a different block which is referenced by block indexes. In order to extract data based on the cube's

dimensions, block indexes are AND-ed/OR-ed to obtain the related blocks. Moreover, they try to avoid a prohibitively big amount of unique combinations by experimenting with different levels of coarseness for each dimension. One disadvantage of this approach is that physical space may be wasted as every block has the same number of pages, which leads to imbalance in skewed data. Another approach which rethinks indexes is database cracking (36), whose main advantage is that it is not based on past workloads, but rather on real-time tuning based on the input queries. Each query is seen as a suggestion to split the columns into smaller pieces. A cracker index keeps track of the current layout and helps gather the data needed in queries. However, cracking was only proved to work well in RAM. In a distributed system, its gradual reorganization is very expensive.

Many workload-driven physical design solutions are based on horizontal (37) or vertical partitioning (38, 39, 40) or a combination of the two (30, 41). One such approach described in (30) differentiates in that the partitioning dimensions are not columns, but features extracted as whole predicates or conjunction of filters that are frequent in the workload. They also exploit the fact that certain filters are relevant only to a subset of columns and thus the data is partitioned vertically (column groups) and horizontally as well.

Materialized views (42) is another frequent technique (5, 6) that pre-computes the results of relevant queries, thus eliminating a possibly big processing cost from query execution time. However, this comes at the price of more storage and more expensive updates.

Automated physical design tools. All major commercial database systems provide tools that analyse the workloads and output advice on how a DBA can improve the physical design. DB2 implements an advisor (5) capable of recommending indexes, partitioning, materialized views and MDC dimensions. Their system outputs candidates for each type of optimization and then assesses the benefit brought by each one by virtually applying the top recommendations. Particularly for MDC, the dimensions are chosen based on a sample workload and data (43). All attributes are reduced to different degrees of coarseness are given a best and worst improvement score. Then, a genetic algorithm performs the search of the output dimensions based on the score. Microsoft has their own advisor (44) for recommending range partitioning, indexes and materialized views (45). Their approach is based on a system called Alerter (6) that periodically analyses the situation and for potential improvements computes a lower and upper bound of the possible benefit. Oracle (46) runs analyze commands to collect statistics and use them for query optimization or to help users decide what columns to index or sort.

Cloud storage systems. Current cloud solutions use partitioning and clustering on columns specified by the user. Snowflake (47) uses micro-partitions as table layout and

automatically sorts them by clustering keys on relevant dimensions. The user can also specify an expression over a column to reduce its cardinality. Although the user can look at how well a column is already clustered by computing a clustering depth index (max depth of overlapping partitions), he still has to take the important decision of specifying the right clustering keys in the best order. Similarly, BigQuery (48) uses clustered tables sorted by manually specified columns in which the order plays an important role. Amazon Redshift (49) runs clustering analysis which can be used as hints for further physical layout improvements.

1.4 Research Questions

The research questions we wish to answer with this project are the following:

Q1 *Can we obtain a good data layout based on Z-order and SOP clustering? How can we best combine them?*

Based on data science, we have hints that both the current Z-order and an approach similar to SOP (30) could bring benefits. Therefore, we investigate methods of combining the bottom-up clustering used in SOP with the Z-order algorithm such that the system can leverage the advantages of both, without losing efficiency for any of them.

Q2 *Given a workload, what are the most relevant features and how can we automatically extract them? Can we detect correlations between them and minimize the set of features accordingly?*

We will perform workload analysis on filter predicates. We can extract either dimensions (attributes) that are frequently used in range comparisons or whole predicates. Clustering on attributes is useful when they are compared against many different points in the filters, while predicates cover complex expressions like UDFs or regex expressions. Because dimensions or predicates can depend on each other, we aim to detect such correlations and pick only the ones that are completely independent. For relevant features that were not selected for partitioning, we can still maintain statistics that help data skipping.

Q3 *How can we enable data skipping on complex expressions?*

In Delta, min-max skipping is currently available only for attributes that are used in simple filters as described in the background section. It needs to be extended for other types of predicates as well. We also need to add additional structures for supporting the SOP scheme.

Q4 *When does it pay off to trigger automatic data layout optimization and how can that be achieved in a Big Data system where storage is decoupled from compute?*
Layout optimization is a costly operation so we need a good heuristic that can decide when it is beneficial to perform it. A starting point is to look at the volume of reads compared to the writes and only consider tables with large scans. We also need to decide on the frequency of optimizations. A check can be made upon every write and when enough new data has been written, we can schedule the optimization. Another decision point is when we detect that the workload has changed significantly. Furthermore, we need to find a suitable way of integrating and running the optimization in the current infrastructure.

Q5 *How can we meaningfully evaluate our optimizer?*
Real world data and workloads have patterns and correlations that are hard to simulate by benchmarking tools. Therefore, we need to build an evaluation framework that performs evaluation on real datasets and queries and outputs relevant metrics to our problem.

Considering the research questions, the contributions of this thesis are the following:

- A clustering algorithm that combines SOP with Z-Order in a way that preserves the benefits of both.
- A workload analyzer that given traces of a past workload, detects relevant features for the clustering algorithm.
- A cost-benefit estimation that is used to decide when layout optimization is necessary
- An end to end solution for Databricks Delta that performs automatic data layout optimization.
- A custom evaluation framework that works on real data and workloads

1.5 Thesis Outline

In this chapter we gave context on data layout optimization and laid out the research questions. Chapter 2 highlights the results of our initial data science. Chapter 3 presents a high level system architecture, in relation to Databricks Delta infrastructure. In chapter 4 we discuss the approach for clustering and feature selection. Chapter 5 presents the cost-benefit estimation and how it help integrate the automatic solution. Chapter 6 outlines the results on the custom evaluation framework and Chapter 7 concludes the research questions and discusses future work.

2

Hypothesis Validation through Data Science

Before proposing a solution, we want to verify our assumption that clustering would indeed bring benefits for Databricks workloads. We instrumented Databricks Runtime to provide aggregate statistics that answer high-level questions and gather data for a limited time on selected Databricks customer workloads. Our instrumentation does not need or involve access to customer data or detailed query features. Rather, we instrumented to answer our high-level research questions only. All data exploration was done alongside a few main questions presented below.

Question: *Are there clues that reorganizing data layout can improve current workloads?*

Filter selectivity Because clustering has a direct impact on data skipping, we looked at current skipping capabilities in relation with filter selectivity. Particularly, we compared the amount of scanned data to the amount of remaining data after applying all the filters in the query. For more relevant results, we reduced our experiment to only scans bigger than 500MB. Fig. 2.1 plots the cumulative percentage of the filtered amount. We can read some datapoints as follows:

50% queries have their predicates filter more than 75% of data.

44% queries have their predicates filter more than 90% of data.

These results indicate that there are many big scans with low selectivity filters that can benefit from improved skipping to reduce the amount of data read.

Files size. Next we examined the average files size across all tables. As reading many small files is problematic, compacting is a performance improvement opportunity that can bring even more benefits if data is clustered at the same time. Fig 2.2 shows that the vast

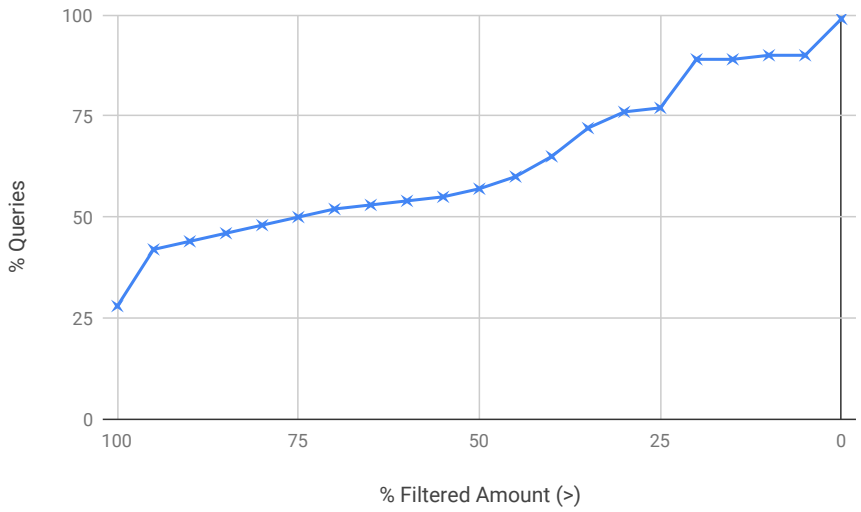


Figure 2.1: Cumulative query distribution based on the filtered amount

majority of tables have an average file size smaller than 50MB. So there is obviously room for improvement in terms of compaction. In order to determine the impact of files size, we conducted an experiment in which we compared the query time when reading 250GB written in files of 50MB versus files of 1GB. We obtained a 2x time improvement in the case of larger files.

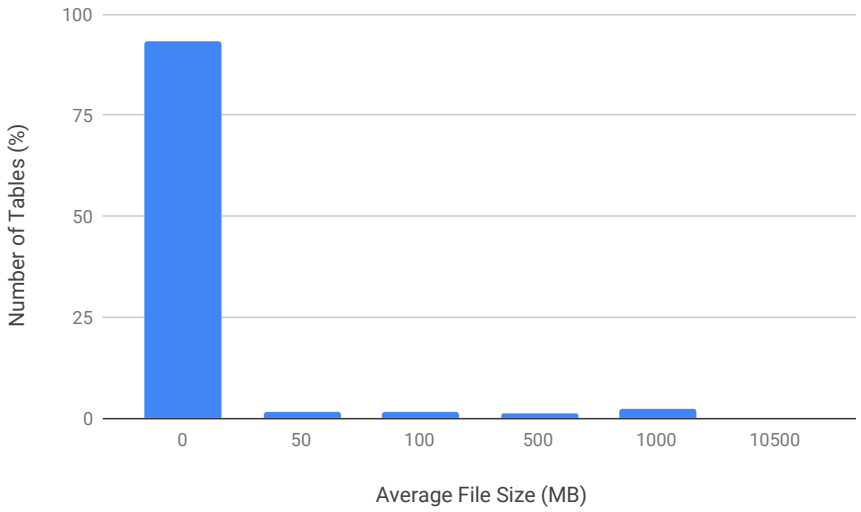


Figure 2.2: Percentage of tables by their average file size

Question: *Are there patterns in the workload that we can exploit when clustering?*

Predicate space. We examined the cardinality of distinct query filters, at table level. We excluded the predicates on partitioning columns. Fig. 2.3 reveals that most of the tables use only 1 or 2 distinct expressions. This makes them clear candidates for clustering.

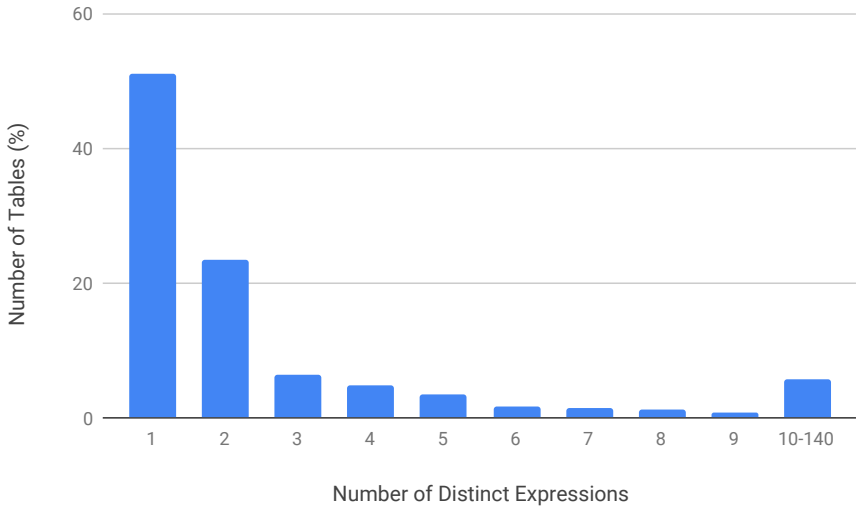


Figure 2.3: Percentage of tables by their number of distinct filter expressions

Frequent predicates. We then looked further to see if there are recurring predicate patterns. We extracted the occurrence rate of the top 4 most frequent attributes for each table. We can observe from Fig. 2.4 reveals that 40% of the tables use their most frequent attributes in more than 50% of the queries. This shows again that there are relevant candidates for clustering.

Moreover, the constant set of attributes over a longer period of time is an indication that workloads tend to remain constant. This suggests that clustering on a set of dimensions chosen from past workloads will also benefit future queries.

We analysed predicate frequency further, in relation with partition predicates, where a partitioning scheme was present. We compared the frequency of filters on partition columns vs the most frequent predicate on a different column. We found that following categories:

17.5% tables have partition filters more frequent than other filters

43.5% tables have partition filters as frequent as at least one other filter

39% tables have one other filter more frequent than partition filters

The results indicates that partitioning is not always the most reliable skipping method and that for a great majority of tables, workload information can reveal the potentially more relevant filtering dimensions, which can be clustering subjects.

