Fragment Skipper Grid Formation: Clustering-Based Horizontal Partitioning

Aparajeeta Jha
Otto-von-Guericke-University
Magdeburg, Germany

Iancu-George Verghelet
Otto-von-Guericke-University
Magdeburg, Germany

Ismail Wahba
Otto-von-Guericke-University
Magdeburg, Germany

Khaled Tawfik
Otto-von-Guericke-University
Magdeburg, Germany

Pawan Joshi
Otto-von-Guericke-University
Magdeburg, Germany

Sharanya Hunasamaranahalli Thotadarya
Otto-von-Guericke-University
Magdeburg, Germany

***Abstract*—The non-stop increase of data volumes encouraged the development of many techniques to better access information on databases. One of the main techniques is data partitioning, where data is divided in a way that enables accessing relevant information more efficiently.**

In this paper, we try to re-implement SOP, a fine-grained data partitioning framework intended to maximize the number of unnecessary data skips using metadata collected about query workloads, following an empirically proven assumption that the majority of workload queries use a small subset of predicates. The framework aims to recognize the pattern in predicates accessing and use frequent feature sets to group data in smaller blocks. Then, by augmenting data using features, partitioning can be carried out using feature vectors. The framework was presented in a previous research paper but with a private implementation.

We implemented the four steps; parsing, featurization, data augmentation and clustering from the framework using Apache Spark, Apache Parquet and HDFS and compared our results to a baseline table that does not use any partitioning techniques and also a partitioned table with normal queries. We were able to show a definite increase in the number of data tuples being skipped and achieve a reasonable performance compared to the baselines.

I. Introduction

These days, data has become a very important asset. Individuals can make informed decisions based on data collected on their smart devices, companies can achieve better profits and gain a competitive edge by deeply analyzing market-related data and researchers can study phenomena and discover hidden patterns. This is why data engineering, in all its forms, gained attention as a field, to study how to efficiently store data for high-performance analysis. The continuous increase of data volumes made the task of querying data efficiently in analytical systems a hard task to achieve, especially in a dynamic world where workloads change rapidly and consequently data organization needs change as well.

One of the main factors that plays an important role in optimizing the performance of database systems is the configuration of physical indices. When a column, or group of columns, is used as an index, this means that rows will be stored physically on disk or in memory based on their index value. Although this helps greatly in increasing throughput and query response speed, it faces a problem of adaptability. Since the design of databases reflects the profile of queries that are using them, it is very hard to keep adapting the physical design of databases to continuously changing workloads [4].

In response to the change of workloads in database systems, new techniques appeared that can adapt to these changes. Concepts like data skipping and partition pruning evolved. By collecting samples of queries and running sufficient analysis with the help of machine learning and data mining models, data can be divided in a way such that queries can access directly only blocks that contain relevant data, while unnecessary blocks of data are excluded.

In this paper we seek to reproduce the work of Sun et. al. [7], which introduces a fine-grained data partitioning framework that featurizes predicates from query logs, partitions the data based on feature vectors and finally tries to maximize the number of data blocks that can be skipped when processing queries. We select this work for our study, since it constitutes one of the leading state of the art solutions for data partitioning, and it could serve as a strong baseline for further research on partitioning. In addition, a public implementation of this work is needed, for better understanding technical aspects that are not exhaustively described by the authors (e.g. details on the workload analysis and partitioning process).

Our main contributions are:

1) We reproduce the performance improvements achieved by Sun et. al. [7] using hierarchical agglomerative clustering for aggressive data skipping in large datasets. We also describe in detail

our solutions for a few implementation steps that were not explained in detail by the authors.

2) We implement our framework with a modern version of SPARK, instead of SHARK (used by Sun et.al), which has a better interface with SQL. As a result, we provide a more up-to-date solution.
3) Finally, since the original implementation and dataset used by the authors were private, we provide an open-source implementation of or work.