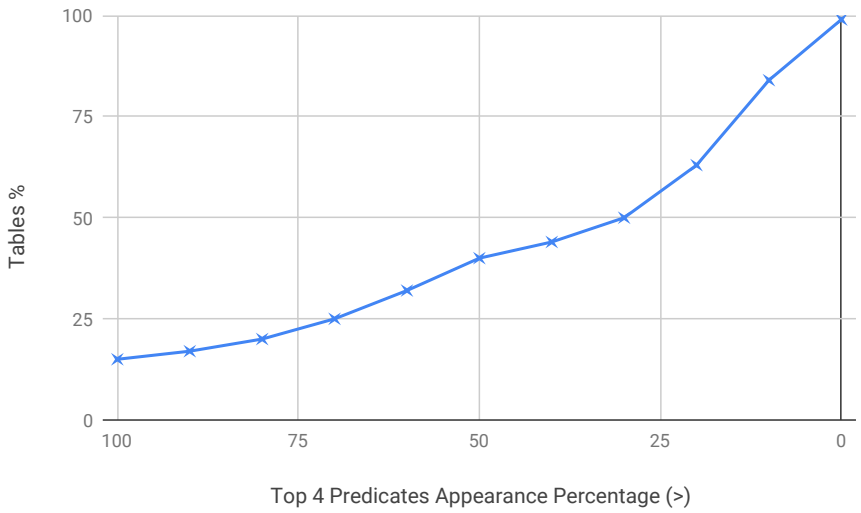


Figure 2.4: Cumulative plot of the appearance frequency of the top 4 most frequent predicates

Question: *Can the current clustering scheme be improved?*

Current Z-Order and file statistics impose an important limitation on the predicates that can leverage data skipping: Z-Order works only on leaf columns and skipping is done on binary comparisons between a column and a literal. Therefore, we looked at predicate formats to find out how many of them comply to these constraints and how many use more complex predicate expressions.

Complex expressions. Out of all scans, we counted 23.9% queries with simple binary comparison predicates which currently can leverage data skipping and 13.2% queries with other types of filters. Specifically, we define these "other" filters as complex expressions, which can be split into two categories: the ones that are in binary comparison formats, but instead of a plain column, they use an expression with function calls, and the ones that are in other formats that are directly evaluated to a true/false value (regex, UDF, spark boolean functions). The most frequent complex expressions use the following expression categories: casts, date related functions (fromunixtime, truncdate, etc), complex type

value extractors, string expressions, regex and UDFs. This result is relevant as Databricks Delta currently supports neither clustering nor skipping on complex expressions. Thus, it shows the skipping opportunity if complex expressions are accounted for.

Comparison literals. For predicates that are min-max skippable, we looked at the number of distinct literals each predicate is compared with (Fig 2.5). We observed that a striking majority of filters involve the same value in all comparisons. This indicates that those predicates can be used in their full format for clustering. Although scarce, there are also cases with more than 1000 different comparison values.

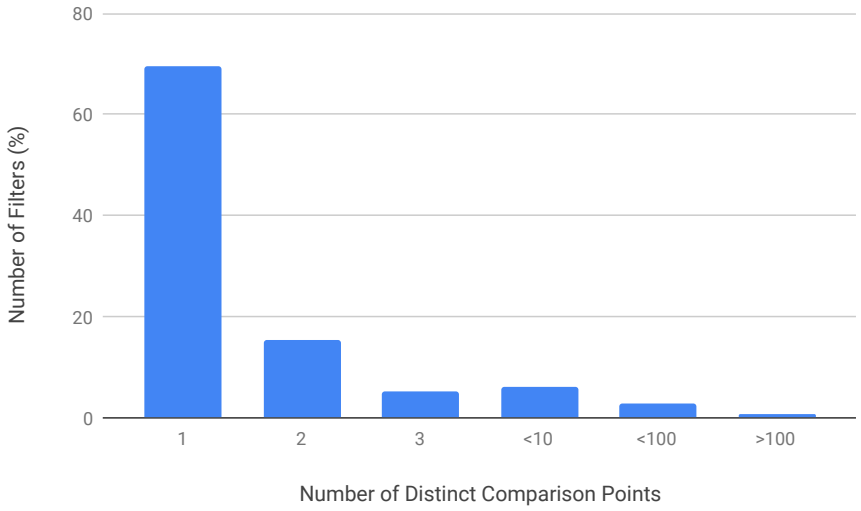


Figure 2.5: Binary comparison filter distribution based on numer of di□erent comparison points

To sum up data science findings, there are a few important takeaways that indicate that automatic physical layout makes sense, while also uncovering some potential directions for improvement. Firstly, the results show that workloads usually have a set of constant and frequent filters. They can make good clustering candidates but also reveal that predicates do not change abruptly, so optimizing a batch of data will benefit future workloads as well. Secondly, filter selectivity shows that queries would benefit from better data skipping, that can prune the amount of read data from the beginning. Moreover, layout improvement can, beside clustering, help reduce the many small files problem, by simply performing compaction. A relevant result for the rest of the thesis is that complex expressions make up a big portion of predicate filters and should be accommodated by the clustering algorithm and the data skipping mechanism.

3
Proposed System Architecture

The goal of the project is to achieve automatic data organization that fits the workload. Given the research questions and the current Databricks Delta infrastructure, we propose a system design that has three main goals:

- Clusters data more efficiently and allows for data pruning on filters that involve complex expressions.
- Monitors the workload and, upon request, outputs a list of relevant features for clustering. A *feature* denotes any predicate, column or complex expression that can be used for clustering.
- Automatically decides when it is optimal to perform clustering

Following the goals, we designed four modules. Used on their own, they can materialize each goal independently, but combined, they construct the final automatic data layout optimizer. All modules are run on the customer side, so they generate additional storage and compute costs. As such, one aspect we keep in mind in the design is to minimize the expenses while increasing the benefits. The four modules that are integrated in the Databricks Delta infrastructure are:

1. Clustering Module - is responsible with performing data layout reorganization, given a set of clustering features.
2. Query Log - records data about query filters as they are issued on the table.
3. Workload Analyser - is concerned with reading, parsing and interpreting workload information, in order to find those features that are good candidates for clustering.
4. Cost-Benefit Estimator - makes an estimation of the potential benefit as a function of clustering cost and gained improvement. Based on the result, it decides whether clustering is needed.

Clustering Module

The clustering component receives as input a list of files and a set of clustering features. Its purpose is to cluster the data from the given files, and write them back to persistent storage. In the process, it also compacts small files and splits data uniformly into equi-sized tuple ranges. The algorithm used for clustering is a combination between the current Z-Order and SOP. Furthermore, clustered files written back to the table have enriched metadata, stored in the Delta Log, that enables skipping for a broader type of predicates.

Query Log

The Query Log is itself a Delta table that is stored in the parent directory of the original table. Upon every query, the log adds information about the filter predicates and read schema. In order to avoid frequent small inserts in the log, we keep two levels of caching - an in memory queue which, when full, is spilled onto local disk files. These, in turn, are appended to the log at fixed time intervals.

Workload Analyser

When triggered, the workload analyser reads the query log information and extracts features for clustering. It searches the predicate space and picks a set of candidates as described in Chapter 4. Furthermore, it is able to detect correlations between features and eliminate redundancies accordingly.

Cost Benefit Estimator

The estimator is a key component in automating data layout organization. Since clustering is a time and resource expensive operation, we want to guarantee that it is performed only if its benefit outweighs the cost. Thus, the module uses a method of estimating the skipping improvement and data reorganization cost. Then, it uses the estimations to decide what data layout is better in the given point in time. If it decides that clustering is needed, it activates the clustering module. As of now, the cost-benefit estimator is automatically triggered upon every insert. The estimations are made only when enough new data has been inserted (currently the threshold is at 10GB).

Fig 3.1 depicts the system components and how they interact for a workload of queries and insertions. We start by describing the workflow for a selection query (red arrows). Once the query is received, its query plan is executed on the current table snapshot. At the end of the execution, the Query Log is updated with metadata about the filters used in the query. Finally, the result is returned to the issuing party.

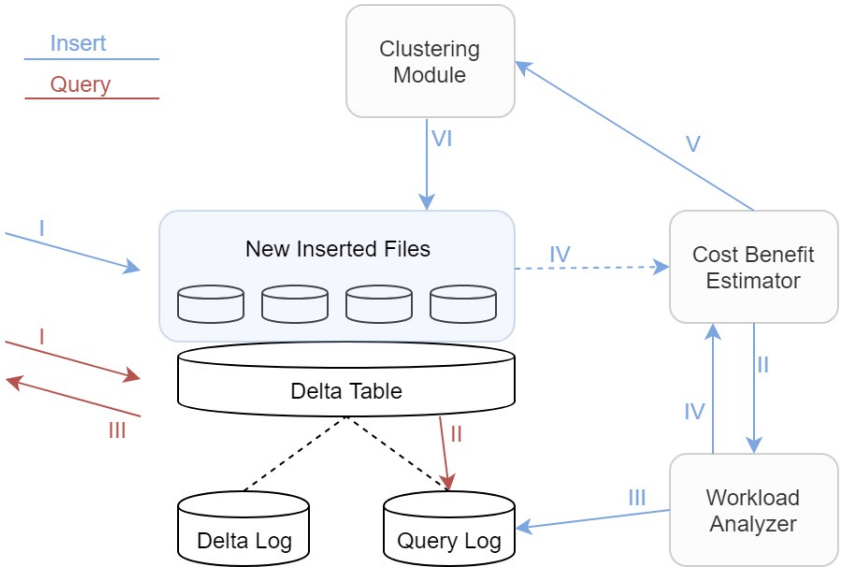


Figure 3.1: System and workflow diagram

The insertions are following a more complex workflow. Once data is appended to the table and there are enough unclustered files, the Cost Benefit Estimator is automatically triggered. It first asks the Workload Analyzer for relevant clustering features (II). This, in turn, reads the Query Log and uses its algorithm to extract relevant features, which are fed back to the Cost Benefit module. Having the features and also extracting a data sample, it can now decide whether the benefit of performing layout optimization is greater than the cost. In this case, it starts the Clustering module, to which it specifies the files and clustering features. This is re-organizing the data and writes it back to the Delta Table.

4
System Design

This chapter outlines the algorithms and decisions behind the clustering and workload analysis components. We describe the main problems and solutions, and then briefly explain how we integrated them in Databricks Delta.

4.1 Integrating SOP and Z-Order

The first section highlights the characteristics of Skipping Oriented Partitioning and Z-Order clustering and presents our final layout re-organization approach, which combines the two.

4.1.1 SOP and Z-Order Characteristics

Skipping oriented partitioning has a number of advantages that makes it highly relevant for our data layout approach. First of all, its goal is to *maximize data skipping on specific workloads*. Therefore, the clustering algorithm itself is workload-aware and exploits data locality to optimize skipping on relevant features. Partitions are created such that their bit vectors favor skipping on frequent and highly selective predicates. Since in our system adjusting data layout is highly coupled to a particular workload, SOP provides a good reference point.

Another relevant advantage of SOP is that it *scales well in terms of number of features used for clustering*. Additionally, skipping is actually improved when more features are used (30). That is because a larger number of predicates characterizes the workload better. Moreover, even though the input size grows with the number of features, the running time of the clustering algorithm is not heavily impacted. And that is because the number of distinct bit vectors that are created based on the dataset is very unlikely to reach its upper bound. For a set of n features, in order to generate approximately all bit combinations - 2^n bit vectors, it requires all feature predicates to have close to 50% selectivity. Additionally, there needs to be enough data tuples in order to create all combinations. We assume

that in reality data is skewed and this situation is improbable. However, we conducted experiments that show how well the clustering scales with exponentially bigger numbers of bit vectors. Figure 4.1 shows a small processing time up to 2^{13} vectors, after which it increases abruptly. The test was performed on a 4 cores and 31GB memory instance. For better scaling, the algorithm can be parallelized. However, tests and experiments carried out in (30) show that such a big number of vectors require a few tens of clustering features.

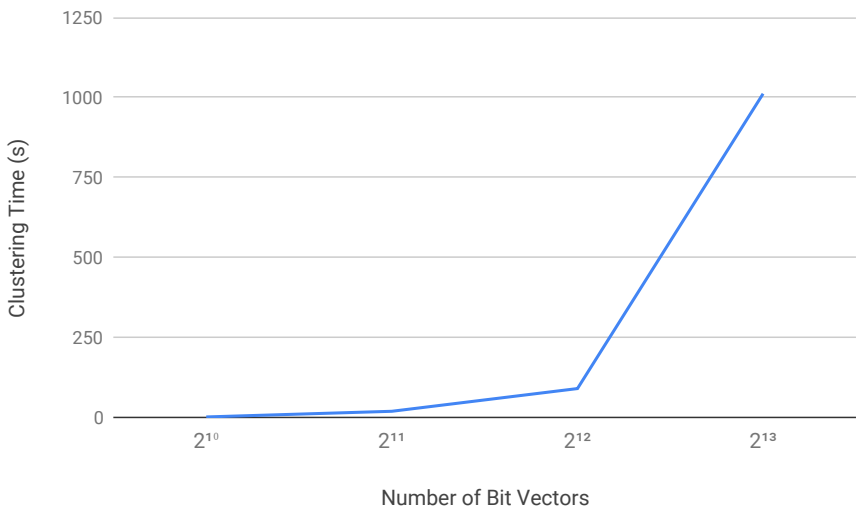


Figure 4.1: Clustering time by the input size

Compared to the current Z-Order solution, SOP solves naturally the *complex expression* shortcoming. It inherently works on any type of filter predicate. However, it does not guarantee efficiency for any type of workload. There is a filter pattern that is disadvantaged by this approach: *binary comparison filters on the same column/expression but with frequently changing literals*. E.g. expressions related to date, timestamps, IPs fall into this category. In these cases, each expression-literal combination creates a different predicate. This grows the predicate space considerably and makes the choice of relevant features more difficult and expensive. Moreover, if each combination is infrequent, they will never be chosen as clustering candidates. SOP has its own method for dealing with this type of predicates: it chooses the filters that subsume the most queries. However, for date filters, the result would be irrelevant. This is where Z-Order proves to be helpful.

One of the main advantages of Z-Order is that it *allows range skipping*. Unlike SOP, clustering can be made on expressions, without taking into account all the different values they are compared with. When performing skipping, the expression is fitted in a min-

Predicates from the workload: ip = 10.3.44.5, ip = 10.22.35.7, ip = 10.63.4.2, ip = 10.94.3.2	
SOP features ip = 10.3.44.5 ip = 10.22.35.7 ip = 10.63.4.2 ip = 10.94.3.2	Z-Order features ip
SOP skipping for predicate ip = 10.77.7.7 None - predicate not found in statistics	Z-Order skipping for predicate ip = 10.77.7.7 Works

Table 4.1: Feature and skipping comparison between SOP and Z-Order

max range, so any comparison literal can be used. Table 4.1 illustrates the clustering and skipping difference between SOP and Z-Order.

Z-Order however, has an important limitation. Although it can be enabled on any type of predicate, its *efficiency decreases with the number of clustering features.* Each additional feature reduces the skipping power of any one dimension. If we model a Z-Ordered dataset of size n by d dimensions as a d-dimensional hypercube, then filtering on one column reduces dimensionality by 1, so the number of tuples that still have to be read is $r = n^{\frac{d-1}{d}}$. As d grows, r grows as well, approaching n and reading increasingly more. Thus, it is impractical to cluster on a large number of features.

Both clustering approaches have strong and weak points. However, each disadvantage on one side can be counterbalanced by a strength on the other. We propose a clustering method that combines SOP and Z-Order, in order to benefit from their combined advantages while minimizing the weaknesses. Therefore, the final clustering approach should meet the following requirements:

1. Scales well to a larger number of features
2. Broader predicate coverage - clustering and skipping work on complex expressions as well
3. Works well for both fixed format predicates and binary comparison filters with frequently changing constants

4.1.2 Clustering Approach

We propose a number of improvements to the current Z-Order solution and a method to combine SOP and Z-Order in order to meet all the goals mentioned above.

Feature scalability. To maintain a good level of skipping effectiveness, we limit the number of Z-Order expressions to 4. (Internal Databricks experiments showed this is an optimal threshold). However, because we complement Z-Order with SOP, which accepts many more dimensions, we achieve feature scalability.

Complex expressions and file statistics. We enabled Z-Order clustering and min-max skipping on complex expressions. When clustering on a complex expression, a new column with the expression evaluation is temporarily added to the table, until the final partitioning is decided. As file statistics, we added minimum and maximum values for complex expressions used in clustering, alongside the other statistics on table columns. The expressions are written in the Delta Log in a normalized SQL string format, which is matched against pushed-down filters at scan time. For SOP skipping, we included bit vector statistics. We keep the bit vector as a long value, alongside a list of the predicates used in clustering.

Furthermore, we added Z-Order and SOP statistics for expressions that were not part of clustering, but are relevant to the workload and can positively impact file skipping at scan time. We call these *additional statistics*. We discuss in the next chapter what types of features form the additional statistics.

Combining SOP with Z-Order. With the new changes, both Z-Order and SOP support clustering on complex expressions. From now on we use the term feature to denote any dimension used for clustering. A feature can either be a predicate evaluated to a boolean, or a column or expression of string or numerical type that is used in Z-Order.

In order to maximize the benefits of the approach, we prioritize predicates with frequently changing literals for Z-Order and leave everything else for SOP. The exact methodology for choosing the final features is described in detail in the next section.

Our clustering approach integrates SOP into Z-Order. Concretely, assuming there are two sets of clustering features, S for SOP and Z for Z-Order, the steps for clustering are the following:

C1 Perform SOP clustering on features from S - augment the dataset with bit vectors and cluster them based on the workload, following the original bottom-up clustering described in (30). This results in a number of clusters, each with a SOP partition index.
C2 Add a new column to the dataset which, for each tuple, stores the SOP partition index it belongs to.

C3 Perform Z-Order clustering on features from Z and the SOP partition index column. The resulting final partitions are written to files. Each file has min-max statistics for features in Z and a bit vector. Depending on the SOP clusters size, a file might have tuples from one or multiple different SOP clusters. The statistics bit vector is the disjunction of all the bit vectors of the corresponding SOP clusters.

An additional problem appears from Z-Ordering on SOP clustering indexes. Originally, each cluster has a bit vector and a corresponding index. There is no constraint on how the indexes are mapped to the clusters. After Z-Ordering, each file can contain multiple SOP partition indexes and the final bit vector combines the bit vectors of each corresponding cluster. Then, it might happen that these bit vectors are unrelated and it degrades the skipping effectiveness. E.g. combining a bit vector with its negation results in a all-ones bit vector, which assures that no skipping can be done. Two bit vectors are related if they have as many overlapping 1's as possible, such that OR-ing them results in as many 0's as possible. Therefore, there needs to be a mapping between SOP partition indexes and bit vectors, such that consecutive indexes correspond to related bit vectors. We achieve this by ordering the bit vectors and assigning SOP partition indexes in an increasing order. The algorithm used for ordering the vectors is described below.

Problem statement: Given a list of m-dimensional bit vectors, sort it in such a way that OR-ing two consecutive bit vectors results in as few 1 bits as possible.

Solution Because imposing an ordering on these vectors is problematic (they cannot be compared), we considered a different approach. We created a fully connected graph where the bit vectors are vertices. Each edge is assigned a cost equal to the distance between the vectors. The problem reduces to devising a distance function and finding a minimum cost traversal of the graph. In the end, this becomes the travelling salesman problem (50). An optimal solution takes exponential time, therefore we use a 2-approximate version (51). It is a relaxed version of TSP, which guarantees the solution cost is at most two times bigger than the optimal one. The algorithm steps are the following:

P1. Choose a vertex as the starting point.
P2. Build a Minimum Spanning Tree (MST) over the graph starting from the chosen vertex. We used Prim's algorithm (52).
P3. Traverse the MST using preorder and store the vertices in a list along the way. This

is the final output.

However, there is one constraint in order for the approximation to work. The graph needs to conform to the triangle inequality, meaning that the lowest cost from any vertex i to any vertex j is always by going directly from i to j:

```
dist(i, j) <= dist(i, k) + dist(k, j), for any i, j, k from V
```

Therefore, we need to define the distance between any two bit vectors (vertices), such that it respects the triangle inequality and is proportional with the ordering that we want to achieve in the end. Therefore, we use the following definition:

```
dist(i, j) = the minimum number of bits that have to be changed to transform
i into j or vice versa.
```

The metric basically counts the number of points in which the two vectors differ. Given that the vectors hold binary values, each such point in the vectors represents a set of opposite bits at the same index, which results in a 1 bit after OR-ing. This is undesirable in our case as the goal is to maximize the number of 0 bits. Therefore, the smaller the distance, the better. Below we show that this metric conforms to the triangle inequality and so that we can apply the 2-approximation algorithm.

Proof of the triangle inequality. We show the inequality holds using proof by contradiction method. Therefore, we want to prove that there are *a,b,c* - 3 bit vectors for which

$$d(a,b) > d(a,c) + d(c,b)$$

Let $I = \{i_1, .., i_p\}$ be the set of indices in a in which vector a differs from b. ($a[i_k] != b[i_k]$).
Let $J = \{j_1, .., j_s\}$ be the set of indices in a in which vector a differs from c. ($a[j_k] != c[j_k]$).
Therefore, d(a, b) = p and d(a, c) = s.
Let K be the set of indices in which c differs from b. Thus, d(c, b) = |K|.
We can express K as $K = \{I \backslash J\} \cup \{J \backslash I\}$. We reached this equality following these logical steps:

- c is a copy of a where bits at positions in J have been flipped
- The bits corresponding to $\{I \cap J\}$ are the flipped bits from a that were needed in b. Since in c they have already been changed, no change has to be done to them.

- The bits corresponding to $\{I \backslash J\}$ are the bits from a (that are copied in c) that need to be changed to obtain vector b.
- The bits corresponding to $\{J \backslash I\}$ are the bits in which c differs from a but they did not need to be flipped to obtain b. Therefore, they need to be flipped back.
- The bits that need to be flipped are $\{I \backslash J\}$ and $\{J \backslash I\}$

We follow with $K = \{I \backslash J\} + \{J \backslash I\} = I - \{I \cap J\} + J - \{J \cap I\} = I + J - 2 * \{I \cap J\}$. If we denote $|I \cap J| = x$, our inequality becomes:
|I| > |J| + |K|
p > s + (s + p - 2x)
x > s - which means $|I \cap J| > |J|$, which is not possible. Therefore our assumption was wrong so the triangle inequality holds.

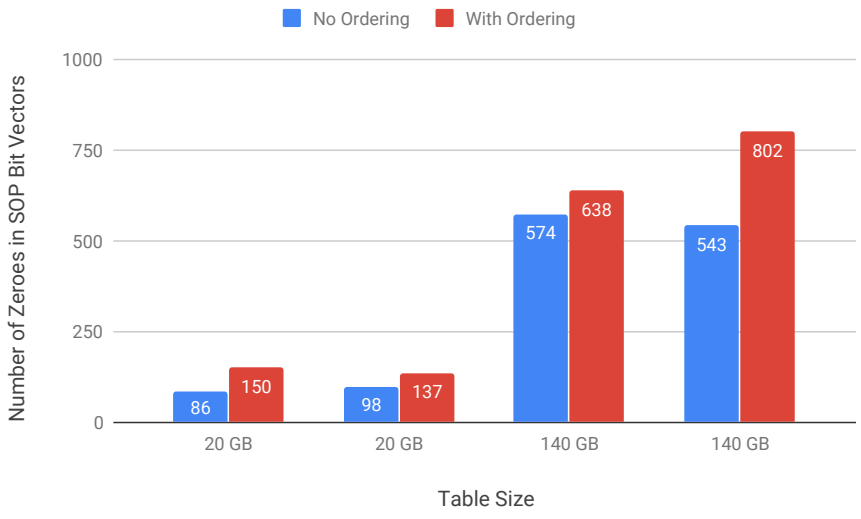


Figure 4.2: The total number of zeroes in the SOP bit vector statistics, with and without bit vector ordering.

In order to test the effectiveness of the bit vector ordering, we measured the improvement in the skipping capabilities when ordering is active, compared with no SOP partition index ordering. The metric we devised counts the maximum number of files that can be skipped for each SOP feature, which is basically the number of zeroes in the bit vectors. We tested on two tables, of 20 and 140 GB. We clustered them only with SOP, with two sets of features, of 5 and 11 elements. Figure 4.2 shows the total number of zeroes for each

experiment, with and without the bit vector ordering. We obtained from 11% up to 74% more zeroes with the ordering being active.

4.1.3 Implementation Details

We implemented the clustering and data skipping additions on top of existing Databricks Delta functionalities. The clustering algorithm is a separate module that can be triggered at any point, given the list of features and files to cluster.

To add support for min-max skipping on complex expressions and also SOP bit vector skipping, we enriched the existing file statistics found in the Delta Log. Figure 4.3 shows an example of statistics that might be written for a file, in Delta Log. There are mainly two types of statistics - for Z-Order and SOP. They are further split into two more categories, features that have been used for clustering and additional features.

Z-Order statistics record minimum and maximum values for Z-Order features. These include plain table columns, as they are currently used in Delta, and complex expressions as well. In the example, orderDate, shippedDate and *total* are some of the table columns, while *month("shippedDate")* is a complex expression feature, selected from the workload, which was part of the data clustering. SOP statistics have a slightly different format. The first entry is a list of features in string format, followed by an entry that records the bit vector for those features, in the order they are enumerated. In reality, additional features are merged with the initial ones, as there is no need for making any distinction when performing data skipping.

These statistics are computed after clustering, when the files are written to persistent storage. The features pass through a normalization step and are transformed to string format. At query time, pushed-down predicates are converted in the same way into strings and matched against the statistics for each file. We chose to write the predicate features into each file metadata for simplicity, as they might change over time. We would otherwise need a system that maps a set of features to certain clustered files.