The remainder of the paper is structured as follows: In Sec. II, we summarize 4 related works on modern data partitioning. With this review we provide a research context to the work we reproduce in this paper. In Sec III we introduce necessary definitions to understand in detail the subsequent design, which is presented in Sec. IV. With our design we not only review the work of Sun et. al. [7], but we also introduce our novel solutions for the featurization steps, which are not made entirely clear by the authors. In Sec. V we expand upon our design with implementation details. We also describe in this section our experimental setup. We present our evaluation in Sec. VI. Finally, we conclude in Sec. VII, by summarizing our work and proposing future research directions.

II. Related Work

This section will discuss recent research papers that exploited the topic of data partitioning and the different techniques for obtaining high performance and efficient response time for queries in database systems. Each of the discussed papers is going to be related to the taxonomy presented by Malysheva and Prymak [4], including the paper by Sun et. al. [7] which we base our project implementation on. In particular, the papers selected represent four different approaches for modern data partitioning, which we can name as follows: Clustering-based, Graph-based, Repartitioning-oriented, and a Reactive strategy-based approach. The first two can be performed offline, and assume a complete knowledge of the workload. The last two are expected to work online, with repartitioning solutions seeking to reduce re-partitioning and relying on cost models, and the latter focusing on strategies to act reactively in the presence of unbalanced partition usage.

A. Skipping-oriented Data Design for Large-Scale Analytics

Our project was inspired by a paper by Sun et. al. [7]. This paper introduces a fine-grained partitioning framework that is called skipping oriented partitioning framework, or SOP, which utilizes a horizontal partitioning scheme. This paper represents a clustering-based approach to the task.

In further research, two enhancements are proposed to SOP. The first one is called Generalized SOP, it's motivation is to combine both vertical and horizontal partitioning schemes. The second is an extension of Generalized SOP to include replication, it exploits the copies of data for reliability and availability purposes. For the sake of our project we focused only on the basic SOP framework.

The motivation behind SOP is to improve the way data is partitioned into data blocks, because it plays a main role in the efficiency of data skipping. Many of the previous techniques adopted a range partitioning scheme, but SOP was able to out-perform them by generating fine-grained data blocks that would maximize the number of partitions skipped. Based on an empirical analysis that they carried out on real world workloads, they concluded that many queries use similar filters and that queries use the same filters many times. Under this assumption, the framework is built consisting of four main steps: Workload analysis, featurization, reduction and partitioning.

They start off with *workload analysis*, by collecting query predicates from the workload and filter out the most frequent ones using association rules learning. Redundant predicates were removed in a predicate augmentation task, based on the idea that the more general predicates include the same information of more specific predicates. Then using the filtered set of predicates, features are extracted such that each feature might be a single predicate or multiple predicates combined. In the *featurization* step, data tuples are scanned and evaluated against these features to mark the presence and/or absence of features per tuple. Then the process enter a *reduction* step, where data is reduced into tuples of vectors and counts before it is fed to the partitioner, which creates a partitioning map using Ward's method to decide how the different partitions will be created. This partitioning is then applied to the data.

For the evaluation of the framework, experiments were conducted using an Amazon Spark EC2 cluster and the datasets were stored in HDFS. They apply it on benchmark datasets like TCP-H with a scale 100 in a uniform and skewed distribution of predicates and on a proprietary Conviva dataset against full table scans and 2 different range partitioning techniques. They were able to achieve better performance in terms of percentage of data blocks accessed and query response time, that it reached up to 5x improvement against a full table scan and 3-4x improvement against the range partitioning techniques.

B. Schism: a Workload-Driven Approach to Database Replication and Partitioning

Apart from clustering-based solutions, graphs can also be used to represent the correlation between tuples with respect to their co-access by queries. A novel work called Schism [2] serves as a representative of this approach. Schism is a static, fine-grained, graph-based partitioning tool that aims at leveraging the scalability of shared-nothing distributed systems. Distributed transactions are considered to be expensive for OLTP systems because usually OLTP systems access very few records and rarely require scans of tables, so it is less expensive to carry out operations on these few records on one node than adding the overhead of multiple nodes' communication over the network to the process. The main goal of Schism is to minimize the number of distributed transactions for OLTP systems while trying to build balanced partitions.