When running a query, the data skipping module verifies the predicates on file metadata. For each predicate we first check whether there is any Z-Order statistics for it, and if not, pass on to SOP. A predicate might match a Z-Order feature, an SOP feature or none. A file is skipped if, for all predicates, the statistics determine that there is no relevant data tuples. For our file example, a complex expression predicate like *month("shippedDate") > 10* is first tested and matched against the Z-Order statistic as it is a binary comparison. The min-max values can determine whether the file contains useful data or not, in this case the answer being no. On the other hand, a predicate like *mapvalue("shippingInfo",*

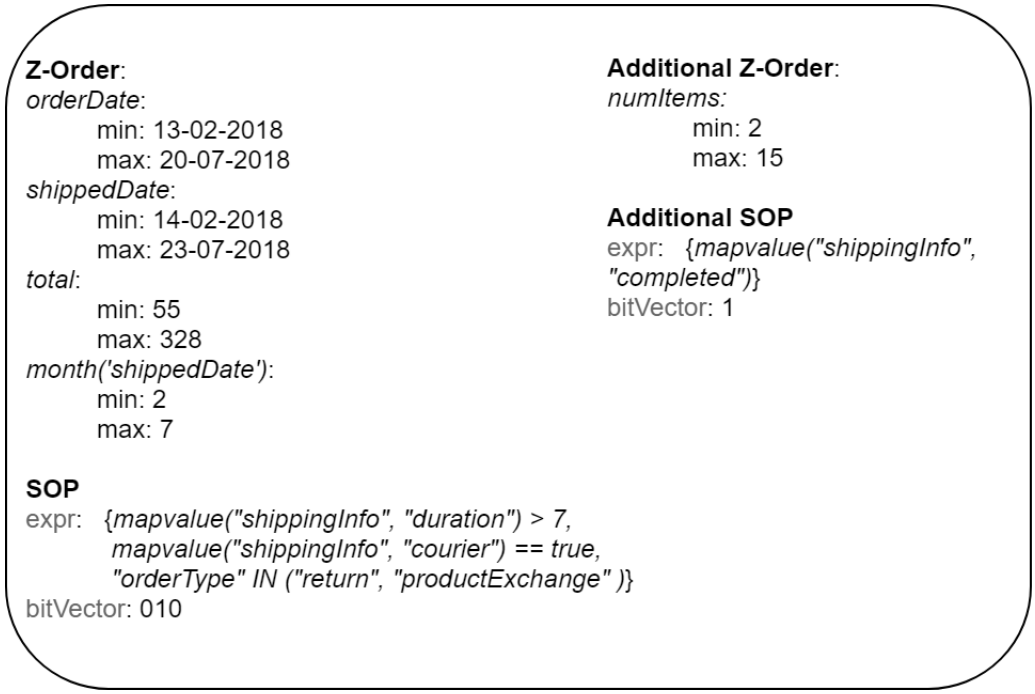


Figure 4.3: Representation of our version of file statistics, as written in the Delta Log.

"courier") == true has no match in Z-Order statistics so it is searched among the SOP features. The bit vector is checked at the index that the feature is found in the expr entry. In this case, the index is 1 and the bit vector the value is also 1, so the file has tuples that satisfy the predicate.

4.2 Feature Selection

This section explains how we navigate the predicate space in order to extract the relevant clustering features. We first dig into the important metrics that we use to establish an initial list of candidates and then how we detect and eliminate feature correlations.

4.2.1 Workload Analysis

The initial step in workload analysis is to reduce the search space of the predicates, based on information derived from the Query Log. Thus, this phase depends solely on characteristics obtained from workload monitoring, with no knowledge over the data. Dataset properties are used later, for correlation detection.

Considering a workload WL is read from the Query Log, the purpose of the initial workload analysis is to find two sets: $Z_{cand} = \{z_1, .., z_p\}$ and $S_{cand} = \{s_1, \ldots, s_q\}$ that contain candidate features for Z-Order and, respectively, SOP.

The question that arises now is what metrics can be used or derived from the Query Log data, in order to narrow the set of all features down to the two sets of candidates. We considered and analysed multiple solutions and ultimately reduced the metrics to filter frequency and comparison points, which are detailed below. The other options that were taken into account are described afterwards.

4.2.1.1 Frequency

We consider feature frequency as the most straightforward and reliable metric, therefore it weighs the most in the workload analysis algorithm. There are two main reasons that support this choice:

- If a feature has been frequent in the past, we can assume it will be used in the future as well, at least over a period of time.
- If future workloads maintain the same patterns (frequent predicates will remain frequent), then it is advantageous to optimize for frequent features that cover the most queries.

Frequency is computed differently for SOP and Z-Order features. The filters that qualify for Z_{cand} are in the shape of binary comparisons. In the frequency count, the value that an expression is compared against is irrelevant. Therefore, the frequency is computed only for the side of the binary expression that contains a variable. For SOP features, the whole predicate counts, irrespective of its format.

4.2.1.2 Number of Comparison Points

This is a metric used only for Z-Order features, for which we prioritize binary comparison predicates. Such a predicate has a left and right hand side, connected by an operator. We consider the left hand side is the one that refers a column and the right hand side is a constant expression. A normalizing step can easily transform comparison filters into this format. For Z-Order, a candidate feature consists only of the left hand side and excludes the constant it is compared with. Therefore, we are interested to see in how many different filter formats each candidate was. To achieve this, the analyser extracts all distinct right hand sides of the filters in which each z_i was found. We refer to these values as comparison points.

The cardinality of distinct comparison points is relevant to determine whether a feature is suitable for Z-Order or not. As explained in the previous chapters, Z-order combined

with range skipping benefit those features that are compared against many different values. Therefore we impose a minimum threshold for the number of distinct comparison points.

Behind this decision lies another important consideration. Z-Order is efficient only with a very limited number of clustering features, therefore they need to be chosen carefully. On the other side, SOP is more relaxed and scales well to a larger number. Thus, features with few comparison points can easily be reconstructed into their original filter format and used in their entirety for S_{cand} instead. Using this constraint, we eliminate from the start all features that do not derive a particularly bigger benefit from Z-Order than SOP.

4.2.1.3 Other relevant metrics

In the query log, we can additionally keep metrics about the size or volume of scans. The three metrics that we considered are *scan volume, scan duration and filter selectivity.*

Scan volume and *duration* are similar and proportional, therefore they share the same characteristics and we are treating them together. They are useful in pin-pointing more relevant queries that scanned high volumes and can potentially benefit more from improved data skipping. However, based on this alone, it is impossible to say whether the query actually needed the whole amount of data or it scanned such a big volume as a result of poor file pruning capacity. This is where filter selectivity would help clear the situation. We can keep statistics as the number of rows needed and the number of rows read so that we can determine exactly when data skipping is inefficient.

However, there are a series of common disadvantages which led us to dropping these metrics. The first one stems from the way these metrics would be computed in relation with a query. Usually, a query has multiple filters which, separately, would determine different scan volumes and selectivity. But when they are placed together in a query, the scan volume or selectivity reflects the combined characteristics of all the filters in that query. As an example, consider a query with filters a and b. Filter a has already good data skipping, so the volume it scans is low, while filter b has poor skipping capabilities. Together, the result reflects good skipping, while in reality this is only due to the first filter.

The second important disadvantage is that by using these metrics, we prioritize filters that scan large volumes and have low selectivity, which is not well exploited. The filters that, on the other hand, already leverage good data skipping, either thanks to the statistics or the natural layout of the data, would lose precedence in clustering. As a consequence, clustering can alter the good data skipping capabilities that these features had initially.

4.2.2 Data Correlation

The second part of the workload analysis uses data characteristics to find correlations. Real datasets can have intricate connections and dependencies between columns. These patterns might determine correlations in the evaluation of predicates as well. Here we target especially cases where clustering on one feature automatically results in another feature being clustered. Therefore we need not cluster on both features, if they are correlated, as it would waste time and resources. These correlations can be frequent and, when passing undetected, they affect clustering efficiency by creating redundancy in the chosen features, instead of making space for other relevant dimensions.

One simple example is illustrated below (4.2). There are two columns from a table which holds information about products of a store. The first column is the release year of the product while the second is the number of items sold until present.

Year	ItemsSold		Year	ItemsSold
2017	1024	⇒	2014	3800
2018	770		2015	3400
2015	3400		2017	1024
2014	3400		2018	770

Table 4.2: Example of data correlation based on values order

It can be observed that ordering by year automatically imposes a descending order on the ItemsSold column. In the situation where this table needs to be clustered on both these dimensions, we can reduce the feature space to just one of the two columns and safely assume that the other is going to be ordered as a result.

Detecting such correlations is not trivial, both from a mathematical and data processing point of view. A good solution would need to quantify the ordering relation between two features while using minimal compute and storage resources.

From a mathematical standpoint, a correlation coefficient might be a good fit for measuring dependencies. However, most correlation coefficients (Pearson (53), Spearman (54), Kendall tau (55), etc.) measure either only linear correlations or statistical dependencies between data columns. One relevant approach that aims to detect correlations between attributes in a database is described by CORDS (56). It uses mean-square contingency (57) as a metric for correlation and it has the advantage of obtaining accurate results on a relatively small sample (a few thousand tuples) irrespective of the dataset size. On a sample, the expression that computes the index for a pair of columns is the following:

$$\chi^2 = \sum_{i=1}^{d1} \sum_{j=1}^{d2} \frac{(n_{ij} - n_{i.} n_{.j})^2}{n_{i.} n_{.j}}$$

where d1 and d2 represent the domain cardinality of the two columns, n_{ij} is the fraction of tuples (x, y) where x = i and y = j, and $n_{i.}$ and $n_{.j}$ are the marginal totals: $\sum_{j=1}^{d2} n_{ij}$ and $\sum_{i=1}^{d1} n_{ij}$.

In CORDS, domains are actually divided into categories in order to decrease granularity and the number of structural zeros (where $n_{ij} = 0$), which can affect the accuracy.

The mean-square contingency is adequate for detecting statistical dependencies. That is, whether a value or category of values in a column determines a specific value/category in another column. Primary keys are by default correlated with all other columns. Having this property, mean-square contingency is unsuitable for Z-Order correlations, but can be easily used for SOP correlations. We describe each case in the next subsections.

From a data point of view, we follow a data sample approach. The method that we propose extracts a sample and then tries to detect correlations between Z_{cand} and S_{cand} separately. Specifically, the steps are as follows:

1. Extract a data sample from the dataset.

2. For each feature z_i in Z_{cand}, order the sample by it and compute a correlation index between the z_i and all others.
   If the correlation index exceeds a threshold, consider the pair correlated. The correlation indexes for Z and S are described in the next subsections.

3. For each feature in S_{cand}, do the same as step 2.

Based on the found correlations, the analyser then builds a correlation graph and uses a traversal algorithm to extract and divide the features into a final set and a correlated set.

4.2.2.1 Z-Order Correlations

One important aspect to note about correlations between Z-Order features is that we are not looking only at statistical correlations, but, more precisely, at correlations that determine the same ordering on features. That is, sorting on one feature results in another feature being sorted as well (see 4.2). If no sorting dependency is involved, then min-max skipping would not be improved on the correlated feature.

Given this constraint, mean-square contingency is not restrictive enough, as it does not require any order, but rather just a dependency. The short example below explains why this index is not suitable.

Col1	Col2
1	1
2	0
3	1
...	...
N	1

Table 4.3: Correlation example

Consider the two columns in 4.3, one that has integers and one that marks the parity of the number. In the CORDS expression, the domains size of the two variables, d1 and d2, are N and, respectively, 2. Thus, the CORDS index is reduced to:

$$\chi^2 = \sum_{i}^{N} \frac{(n_{i0} - n_{i.}n_{.0})^2}{n_{i.}n_{.0}} + \frac{(n_{i1} - n_{i.}n_{.1})^2}{n_{i.}n_{.1}}$$

Since $n_{i0} = \frac{1}{N}$ where i is odd and 0 where i is even,
$n_{i.} = n_{i0} + n_{i1} = 0 + \frac{1}{N} = \frac{1}{N}$,
$n_{.0} = \sum_{i=1}^{N} n_{i0} = \frac{1}{N}\frac{N}{2} = \frac{1}{2}$ (n_{i0} is $\frac{1}{N}$ for $\frac{N}{2}$ values and 0 for the rest),
and similarly for $n_{.j} = \frac{1}{2}$.
Finally, replacing all the terms, χ^2 becomes 1, which denotes a perfect dependency. In our situation, it does not bring any insight into how data is clustered, as Col2 does not have its values arithmetically ordered. Splitting it into files will always generate the maximum statistics range: a minimum value of 0 and a maximum of 1, which is irrelevant for data skipping.

Since mean-square contingency does not provide any information on the sorting correlation, we propose a sorting index that describes how well-sorted one array is. In the correlation algorithm, the first step is to sort the sample on each feature. For each iteration, the sorting index is computed on all the other features in order to determine the sorting degree. The sorting index is computed as follows:
Consider array A = [1, 3, 5, 2, 4].
We compute $k = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \left\{ \begin{array}{c} 1, \text{ if A[j] } > \text{ A[i]} \\ 0 \text{ otherwise} \end{array} \right\}$

For each element in the array, the expression counts the number of elements to its right that are larger than itself and then adds the results. If A is in perfect ascending order, k would be $k_{max} = \frac{n(n-1)}{2}$.
If A is in perfect descending order, k would be $k_{min} = 0$.
We compute the sorting index as being $sort - index = \frac{k}{k_{max}}$.
If $sort - index > 1 - \epsilon$, or $sort - index < \epsilon$, we consider there is a correlation.

The algorithm for calculating the index has quadratic complexity, but given we operate on data samples, our experiments did not reveal any bottleneck in the computation.

One disadvantage the index has is that it can output false positives in the case of skewed data or tuples with many duplicates. Specifically, if column a has all or mostly distinct

values and column b is composed of only one value, ordering on column a has no effect on b. Column b will appear to be sorted regardless of other columns' ordering. Therefore a will be correlated with b all the time. One way to eliminate the false positives is to always verify correlations in both directions. Column b is correlated with a only if a is naturally sorted in the sample, which we consider to have small probability. Therefore, for Z-Order candidates pairs, we verify correlation in both directions. If the relation is bidirectional, then we consider the correlation holds.

4.2.2.2 SOP Correlations

SOP correlations are correlations between features which evaluate to a boolean. Thus, sorting them is unnecessary, as all the possible combinations are known beforehand and limited in number. Since there are only two possible values, any arrangement (true $\Rightarrow$ false or false $\Rightarrow$ true) is ordered. So the question reduces to finding if there is a correspondence between values from one column with the values in another. In other words, whether a true value in column a corresponds to a single value (true or false) in column b. This measure is exactly what mean-square contingency determines. Considering our domains have two possible values, it measures the statistical dependency between the two categories. Moreover, for each pair of features, we need to compute the index only once, which reduces from time and resource pressure.

4.3 Feature Selection Algorithm

This section presents the whole feature selection algorithm, from reading the query log, to the final feature choice.

Problem statement: Given a dataset D and a workload $WL = \{Q_1, .., Q_n\}$ on D, where each Qi is a query characterized by its pushed-down filters: $\{F_1, .., F_{qi}\}$, find the Z-Order and SOP features (Z and S sets) that are most suited for clustering. Alongside clustering features, we build two more sets - Z_{add} and S_{add} - which consist of *additional candidates* that can improve data skipping by being present only at file statistics level. That is, they do not directly contribute to clustering, due to lower frequencies or being correlated, but will be present in data statistics at write time and will be used at scan time to increase the level of skipping.

There are two main indicators of feature suitability: the number of queries it covers and its degree of correlation with other features. Below is the pseudocode (Algorithm 1) for choosing Z and S, followed by detailed explanations for each step.

Algorithm 1 Feature Selection

P0. Initialize all required sets to the empty set and extract a data sample, D_{samp}.

Choose Z-Order Features

P1. *Extract initial candidates*
for all f_i in F **do**
  if f_i is in format: $Expr1 > / \geq / < / \leq / = / startsWithExpr2$ and numColumns(Expr1) == 1 and numColumns(Expr2) == 0 **then**
    $Z_{cand} = Z_{cand} U \{Expr1\}$
    Freq[Expr1] ++
    $C_{Expr1} = C_{Expr1} U \{Expr2\}$

P2. *Eliminate infrequent candidates*
for all z_i in Z_{cand} **do**
  if $|C_i| < \theta$ **then**
    $Z_{cand} = Z_{cand} \backslash \{z_i\}$
P3. $graph \leftarrow buildCorrelationGraph(Z_{cand}, D_{samp}, "ZORDER")$
P4. $(Z_{chosen}, Z_{additional}) \leftarrow traverseCorrelationGraph(graph, Z_{cand}, Freq)$

Choose SOP features

P5. *Extract initial candidates*
for all f_i in F **do**
  if f_i is NOT in format: $Expr1 > / \geq / < / \leq / = / startsWithExpr2$ OR f_i is in this format but ($Expr1 \notin Z_{chosen}$ and $Expr1 \notin Z_{additional}$) **then**
    $S_{cand} = S_{cand} U \{f_i\}$
    Freq[f_i]++

P6. *Eliminate infrequent candidates*
for all s_i in S_{cand} **do**
  if $|C_i| < \gamma$ **then**
    $S_{cand} = S_{cand} \backslash \{s_i\}$
P7. $graph \leftarrow buildCorrelationGraph(S_{cand}, D_{samp}, "SOP")$
P8. $(S_{chosen}, S_{additional}) \leftarrow traverseCorrelationGraph(graph, S_{cand}, Freq)$

The algorithm can be split in two parts, P1-4 and P5-8 which correspond to choosing Z and S respectively.

Starting with Z-Order features, in P1, only predicate filters that qualify for min-max skipping are selected. There is an additional condition that the left hand side of the filter references only one column, while the right hand side does not reference any. However, Expr1 and Expr2 can take any expression format. Then, Expr1 is added to the list of Z candidates while updating its frequency, and Expr2 is appended to the set of distinct comparison points for Expr1.

P2 performs an initial filtering on the candidates, excluding the ones that have few comparison points. The reason behind this decision is that range skipping does not bring much added benefit in this situation. Each filter that contributed to this candidate can be reconstructed and considered for SOP clustering instead, where more features are allowed.

The routines used in P3 and P4 are used to detect correlations and are described in more detail below. The functions use the correlation indexes and methods presented in the previous subsections. Based on them, a correlation graph is constructed and traversed, the output being the final list of clustering features, alongside the set of additional features. P7 and P8 are the analogous operations for SOP features.

P5 marks the start of the algorithm for choosing SOP dimensions. Similar to the first part, it starts with checking the filter format: if it is not min-max skippable, it is automatically considered a candidate; if it is min-max skippable but its left hand side was not chosen in neither final Z features nor in additional Z-Order ones, then the whole predicate is added to Scand.

The filtering step eliminates all candidates that are below a minimum frequency threshold. The last two steps compute the correlations and output the final list of chosen SOP candidates and additional features which will contribute only to the statistics.

Correlation Graph

We discussed how correlation is detected between two features, on a data sample. In our algorithm, correlation is tested against each pair of features. Consequently, a feature can be correlated with multiple others, which, in turn, are further correlated with more features and so on. Having a complex network of correlations, the problem that appears is which are the main features and which are considered correlated? The most relevant ones are used in the clustering algorithm, while the correlated set is merged with the additional statistics.

To help answer this question, we model the correlations into a *correlation graph*. The vertices represent the features and the presence of an edge between any two vertices marks the existence of a correlation between the two corresponding features. The problem then transforms into how to traverse the graph in such a way that we can obtain two disjoint sets of vertices that represent the clustering features and the additional ones.

The buildCorrelationGraph (algorithm 2) method shows the steps for creating the graph. Z-Order and SOP correlation graphs are built separately. In the code snipped we present the case of Z-Order. For each feature, we sort the sample and compute the correlation index. If it exceeds a threshold, a directed edge from i to j is added to the graph. For Z-Order, the correlation method uses the sorting index, while for SOP the mean contingency index, as described in the previous sections. In the end, all edges that are not bidirectional are deleted from the graph, in order to eliminate false positives. For SOP, the method is simplified, as it does not need the sorting step and the final bidirectionality check.

Algorithm 2

```
function buildCorrelationGraph(candList, D_samp, corrType)
    V ← candList
    E ← {}
    Graph ← (V, E)
    for all c_i ← V do
        Sorted = sort(D_samp, c_i)
        for all c_j ← V, c_j! = c_i, j > i do
            Corr ← correlation(c_i, c_j, sorted, corrType)
            if Corr > φ then
                E = E ∪ {c_i → c_j}
    for all (c_i, c_j) ← E do
        if (c_j, c_i) ∉ E then
            E = E (ci, cj)
    return Graph
```

Having the correlation graph, the subsequent issue is traversing it and choosing the features that will be subject to clustering and the ones that are correlated. We use the term *primary features* for the ones that are the chosen features and *correlated features* for the ones that depend on the primary ones, according to the graph. To better understand the problem, consider a correlation graph as the one below:

There are multiple possibilities of choosing primary and correlated features. Once a vertex is chosen, its direct neighbours can be added to the correlated set. But in the situation where these correlated features have dependencies further down, they cannot simply be dismissed. Moreover, we cannot assume that the correlation relation is transitive.

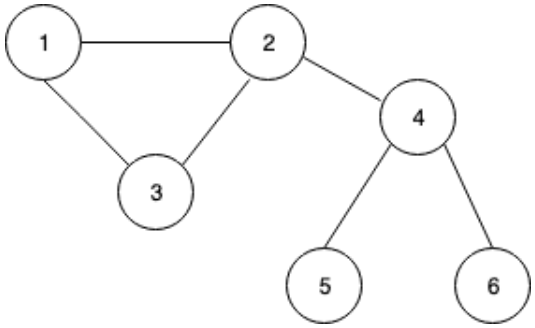


Figure 4.4: Example of correlation graph. Each node is a feature. Each edge denotes an existing correlation between the two features.

Thus, vertex 1 being correlated with 2 and 2 with 4 does not imply any correlation between 1 and 4.

We identified two subproblems to be solved in the graph traversal algorithm:

- The order of vertex traversal - such that relevant features are chosen first
- A strategy that decides whether a vertex is added to the correlated set or remains active in the traversal

Considering the first problem, the order can be established by different policies. One can be the cardinality of the nodes - the more dependencies they have, the more important they become. Another one can be based on the frequency of the features in the workload. The latter guarantees more reliability in the face of correlation index errors, as it selects the most frequent features first anyway. Also, a combination of the two policies is possible.

We solve the second problem by adding vertices to the correlated set only if they have no further unexplored correlations. This guarantees that no correlation pair is ignored. Moreover, combined with the traversal order, in every pair, the most important vertex takes precedence.

The correlation graph traversal algorithm is described below:

The method uses a policy for selecting the current visited vertex and a heuristic that stops the algorithm once a condition is met (e.g. the primary reached a max size or there are no more frequent enough features). For each current vertex, it cuts all the edges between it and its neighbours, and, all the edges between the neighbours themselves. This marks that those correlated pairs have been visited. If a neighbour has no connection anymore, it is added to correlated set. Otherwise, it means the vertex has other correlations and it might be added to the Primary later on.

Algorithm 3

```
function traverseCorrelationGraph(graph, cand, Freq)
    corr ← {}
    primary ← {}
    while there are more possible candidates for primary do
        crt ← extract_vertex(graph, freq, policy)
        primary = primary ∪ {crt}
        for all n in neighbours(crt) do
            for all n1 in common-neighbours(n, n1) do
                if degree(n1) == 1 then
                    delete edge n - n1
            delete edge crt - n
            if degree(n) == 1 then
                corr = corr ∪ {n}
    return (primary, corr ∪ {allunvisitedvertices})
```

Fig. 4.5 depicts the graph traversal. The order follows a most frequent feature policy. First, vertex 1 is visited and added to the primary set. Vertex 3 does not have any other dependencies, so is considered correlated. Node 2, however remains the same in this step. The edges between 1, 2 and 3 are all cut. In the next iteration, vertex 4 is the second most frequent feature, so is added to primary, while all its neighbours are added to the correlated set.

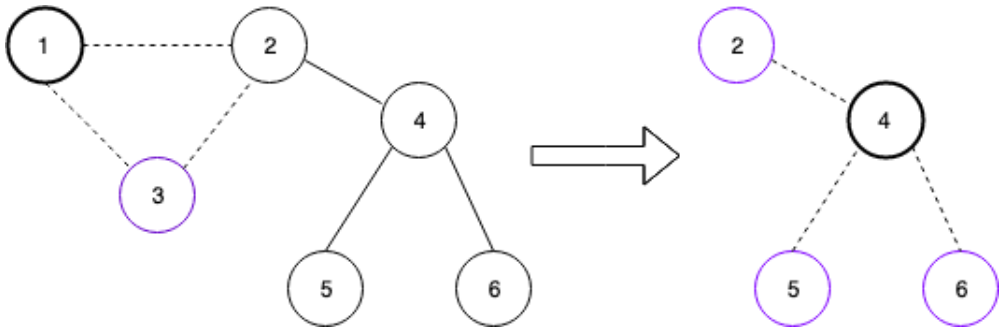


Figure 4.5: Correlation graph traversal
bold line - primary vertex; Purple line - correlated vertex; Normal line - vertex not visited yet

So the final clustering features are 1, 4 and the additional features - 2, 3, 5, 6. Of course, there will be many isolated vertices, which are either added to the primary or to the additional features set, depending on the traversal order.

To sum up, the feature selection algorithm first greedily selects a set of relevant candidates based on frequency and then extracts the final list of features, taking into account data correlation. The final clustering features set is limited in size and is built based on correlation graph traversal order. All other features that are either correlated or were not included in the final set, form the set of additional features, which are to be used only at file

statistics level. Z-Order an SOP have separate sets of clustering and additional features.

4.4 Summary

In this chapter we discussed the approach we proposed for physical layout and feature extraction. We built our solution in Delta, alongside the current Z-Order optimization command, which only clusters data on a set of manually provided columns.

We identified two main shortcomings in the Z-Order algorithm: it does not work in its current format for complex expressions and its efficiency decreases with the number of clustering features. To solve these problems, we first accommodated complex expressions for Z-Order clustering and min-max skipping. Next we proposed a way of integrating Skipping Oriented Partitioning with Z-Order. SOP works naturally on any type of predicate and, moreover, maintains its efficiency even with a large number of features. However, compared to Z-Order, it works only on binary expressions and does not allow range skipping. Therefore, by combining the two clustering approaches we aim to minimize the disadvantages and derive the most benefit out of each one.