Schism follows a two-step process: the first step is building the graph based on a workload analysis and the second step is an explanation phase for how the partitioning was done. In the first step, Schism starts building the graph by representing each of the data tuples as a node, then creates links between them based on the transactions. Then using the workload, for each transaction accessing more than one tuple, a link is created between these tuples. Links are assigned more weight when the number of transactions accessing the same tuples increase. After the whole graph is finished, Schism adopts the graph theory min-cut approach to dissect the graph into K disjoint clusters while obtaining an overall minimum cost of cutting. The final result of the graph is a fine-grained mapping between each tuple and the partition it was assigned to. This mapping was then stored as a lookup table in memory for direct access of tuples. A middleware routing component was used to compare WHERE clauses of incoming queries to the partition numbers from the lookup table. The explanation phase uses a decision tree model to try to capture the mapping between tuples and partitions. The branches of the tree would represent the set of rules, in the form of predicates, that describe how tuples should be assigned to partitions.

To evaluate Schism, a comparison was made in terms of the distribution of processes against manual partitioning that was done by the researchers, full replication of the data on all nodes and hash-based partitioning of the data using the primary key of data. For the evaluation process, many datasets and workloads were used with different configurations including YCSB, TCP-C, TPC-E, and a randomly generated workload. Schism proved to be the second-best performing technique after the manual partitioning, while outperforming automatic partitioning techniques.

C. *Advanced Partitioning Techniques for Massively Distributed Computation*

Other than offline solutions, which assume a complete knowledge of the expected workload, authors have also researched online solutions, which seek to react to changes in the workload. Solutions in this domain can be cost-model based, or also use more sophisticated heuristic strategies, such as greedy behavior.One approach in this line of work, which relies on cost models is proposed by a team from Microsoft [9]. They describe how advanced horizontal partitioning techniques can be used dynamically to optimize processing in distributed systems. Their main focus was to optimize the process of data shuffling, either by completely avoiding re-partitioning, if possible or by carrying out only partial re-partitioning if shuffling is inevitable. This would make it possible to efficiently move data across the network, which is one of the most expensive processes when it comes to query processing in distributed systems.

The paper used two of the most common partitioning techniques, which are Range-based partitioning and Hash-based partitioning. Range-based partitioning splits the data based on the partitioning columns into disjoint blocks, while Hash-based partitioning utilizes a hashing function on the partitioning columns and the result of the function determines to which partition a row will be assigned. They also introduced a new partitioning method called Index-based partitioning which considerably improved efficiency of I/O. Index-based partitioning, instead of partitioning the data, adds a new column 'pa' to each tuple containing the value of its partition number, after which a stable sort operation is carried out on 'pa' ordering all tuples by partition while maintaining the order of tuples inside each partition. Then it is stored in a B+ tree structure in memory. In this way accessing records of a partition can be done using a key-value instead of locating files and reading their content.

Finally, the different techniques are integrated into the query optimizer of Microsoft SCOPE, which uses a cost-based comparison to choose the most efficient re-partitioning method to apply during run-time. As results for their experiment, they managed to achieve 6x faster results, in terms of latency and total work of query, by using re-partitioning compared to traditional full-partitioning. Also comparing the newly introduced Index-based partitioning to the default re-partitioning of SCOPE, a script was run for re-partitioning while varying the target number of partitions. Though the new technique showed less performance for less than 500 partitions, it was 1.6x faster for increasing partitions up to 4000.

D. *E-Store: Fine-Grained Elastic Partitioning for Distributed Transaction Processing Systems*

The final paper we review, which we selected to represent reactive strategy-based solutions introduces a new elastic partitioning structure for distributed OLTP DBMS called E-Store. E-Store manages its resources based on being able to detect the unanticipated changes in the application's workload automatically and avoid bottlenecks through a two-tier data placement plan of cold and hot tuples. OLTP applications are always facing an unexpected change in demand, more specifically, web-based applications that manage a vast amount of requests every day considering various factors. Hence, the system should be elastic, to be able to handle all the different workloads while still ensuring the application performance, throughput and latency wise, and preserving ACID properties for transactions.