There are multiple solutions for combining SOP with Z-Order. We chose a solution that basically injects SOP into Z-Order: we perform SOP and then use SOP partition indexes as a new dimension for Z-Order. This solution is practical and integrates easy within Delta. However, one disadvantage is that Z-Order features are given more relevance than the SOP ones (the SOP partition index resulted from combining all SOP features has the same weight as one Z-Order feature). We accepted this shortcoming as our data science results show that filters that can be candidates for SOP have a lower frequency than those of Z-Order (binary comparisons).

Based on the clustering model, we then proposed a way of extracting relevant features from the workload. Because there are two separate clustering algorithms with different constraints and characteristics, we moulded the feature selection such that it enhances each algorithm's advantages. Therefore, there are two sets of features - the Z-Order one consists of a few elements which would benefit the most from range skipping, while SOP features include the other relevant predicates, that usually appear under the same format in the workload. We then went on to find correlations between the features and reduce the cardinality of the two sets. We proposed an approach based on a data sample, in which we test if ordering on one feature determines other features to be ordered as well. The found correlations are then modeled into a graph, which is traversed in a custom order that help separate the final clustering features from the correlated ones. The latter are then used as

additional features, for which we only provide file level statistics that are leveraged in data skipping.

Although in our tests, the chosen features are the ones we expected, the final configuration is affected by the data sample. It might happen that the sampled values are highly skewed and the correlation algorithm fails to detect feature dependencies. Especially in the case of Z-Order, where the number of features is very limited, redundancy due to correlation creates resource waste and decreases efficiency.

5
Integration in Databricks Delta

This chapter discusses how the clustering and feature selection algorithms are combined in an automatic solution, that runs in Databricks Delta. We answer the questions of "when", "what" and "where" to optimize, taking into account the system architecture.

First of all, in Databricks, computation is separated from storage: clusters are started on-demand for running jobs or on-line queries while the data layer is external. Moreover, clusters can have different access permissions that cannot be changed. This decoupling introduces infrastructure complexity when deciding how the optimization takes place. The system needs to assure that layout optimization finished running before the cluster is shut down, that it has write permissions or it does not perform the same operations simultaneously on different clusters? Because providing a reliable and well defined solution for all these problems is out of scope for this project, we settled for an approach that runs correctly on a single cluster. In the case of concurrent optimizations, the first one to commit its changes will win.

The automatic layout optimization is triggered at *every insert query* and performs clustering, if necessary, after writing data to the storage layer. The layout optimization is applied only to *new, unclustered data*. By running on the same cluster on which data insertion completed, we are sure that it has read/write permissions and can perform subsequent writes as well. Moreover, by piggybacking the optimization on writes, the cluster has to wait until the operation is completed.

However, layout optimization is a costly operation. Not only we cannot afford to run it on every single write, but it would also be inefficient. Therefore, it is necessary that the decision to optimize is taken with caution, as resource waste is not desirable. As all computation runs in the customer account, it generates additional costs and, moreover, a considerable overhead on write operations. Thus, the system needs to *estimate the cost-benefit* dynamically, depending on data and workload, and take a decision accordingly. With this purpose, we propose a methodology for computing the cost-benefit function,

implemented by the Cost Benefit module presented in the System Design chapter. It is a way of assessing the potential improvement that the layout optimization can bring, as well as the clustering cost. In the situation where the benefit outweighs the cost, the optimizer can trigger the reorganization operation.

We define the following function that computes the cost-benefit:

$$f(WL, Dataset) = OPTIMIZE_{IMPROVEMENT} - OPTIMIZE_{COST}$$

where WL is the current workload being analyzed and Dataset represents the dataset to be optimized.

The two terms in the expression quantify the cost of performing the clustering and the benefit gained by the current workload if the optimization takes place. When f is positive, the improvement is greater that the optimization cost, thus performing clustering is a good decision.

5.1 Optimization Cost Estimation

Data clustering is a resource intensive and expensive operation. We want to estimate its cost in terms of time before triggering it. The factors that can influence it are: dataset size, number of SOP features, number of Z-Order features, number of additional statistics. Therefore, we model the optimization cost as a function of these parameters.

We take an experimental approach to determine the importance of each parameter. Given a table, we measure the effect of one parameter at a time, by incrementally increasing it. Specifically, for each data point, we perform clustering and measure the running time. We then plot the results and fit them to a line with linear regression.

Setup: We used a cluster of 8 cores and 64 GB memory that use storage cache (20). We cleared the cache after each cluster run. We tested on 3 different tables, of 23, 60 and 80GB in size. The goal is to test both the situation where the whole table fits easily in memory (23 GB) and when it is roughly to the limit or clearly exceeds it (60 and 80GB). In order to meaningfully compare the results of the three different sized tables, we plot the clustering time per GB (ms/GB).

5.1.1 Dataset Size

We start by determining the base cost of clustering, which depends solely on the dataset size. The minimum condition that starts the optimization is when there is one Z-Order

feature. In the case of no explicit Z features, there is still one created artificially that contains the SOP partition index.

For the three tables, we measure the clustering time with one Z-Order feature. We observe a relatively constant time, which can guarantee a certain degree of reliability for our estimation.

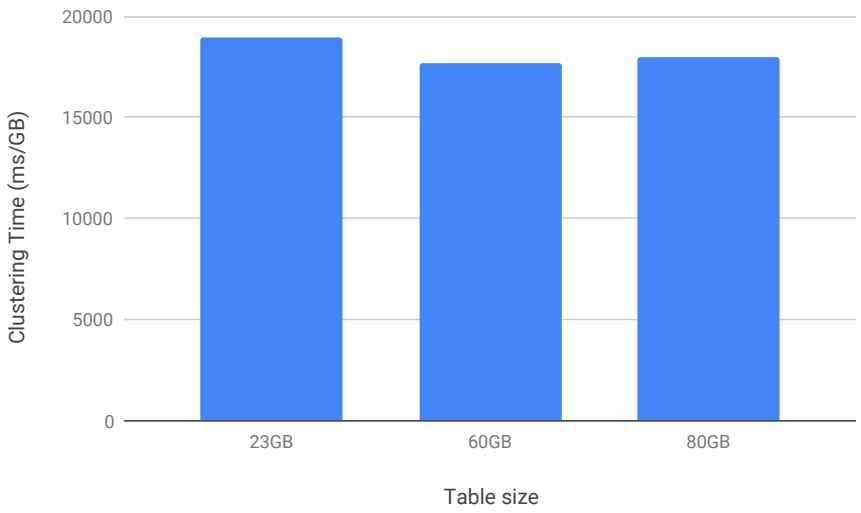


Figure 5.1: Clustering time for different table sizes

5.1.2 Number of Z-Order Features

We then measure how clustering time is affected by the number of Z-Order features. Since we only allow a maximum of 4 dimensions, we tested with feature cardinality only in this range.

We can observe (Fig 5.2) that the three linear regressions follow a similar path so we can obtain a good approximation of the real values. Then, we fit all the points to one line, described by the equation:

$$timeZOrder/GB = m_{ZOrder} * num\_features + base\_cost$$

mZOrder is the line slope and base_cost represents the clustering cost with only one feature, which is a mean of the three values.

Figure 5.2: Clustering time for di□erent number of Z-Order features

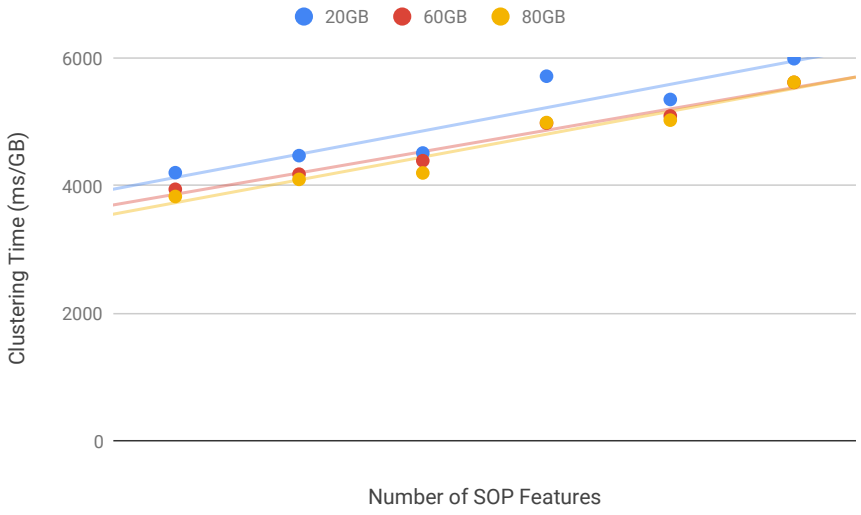


Figure 5.3: Clustering time for di□erent number of SOP features

5.1.3 Number of SOP Features

In the case of SOP features, we only measure the SOP clustering time, for increasingly bigger amounts of features. We plot the running time per GB, as a function of number of features.

The results in Fig. 5.3 show that all tables grow similarly. However, the 60GB and 80GB ones have an almost overlapping trendline. The increase in time is not due primarily to the clustering algorithm itself or the number of different bit vectors, but rather from

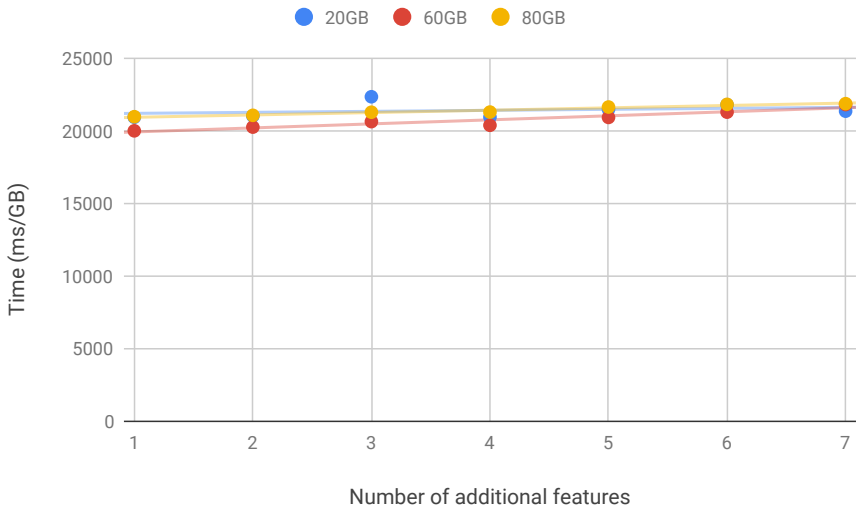


Figure 5.4: Clustering time for different number of additional features

the augmenting step, where the bit vectors are created. Each additional SOP feature adds more memory and compute pressure.

Therefore, we describe the influence of SOP features as a linear equation in the form:

$$time_{SOP}/GB = m_{SOP} * num\_features + sop\_base\_cost$$

5.1.4 Number of Additional Features

The effect of additional features is less intense than other parameters (Fig 5.4). This is due to the fact that additional features contribute only to file statistics (both min-max and bit vector statistics) and are computed when data is written in the storage layer. The trendline is similar for all tables and we follow the same approach of describing the influence as a linear function.

5.1.5 Final Cost Function

Finally, we combine all results in order to obtain an optimization cost for the whole dataset.:

$$OPTIMIZE_{COST} = dataset\_size * (base\_cost + m_{ZOrder} * num\_features_{ZOrder} +$$

$$m_{add} * num\_features_{add} + sop\_base\_cost + m_{SOP} * num\_features_{SOP})$$

The expression estimates a cost based on data size expressed in GB. In the end, this cost is compared with the reading cost of the volume skipped after optimization. The equation

does not consider the cluster size. Both optimization and reading costs were determined on the same cluster setup and we assume that the values grow or decrease proportionally when the cluster size changes. Of course, another assumption is that workloads and clustering are performed on the same cluster.

5.2 Benefit Estimation

The purpose of data layout reorganization is to improve data skipping, which minimizes the amount of scanned data at query time. Thus, it is logical that the optimization benefit should be measured by the improvement in data skipping capabilities on the current workload. This can be estimated in terms of number of files read or scanned volume. Therefore, a direct comparison between scanned volume by the queries in the workload, before and after optimization, would give a clear sense about the scale of data skipping improvement. The challenge here lies in estimating the improvement before performing the optimization, with minimum time and resource costs.

The ideal solution would be to predict the file statistics after optimization. Then, it would be just a matter of verifying what files are read by each query. However, computing the statistics beforehand has no easy or cheap solution. We can opt for making a best case scenario estimation, in which we consider each dimension perfectly clustered. But even then, it requires knowledge on the value distribution of that dimension in order to know what min-max ranges each file would have. This cannot be achieved without scanning the entire dataset multiple times which is already an unacceptable price.

In order to avoid full data scans (even of only the newly added data), we can fallback to making an estimation using a data sample. This gives the possibility of computing accurate metrics for the sample but, of course, its capability of characterizing the whole dataset depends on the data and the sampled tuples. However, we consider that the estimation on a sample is sufficient for providing a clear picture over the possible improvement.