The store architecture is composed of 3 parts: E-Monitor, E-Planner, and Squall. E-Monitor communicates with the Distributed OLTP system, to recognize load imbalance and identify hot tuples through collecting data statistics of the read/write access count. E-Planner uses this information to divide the cold tuples into large disjoint sets, assigning the hot tuples into the appropriate partitions to even the distributed load, and at the end, distribute the cold tuples over the remaining capacity. Squall uses the reconfiguration plan made by E-Planner to move the data physically while transactions are being executed.

A comparison was made between One-Tiered and Two-Tiered Partitioning to evaluate E-Store efficiency by comparing application throughput and latency. For this experiment, the benchmarks that were used are YCSB and TPC-C; also, different model plans have been implemented, such as First Fit, Greedy, and Greedy Extension. The main goal of the experiment is to reach the same results as no-skew case even if there's skew. As results, both algorithms outcome was almost the same in case of low skew, but latency was achieved faster in case of the two-tiered approach. Also in the case of high skew, the two-tiered method performs better in general as it can quickly identify the workload imbalance and adapt to it.

To close this section, we note that many approaches exist for data partitioning, representing diverse expectations about whether the partitioning will be done online or offline. In this paper we will focus on offline approaches, and we study a clustering-based solution, which represents an approach that is straightforward. Other approaches would be interesting to study as well.

III. Background

After providing a perspective on the approaches in the field, in this section we introduce fundamental concepts and background for the techniques that we adopt in our study.

A. Predicate

In a database context, a predicate is an expression that evaluates to true or false. It is used in the "Where" condition of SQL queries to filter data selected from tables. For example in the query "Select * from salary where name like 'John'", only records that evaluate the expression "name like 'John' to true will be returned, while the others will be excluded.

B. Predicate Set

A predicate set, is any set of conjunctive predicates, where the conjunction is denoted by an "and" in database context. "name like 'John'" is a predicate set of only one element, while "name like 'John' and age >= 18" is a two element predicate set.

C. Subsumption Relation

Two predicates are considered to be equal if they use the same columns, operators and constants.

On the other hand, a predicate P_i is said to subsume another predicate P_j, if P_j is stricter than or equal to P_i. For example, if P_i is "age $\geq$ 18" and P_j is "age $\geq$ 50", then P_i subsumes P_j or $P_i \supseteq P_j$.

For predicate sets, P_i and P_j, P_i is said to subsume P_j if $P_i \subseteq P_j$, or if any predicate P_y in P_j is subsumed by predicate P_x from P_i. For example, if P_i is "age $\geq$ 50" while P_j is "name like 'John' and age $\geq$ 50", then P_i subsumes P_j because P_i contains all people of age 50 years whether they have the name John or not.

D. Data Fragmentation

Data fragmentation is the process of breaking down a relation, or a table, into smaller entities called fragments, each fragment is treated as a unit [5]. The motivation behind data fragmentation is to define a more suitable unit other than a whole table. In most cases, applications view only subsets of tables and not whole tables. Data fragmentation allows concurrent transactions execution and parallel distributed execution of single queries, hence increasing the level of concurrency and the throughput of the system.

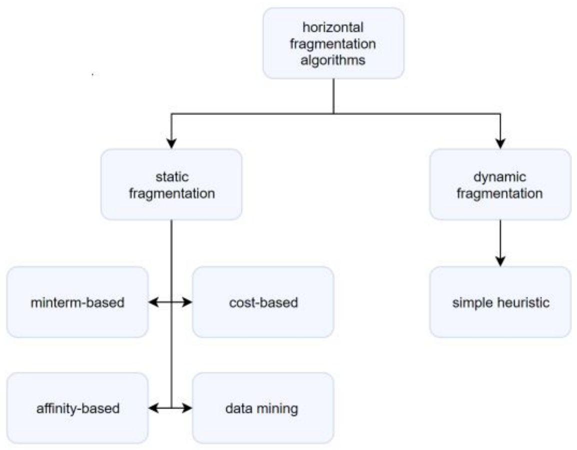


Fig. 1: Horizontal Fragmentation Taxonomy, as described by Malysheva and Prymak [4]

E. Horizontal Fragmentation

Horizontal fragmentation or horizontal partitioning is one of the types of data fragmentation. It involves dividing a table, horizontally, into multiple disjoint fragments. Each of these fragments is assigned boundaries using values from the columns of the table. Then based on the boundaries defined, the DBMS can assign tuples to the fragments using different techniques like range and hash partitioning [6]. Several heuristic-based algorithms are used to apply horizontal fragmentation, these can be shown in Figure 1 introduced by Malysheva and Prymak [4]. They present a taxonomy with two main approaches for horizontal fragmentation. The first is Dynamic Fragmentation, which conforms to cases where there is a changing pattern of user queries. The second approach and the focus of our project is Static Fragmentation, which depends on known collected workload of queries that is usually fixed over time. We will discuss each of the algorithms under the static fragmentation category.

1) MinTerm-based Horizontal Fragmentation

It is the creation of fragments based on conjunction of simple predicates.

2) Cost-based Horizontal Fragmentation

This method of fragmentation is based on creating multiple fragments plan, and choosing the best one using a cost model.

3) Affinity-based Horizontal Fragmentation

This algorithm is constructed in two steps, first is to cluster simple predicates together based on their affinities, then the predicates that belongs to the same group are conjuncted together to form the horizontal fragment.

4) Data Mining Horizontal Fragmentation

This approach covers many data mining technique such as clustering approach, which we will be using in this report.

From the work we discussed in Sec. II, offline papers are data mining-based, with most of them using some form of cost models, and Schism being a work that is close to affinity concepts. Overall this taxonomy also helps to explain previous work.

F. Aggressive Data Skipping

Aggressive data skipping is a concept introduced by Sun et. al. [7], which involves applying horizontal and vertical fragmentation in data systems. The fragmented data is stored in data units called blocks along with metadata which describes the data inside them. This technique can help database systems to aggressively skip many blocks of data that contain data not related to queries [8].

IV. Design

This section is devoted to the details of the two main phases that constitute our project, workload analysis and partitioning, as it is here were our core contribution lies. We will discuss in sequence the steps that compose each of the phases and for each of the steps we will explain the purpose, algorithms and techniques used, constraints considered and finally output produced. These constitute our choices to reproduce the work of Sun et. al. [7].

A. Workload Analysis: Parsing

Parsing represents the first part of the workload analysis. This process aims to extract the relevant information out of the generated queries. So, each query represents either single filter information or a set of information. This process will be further useful for finding the features in the next part that is the featurization part of workload analysis. To extract information from queries, we only take into account the "where" clause. This clause consists of a predicate or a set of predicates.

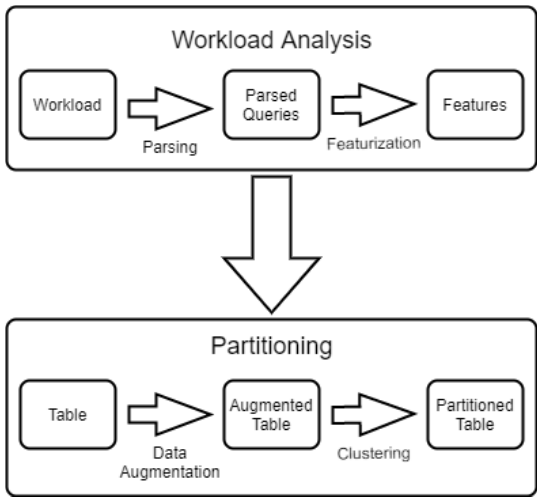


Fig. 2: Overview of our framework

Given below is a simple SQL query that contains predicates that correspond to several other tables. For our evaluation and for this example we are only considering the predicates which belong to the Line-item table, as it is the largest in the benchmark.

```
select
from
  customer,
  orders,
  lineitem
where
   and c_custkey = o_custkey
   and l_orderkey = o_orderkey
   and o_orderdate < date '1995-03-01'
   and l_shipdate > date '1995-03-01'
```

For the above query we have only one predicate that touches the Line-item table, so we will consider that filter predicate and process it further.