Because we need the concept of files for measuring data skipping, we propose a method that introduces *virtual files* in Spark. These are in-memory file abstractions generated from DataFrame partitions and are able to hold information about the size and data statistics that would normally be computed on real files. Using them, the optimizer can simulate performing the queries in the workload and obtain metrics on data skipping. We make the assumption that without statistics, the queries would have read the whole sample each time. Then we can compare the data skipping measurements against this baseline and compute the improvement. The steps for this method are described below:

Given a workload $WL = \{Q_1, Q_2, \ldots, Q_w\}$ where each Q_i is a query containing $F_i = \{F_i1, \ldots, F_iqi\}$ filters and the sets Z and S containing the chosen Z-Order and, respectively, SOP features, we compute the $OPTIMIZE_{improvement}$ as follows:

Benefit Estimation Method

P1. Extract *Samp*, a sample DataFrame from the dataset.

P2. Optimize *Samp* with features from Z and S and obtain $Samp_{Optimized}$.

P3. Foreach p_i in RDD_Partitions ($Samp_{Optimized}$)

Create a virtual file V_i with Min-Max and SOP statistics;

Compute the size of each column C_j in V_i and add it to statistics as well;

P4. Foreach Q_i in WL, evaluate the filters in F_i against the statistics of all virtual files and obtain the scanned volume:

$scan_{Q_i} = \sum_{k=1}^{m} \sum_{j=1}^{p} size(C_j)_k$, where

$C_{1,..p}$ are the columns from the read schema of Q_i

$V_{i1,..im}$ are the virtual files read by Q_i after applying data skipping

$size(C_j)_k$ represents the j-th column size in virtual file V_{ik}

P5. Calculate

$scan_{WL} = \sum_{i=1}^{w}(scan_{Q_i})$ - the total scanned volume by the workload WL after optimization and $scan_{WL-initial} = sum(scan_{Qi} - INITIAL)$ - the total scanned volume by the workload WL before optimization (when all virtual files are read all the time)

$P_{scan} = \frac{scan_{WL}}{scan_{WL-initial}}$ - the fraction of scanned volume after optimization compared to the one before optimization, on the data sample

Finally, compute:

$$OPTIMIZE_{improvement} = (1 - P_{scan}) * size(Dataset) * COST/MB$$

P_{scan} is the scanned percentage from the sample dataset. We extrapolate and consider that the same percentage applies to the entire dataset. Following, $1 - P_{scan}$ is the estimated pruned fraction of data. Therefore, the whole pruned volume becomes $(1 - P_{scan}) * size(Dataset)$. We then transform this value to cost units in time or money by multiplying the scanned size in MB with the cost of reading 1MB, which is the constant COST/MB.

NumZ-Ord\$NumSOP$	0	4	8	12
0	1.02	1.03	1.1	1.44
1	1.11	1.08	1.11	1.32
2	1.22	1.29	1.29	1.4
3	1.23	1.37	1.29	1.30

Table 5.1: Benefit Estimation Error Rate. Estimation is relatively accurate (always less than a factor 1.5 o□)

In other words, the algorithm estimates the amount of pruned data as a result of the optimization and translates it in units of time.

We conducted a set of experiments to assess the accuracy of the benefit estimation. We compared the estimated volume of skipped data with the real skipping improvement. The setup included creating a workload and extracting a set of clustering features. For the real values, we clustered the table and measured the skipped volume. The table we used is 60GB and has a schema of 8 columns, including numeric and string values. Table 5.1 records the estimation error rate, determined by the difference between the real and estimated values, in bytes. We tested with increasingly bigger number of Z-Order and SOP features, in all combinations. The errors vary from a factor of 1.25, up to 1.44. We observe how the estimation looses accuracy at higher numbers of clustering features. However, such differences are expected since there are multiple sources of inaccuracies in the algorithm:

- Estimation is dependent on the data sample
- We assume the current data has little to no skipping capabilities for the selected features
- Volume estimations are made strictly on data size, whereas in reality Parquet uses complex structures and compression schemes

6
Performance Evaluation

Real data has complex correlations that are difficult to simulate by benchmarking frameworks. As we want to capture these patterns and realistically assess our approach, we built a custom evaluation framework that operates on real data and workloads.

6.1 Evaluation Framework

As the subject table, we used one of the most frequently queried internal tables in Databricks, on which workload information statistics were collected over a period of time. These workload statistics are regularly queried by meta-statistics generating queries, or by standing streaming queries that feed into certain system health dashboards. Databricks developers sometimes also perform ad-hoc analysis queries on these stats tables for various purposes (e.g. problem identification). Databricks does not run customer query workloads (they run in the customer account) and so for both technical and privacy reasons it is impossible to use Databricks customer workloads for the purpose of this thesis. Hence we were happy to be able to use this internal workload. The specific content of the tables and the queries does not matter here per se, rather the fact that this is a sizeable real-work workload that can be analyzed for this project.

We first extracted information about the pushed down filters that were used in real queries. Then we obtained the inserted data over the same period of time. The final setup replays the scans and insertions in the same order they happened in reality. It is worth noting that we are not replaying whole queries, but rather only the scans with their corresponding filters. Also, the inserts contain the original data, in the same amount, but they do not maintain the same file structure, meaning that the file sizes may differ as they depend on the cluster size (parallelism level).

The total data amounts to 5TB and 135 scans. The cluster we used has 10 workers with 8 cores and 61GB of RAM and a driver with the same specification. The instances also have Delta caching enabled, which speed up processing time by caching recently accessed

data. The cluster sums up to 671GB and 88 cores. We replayed the workload for three different setups:

- when no optimization takes place
- when we run an OPTIMIZE ZORDER BY halfway through the queries.
- when we enable the automatic layout optimization. The optimizer then decides by itself when clustering is necessary and performs it automatically.

6.2 Results

We measured the performance in terms of scan time, volume read, insert time and total time (scans and inserts), which are plotted in Fig. 6.1.

In terms of scan time, we observed a 3.5 speedup compared to normal version and 1.6 compared to Z-Order. For volume read, we obtained a 2.9 and 1.5 improvement. For insert time, it was expected that the automatic solution is much slower, as it also performs clustering. The total insert time is 5 times slower than no clustering version and 2 times slower compared to one simple Z-Order. However, when adding both query time and insert time, the final speedup is still high: 3.2 and 1.5 respectively. This is due to the high cost of the scans in comparison with the time that clustering takes.

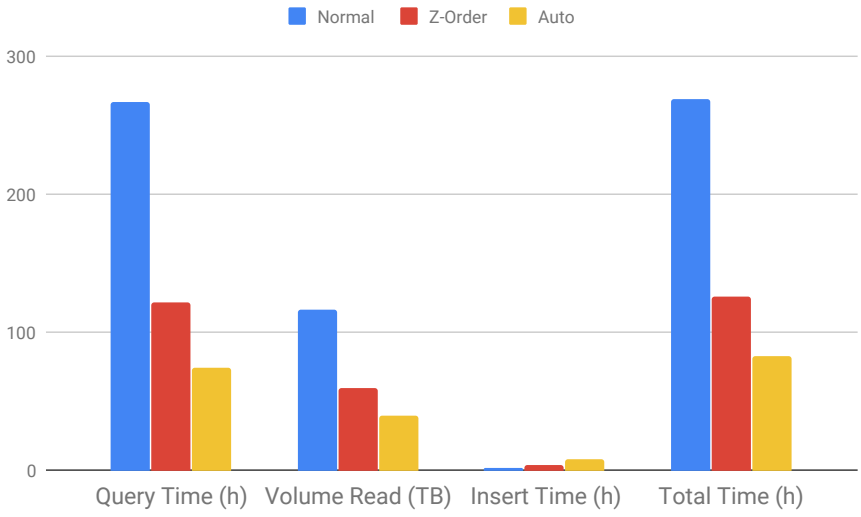


Figure 6.1: Running time comparison between three configurations: no optimization, one manual Z-Order run and automatic optimization.

Although the speedup over no clustering might be comparable at any point in the future, the improvement compared to Z-Order happens only when the Delta OPTIMIZE command is actually triggered regularly.

When analysing the automatic feature choice, we observed first that the most relevant feature is a column that is currently used for Z-Order but, additionally, there were important predicates under complex expression formats (most frequently map column type extractors). Therefore, the improvement over normal Z-Order stems mainly from the new clustering scheme that enables many more types of predicates. Of course, another contribution is the automatic mechanism that schedules data reorganization exactly when it is necessary, compared to manual Z-Order.

We then examined how insert time is spent in the case of automatic layout and plotted the results in figure 6.2. Regular file insertion takes 17.3% of time. This amount of time is constant across all three experiments, as all data needs to be written to disk first, before any layout optimization takes place. Then, by far the most time is spent on reclustering data. This includes reading it and writing it back down to S3. Workload analysis time is the least expensive and takes only 1.1% of the total time. Scaled to the real duration, it translates to an average of 15 seconds. The benefit estimation function is slightly more costly and expands to 2.1% of the total time. Both workload analysis and the cost benefit estimation incur considerable latencies from extracting data samples and applying Spark operations on them.

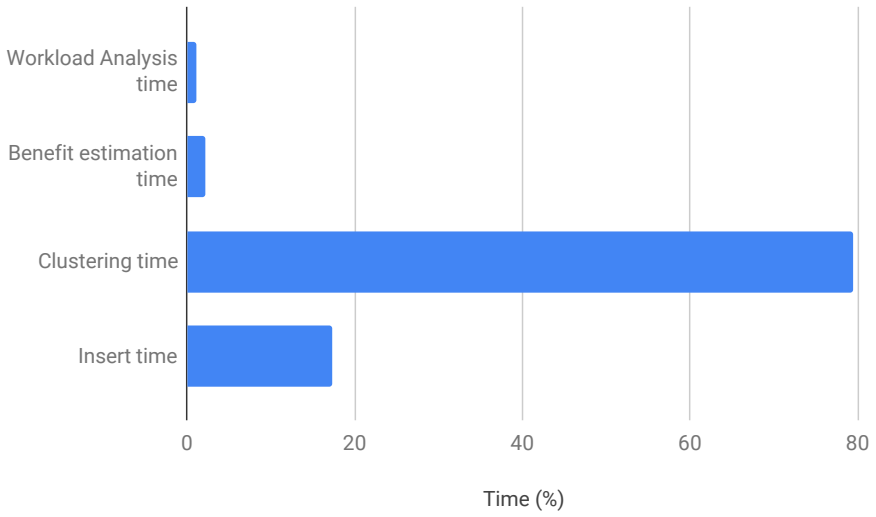


Figure 6.2: How time is spent in inserts.

6.3 Discussion

The project's goal is to adjust the data layout automatically, such that it fits the workload, with the constraint that the benefits should not exceed the costs. Our evaluation framework showed that the system can successfully achieve this goal and, moreover, overcome the current Z-Order solution in Delta. This proves that both the automatic mechanism and the new clustering approach positively impact the skipping capabilities.

One of the research directions was to provide support for complex expressions and improve the clustering approach. The results we achieved validates that complex expressions are relevant in the skipping process. Among the most frequent predicates in our evaluation are filters with extractors for complex data types, which previously could not be used in clustering nor skipping. By combining SOP with Z-Order, we were able to use more features and thus further increase the skipping capabilities. However, the clustering time with our approach is bigger than performing only Z-Order, which is justifiable by the introduction of SOP algorithm. In our approach data is read in its entirety twice, compared to only once in the Z-Order approach.

Given the filter queries, the Workload Analyzer made sensible decisions, extracting the most promising features. A manual inspection of the workload confirmed that the chosen features were among the most frequent. There were also a few correlations detected, which were not straightforward to the naked eye. Two of them were among the Z-Order features, which are especially valuable due to their limited number. Therefore we consider that having the correlation test brings significant improvements.

The Cost-Benefit module proves to be a key element in automating the physical design. By taking into account the clustering cost and the potential benefit, it is able to estimate when it pays off to trigger the optimization. Even though the data sample approach is prone to inaccuracies, the decisions made by this module in the evaluation are reasonable and proved to be effective. The clustering was triggered only when enough new data (a few GB) was accumulated and the frequency and scale of the workload queries justified the optimization.

Given the results we achieved, we consider our automatic solution to be successful. There is, of course, room for improvement in most areas. One direction that can have big potential is to explore other methods for combining SOP with Z-Order. This can have impact on both clustering time and skipping effectiveness. We discuss it further in the Future Work section.

7
Conclusion

7.1 Answering the Research Questions

With this thesis we explored the area of an automatic data layout optimizer, that clusters the data based on information extracted from a specific workload. We propose new algorithms and method for clustering, feature and data correlation detection and cost-benefit estimation. We integrated the solution in Databricks Delta and achieved more than 3x speedup.

Below we go through the research questions again and specify how we answered them in this project.

Q1 *Can we obtain a good data layout based on Z-order and SOP clustering? How can we best combine them?*
Yes, we identified strong and weak points for both solutions and proposed a way to combine them and gain the benefits from each one of them, which sum up to:

- better scaling in terms of number of clustering features
- broader predicate coverage, that includes complex expression filters
- overcome the SOP shortcoming on one particular filter pattern, by prioritizing it for Z-Order instead

Our evaluation showed better results with this new clustering scheme compared to only Z-Order.

Q2 *Given a workload, what are the most relevant features and how can we automatically extract them? Can we detect correlations between them and minimize the set of features accordingly?*
We implemented a monitoring infrastructure that records information about the filter

queries. Then, we proposed an algorithm that parses the logs and extracts relevant features candidates, separately for Z-Order and SOP. It also detects data correlation, using a data sample approach and a series of correlation metrics. We extract the final set of features by building a correlation graph and having a way to traverse it, such that it prioritizes the most relevant features.

Q3 *How can we enable data skipping on complex expressions?*
In SOP, complex expressions are treated the same as any other predicate. Skipping is done based on bit vectors. Therefore, we added this new type of file statistics to the Delta Log. Moreover, we enriched the current min-max statistics and allowed them to be computed on complex expressions as well. Also, we appended additional statistics, for both min-max values and bit vectors, on features that were not used for clustering, but are relevant for skipping.

Q4 *When does it pay off to trigger automatic data layout optimization and how can that be achieved in a Big Data system where storage is decoupled from compute?*
We answered this question with the cost-benefit module, that can estimate the optimization cost and benefit before clustering. Based on this information, the system can decide when it is optimal to perform a layout change. Because performing this operation depends on having a running cluster with write permissions, the automatic triggering mechanism that we use is to piggyback the optimization on writes. This incurs sensible overhead during reclustering, but the user can be informed about the reasons and the expected benefits.

Q5 *How can we meaningfully evaluate our optimizer?*
Real world data and workloads have patterns and correlations that are hard to simulate by benchmarking tools. Therefore, we built an evaluation framework that performs evaluation on data and queries extracted from a frequently used Databricks internal table. We replayed filter queries and inserts in the way they originally happened and compare the running times of three different setups: simple replay, issuing an OPTIMIZE ZORDER BY command halfway through the queries and enabling our automatic layout clustering solution. We achieved significand speedup over both no optimization and simple Z-Order versions.

7.2 Future Work

In this thesis, we experimented with a new clustering approach and a method of approximating when clustering is necessary, which enabled implementing an automatic solution. While our estimations and methods could certainly benefit from improvements in order to make them more accurate and efficient, in this chapter we discuss what other directions can the automatic layout optimizer take for considerable impact.

First of all, there is one important area that was left unexplored by this project: changing workloads. Our approach is basically equivalent to having a sliding window in time, in which all new files are layout optimized according to the workload in that time period. However, there is no guarantee that the current workload does not touch older files and, moreover, that it does not substantially change, such that reading files clustered on completely different features becomes highly inefficient. To address this problem, we would have to continually re-cluster old and new data alike, while satisfying the current workload. The challenge comes from establishing how far in the past should the time window extend, such that the clustering operation does not become prohibitive. Old data can be reclustered incrementally in order to reduce the impact on resource consumption. It can also follow a LSM (58) approach, similar to Cassandra. Moreover, monitoring exactly the access patterns of the workload can help identify only a specific subset of old data that needs to be reclustered.

A second direction for improvement is to make use of the benefit estimator to select the best clustering features. Right now, features are statically selected based on format, frequency and correlation. But the benefit estimation can actually predict the improvement one feature configuration can bring, and of course, try to maximize it. However, this would generate a huge search space, where the current way of estimating would be too expensive. So this calls for alternative solutions for both reducing the number of feature combinations that are analysed and for the way the benefit itself is computed.

Another potential improvement lies in system design details and the way this automatic solution is implemented in distributed Big Data systems, such as Spark. Currently, the optimization checks are piggybacked on writes, which adds a constant small overhead. Moreover, we assume that all operations happen on the same cluster, with the same configuration. But the layout optimization can be run in a variety of ways, both in terms of time and place. Firstly, it may be unnecessary to perform all checks on every write. Maybe only expensive operations (both reads and writes) can incorporate them, which makes the overhead be assimilated much better. Another option is to have scheduled jobs that checks

all tables, or even leverage cluster low utilisation times to do some work, even partially. To hide the clustering overhead altogether, the optimization can take place on dedicates resources.

References

[1] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010. 1, 3

[2] Databricks Delta Guide. https://docs.databricks.com/delta/index.html. (Accessed on 02/22/2019). 1, 5

[3] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *ACM Sigmod record*, 26(1):65–74, 1997. 1, 4

[4] Thomas Stöhr, Holger Märtens, and Erhard Rahm. Multi-dimensional database allocation for parallel data warehouses. In *VLDB*, 2000, pages 273–284, 2000. 1, 8

[5] Daniel C Zilio, Jun Rao, Sam Lightstone, Guy Lohman, Adam Storm, Christian Garcia-Arellano, and Scott Fadden. DB2 design advisor: integrated automatic physical database design. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 1087–1097. VLDB Endowment, 2004. 1, 12

[6] Sanjay Agrawal, Nicolas Bruno, Surajit Chaudhuri, and Vivek R Narasayya. AutoAdmin: Self-Tuning Database SystemsTechnology. *IEEE Data Eng. Bull.*, 29(3):7–15, 2006. 1, 12

[7] Amazon S3. https://aws.amazon.com/s3/. (Accessed on 02/22/2019). 2

[8] Processing petabytes of data in seconds with Databricks Delta. https://databricks.com/blog/2018/07/31/processing-petabytes-of-data-in-seconds-with-databricks-delta.html. (Accessed on 02/22/2019). 2, 8

[9] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: Machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016. 3

[10] Reynold S Xin, Joseph E Gonzalez, Michael J Franklin, and Ion Stoica. Graphx: A resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013. 3

[11] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. Structured streaming: A declarative API for real-time applications in apache spark. In *Proceedings of the 2018 International Conference on Management of Data*, pages 601–613. ACM, 2018. 3

[12] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394. ACM, 2015. 3

[13] Catalyst Optimizer. https://databricks.com/glossary/catalyst-optimizer. (Accessed on 07/20/2019). 3

[14] Project Tungsten: Bringing Spark Closer to Bare Metal. https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html. (Accessed on 07/20/2019). 3

[15] Spark DataFrames and DataSets. https://spark.apache.org/docs/latest/sql-programming-guide.html. (Accessed on 02/22/2019). 3

[16] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010. 4

[17] Apache Cassandra. Apache cassandra. *Website. Available online at http://planetcassandra. org/what-is-apache-cassandra*, page 13, 2014. 4

[18] Apache Parquet. https://parquet.apache.org/. (Accessed on 02/22/2019). 4

[19] Databricks. https://databricks.com/. (Accessed on 02/22/2019). 4

[20] Databricks Cache Boosts Apache Spark Performance. https://databricks.com/blog/2018/01/09/databricks-cache-boosts-apache-spark-performance.html. (Accessed on 07/20/2019). 4, 46

[21] DeltaLAke. https://delta.io/. (Accessed on 07/20/2019). 5

[22] Apache Hive. https://hive.apache.org/. (Accessed on 07/20/2019). 6

[23] Databricks partition pruning. https://docs.databricks.com/user-guide/tables.html#partition-pruning. (Accessed on 02/22/2019). 8

[24] Bishwaranjan Bhattacharjee, Sriram Padmanabhan, Timothy Malkemus, Tony Lai, Leslie Cranston, and Matthew Huras. E□cient query processing for multi-dimensionally clustered tables in DB2. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 963–974. VLDB Endowment, 2003. 8

[25] Volker Markl, Frank Ramsak, and Rudolf Bayer. Improving OLAP performance by multidimensional hierarchical clustering. In *Proceedings. IDEAS'99. International Database Engineering and Applications Symposium (Cat. No. PR00265)*, pages 165–177. IEEE, 1999. 8

[26] Bishwaranjan Bhattacharjee, Sriram Padmanabhan, Timothy Malkemus, Tony Lai, Leslie Cranston, and Matthew Huras. E□cient query processing for multi-dimensionally clustered tables in DB2. In *Proceedings of the 29th international conference on Very large data bases-Volume 29*, pages 963–974. VLDB Endowment, 2003. 8, 11

[27] Jack A Orenstein and Tim H Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 181–190. ACM, 1984. 8, 9

[28] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *IEEE Transactions on knowledge and data engineering*, 13(1):124–141, 2001. 8

[29] Databricks Delta Optimization. https://docs.databricks.com/delta/optimizations.html. (Accessed on 02/22/2019). 8

[30] Liwen Sun, Michael J Franklin, Jiannan Wang, and Eugene Wu. Skipping-oriented partitioning for columnar layouts. *Proceedings of the VLDB Endowment*, 10(4):421–432, 2016. 9, 10, 12, 13, 23, 24, 26

[31] David Eppstein. Two dimensionsl Z-OrderCurve. https://commons.wikimedia.org/wiki/File:Z-curve.svg. (Accessed on 07/20/2019). 9

[32] Robert Dickau. Three dimensionsl Z-OrderCurve. https://en.wikipedia.org/wiki/File:Lebesgue-3d-step2.png. (Accessed on 07/20/2019). 9

[33] Rudolf Bayer. The universal B-tree for multidimensional indexing: General concepts. In *International Conference on Worldwide Computing and Its Applications*, pages 198–209. Springer, 1997. 11

[34] Chee-Yong Chan and Yannis E Ioannidis. Bitmap index design and evaluation. In *ACM SIGMOD Record*, 27, pages 355–366. ACM, 1998. 11

[35] Lefteris Sidirourgos and Martin Kersten. Column imprints: a secondary index structure. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 893–904. ACM, 2013. 11

[36] Stratos Idreos, Martin L Kersten, Stefan Manegold, et al. Database Cracking. In *CIDR*, 7, pages 68–78, 2007. 12

[37] Jingren Zhou, Nicolas Bruno, and Wei Lin. Advanced partitioning techniques for massively distributed computation. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 13–24. ACM, 2012. 12

[38] Alekh Jindal, Jorge-Arnulfo Quiané-Ruiz, and Jens Dittrich. Trojan data layouts: right shoes for a running elephant. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, page 21. ACM, 2011. 12

[39] Ioannis Alagiannis, Stratos Idreos, and Anastasia Ailamaki. H2O: a hands-free adaptive store. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1103–1114. ACM, 2014. 12

[40] Alekh Jindal, Endre Palatinus, Vladimir Pavlov, and Jens Dittrich. A comparison of knives for bread slicing. *Proceedings of the VLDB Endowment*, 6(6):361–372, 2013. 12

[41] Sanjay Agrawal, Vivek Narasayya, and Beverly Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 359–370. ACM, 2004. 12

[42] Surajit Chaudhuri, Ravi Krishnamurthy, Spyros Potamianos, and Kyuseok Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 190–200. IEEE, 1995. 12

[43] Sam S Lightstone and Bishwaranjan Bhattacharjee. Automated design of multidimensional clustering tables for relational databases. In *Proceedings of the Thirtieth international conference on Very large data bases-Volume 30*, pages 1170–1181. VLDB Endowment, 2004. 12

[44] Surajit Chaudhuri and Vivek Narasayya. Self-tuning database systems: a decade of progress. In *Proceedings of the 33rd international conference on Very large data bases*, pages 3–14. VLDB Endowment, 2007. 12

[45] Jun Rao, Chun Zhang, Nimrod Megiddo, and Guy Lohman. Automating physical database design in a parallel database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 558–569. ACM, 2002. 12

[46] Oracle AWR. `https://oracle-base.com/articles/10g/automatic-workload-repository-10g`. (Accessed on 02/22/2019). 12

[47] Snowflake Table Structure. `https://docs.snowflake.net/manuals/user-guide/tables-micro-partitions.html`. (Accessed on 02/22/2019). 12

[48] BigQuery Clustered Tables. `https://cloud.google.com/bigquery/docs/\clustered-tables`. (Accessed on 02/22/2019). 13

[49] Redshift Automatic Analyze. `https://docs.aws.amazon.com/redshift/\latest/dg/t_Analyzing_tables.html#t_Analyzing_tables-auto-analyze`. (Accessed on 02/22/2019). 13

[50] R. M.; Chvátal V.; Cook W. J. Applegate, D. L.; Bixby. *The Traveling Salesman Problem*. 2006. 27

[51] Sanjeev Arora. Polynomial Time Approximation Schemes for Euclidean Traveling Salesman and Other Geometric Problems. *J. ACM*, 45(5):753–782, September 1998. 27

[52] R. C. Prim. Shortest connection networks And some generalizations. *Bell System Technical Journal*, 36(2). 27

[53] Karl Pearson. Notes on regression and inheritance in the case of two parents. *Proceedings of the Royal Society of London*, 58. 34

[54] Wayne W. Daniel. *Spearman rank correlation coefficient*. 1990. 34

[55] M. Kendall. A New Measure of Rank Correlation. *Biometrika*, 30. 34

[56] Ihab F Ilyas, Volker Markl, Peter Haas, Paul Brown, and Ashraf Aboulnaga. CORDS: automatic discovery of correlations and soft functional dependencies. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 647–658. ACM, 2004. 34

[57] H. Cram. *Mathematical Methods of Statistics*. Princeton, 1948. 34

[58] Leveled compaction in Apache Cassandra. https://web.archive.org/web/20140213134601/http://www.datastax.com/dev/blog/leveled-compaction-in-apache-cassandra. (Accessed on 07/20/2019). 59