Given below is the demonstration of how predicates are extracted from the above query and stored in an organized way.

```
 {
      "Predicates":[
         {
            "Operator":">",
            "ColumnName":"l_shipdate",
            "Value":"DATE '1995-03-01'"
         }
      ]
   },
```

Here, the predicates are stored in a special structure where we break the predicates into "Operator", "ColumnName" and "Value"; where "Operator" denotes the set of operators that are used in a filter predicate or set of predicates, "ColumnName" reflects which column has been included in a predicate and "Value" represents the value that a particular column takes in the query.

B. Workload Analysis: Featurization

Featurization constitutes the second part of the workload analysis. The goal of this process is to find the most representative predicates that appear inside the workload. A set of predicates that is representative of the workload will be called a feature. A feature is considered to be representative when it can be used to help a large number of queries to skip data. Features can contain one- or multiple predicates. A feature with multiple predicates is formed in the case where two or more filters that appear together in a large number of queries are found. This process will be further dived into three steps. The first one, named predicate augmentation, will prepare the queries to be used by a frequent-itemset mining algorithm. The second step is represented by a frequent-itemset mining problem, where the non-representative predicates will be removed. The remaining filters will be grouped into features. The last step will handle some of the inconsistencies that were introduced in the predicate augmentation step and remove redundant features.

1) Predicate augmentation

In order to find the predicates that help the maximum queries to skip data, we have analyzed their subsumption relations with queries. If a given predicate set subsumes a query it means that the query will be able to skip the same data blocks that can be skipped for the given predicate set. In order to select representative predicate sets we should find sets that subsume a lot of queries. A classic frequent-itemset mining algorithm like the apriori algorithm uses the number of occurrences or frequency of items as a selection criteria. The number of times a predicate set appears is different than the number of queries a predicate set subsumes. By augmenting each query with the predicates that subsume it we can achieve a state where the frequency of each predicate is equal to the number of queries it subsumes. After this, we can apply a classic itemset-mining algorithm to find the most representative filter sets.

In order to augment the queries, we iterate over F which represents the set of all predicates extracted from the queries. In the case where the predicate f subsumes a query and the query does not already contain it, we copy it to the query. This procedure is described in *algorithm 1*.

After applying this process we reach a state where each predicate has the occurrence number equal to the number of queries it subsumes. At this point we can apply frequent itemset-mining, for example in the form of the apriori algorithm.

Algorithm 1 Predicate augmentation

```
Result: Augmented Query Set
$F$ = Set of all predicates  $Q$ = Set of all queries  foreach $f_i \in F$ do
    foreach $q_j \in Q$ do
        foreach $f_k \in q_j$ do
            if $f_k \sqsubseteq f_i$ then
                $q_j := q_j \cup f_i$
            end
        end
    end
end
```

2) Frequent Itemset-Mining

The main goal of a frequent-itemset mining algorithm is to find items that cooccur. In the context of such algorithms, the terms items and transactions are often used. In this specific case, an item represents a predicate and a transaction represents a query. In simple terms, a frequent-itemset mining algorithm finds all the itemsets that appear in at least N transactions. The threshold N is called support.

The apriori algorithm makes use of the apriori property that states that all subsets of a frequent itemset are also frequent. By reversing the property we can conclude that all supersets of an infrequent itemset are infrequent. As described in [1], we start by finding all the one-item sets that are frequent. After that, in the next iterations, we look at all the combinations of these items that also satisfy the condition of having the minimum support. Itemsets that are infrequent will be pruned. By doing early pruning we avoid visiting infrequent supersets. The support we chose will dictate the number of features we will have at the end of this step.

After applying the apriori algorithm we will end up having features that contain redundant predicates. These redundancies are a result of the predicate augmentation step. The last step of the featurization process handles the removal of the redundant predicates.

3) Redundant predicate set removal

After completing the frequent-itemset mining step we obtain predicate sets that subsume at least N queries. Because of the changes we applied to the queries in the predicate augmentation step some of these predicate sets contain redundancies and other are redundant themselves. For example, a predicate set that contains redundant predicates could be *{l_quantity > 20, l_quantity > 30}*. The first predicate *{l_quantity > 20}* is subsumed by the second one *{l_quantity > 30}* so it is clear that this represents and invalid query. Another redundancy case can happen when one of the predicate sets represents a superset of another predicate set. For example, if we consider f_1=*{l_quantity > 20, l_discount > 0.2}* and f_2=*{l_quantity > 20, l_discount > 0.2,* *l_shipmode="AIR"}* as frequent predicate sets, the first one is a subset of the second one. The second one will always subsume more queries than *{l_quantity > 20, l_discount > 0.2}*. For that reason, we have to look at the number of queries that are additionally subsumed by f_2 and treat that value as our support.

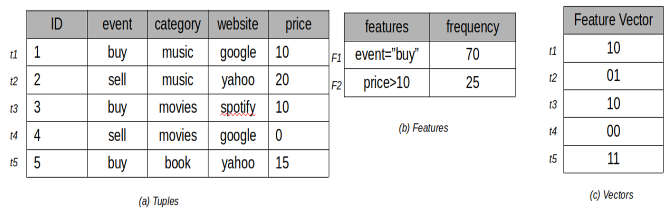
Both types of redundancies mentioned above will be removed after applying the procedure described in below.

In summary, the redundant predicate set elimination step represents the inverse of the predicate augmentation step.

At this point we reach the end of the workload analysis process. Given a set of queries we have extracted representative features. Out of the remaining number of features we choose only the top K ones by sorting them in descending frequency order. K represents an input parameter for the featurization process. In the next steps, the selected set of features will be used for partitioning.

C. Partitioning: Data Augmentation

As observed in several real-world cases, certain queries are executed over and over when the workload is updated with new data. Following featurization, the feature vector is denoted as an n-dimensional bit vector, where n is the number of features extracted. Subsequently, a bit with value "1" indicates that the tuple satisfies a certain feature, and vice-versa, where each bit position (i.e. i-th bit) refers to the feature order within the feature list(Fi). The next Step will be the clustering step where it takes as input this extracted feature bit-vector.



	ID	event	category	website	price
t1	1	buy	music	google	10
t2	2	sell	music	yahoo	20
t3	3	buy	movies	spotify	10
t4	4	sell	movies	google	0
t5	5	buy	book	yahoo	15

(a) Tuples

	features	frequency
F1	event="buy"	70
F2	price>10	25

(b) Features

	Feature Vector
t1	10
t2	01
t3	10
t4	00
t5	11

(c) Vectors

Fig. 3: Data Augmentation Example

Fig. 3 describes a scenario of online shopping with different product categories and merchants as shown in part (a). The table in part (b) contains frequent features extracted based on this table items and common queries. We scan the table using a select statement with the designated features part of the *where* clause. If the tuple satisfies the feature, it is then selected and we assign "1" to a new column called Feature Vector and "0" otherwise. The same applies for all tuples in the table. We repeat this step in a looping scheme over all features included in the feature set. The end result is illustrated in the last part of the figure (c) where each bit of the feature vector corresponds to one of the two feature in the features table.

D. Partitioning: Clustering

After the data augmentation step, a feature bit-vector is computed for each tuple. In the clustering step, we find an optimal partitioning over these feature vectors using the Hierarchical Clustering. Since this algorithm could be really expensive for large datasets, the input size is reduced from the total number of tuples to the number of unique feature vectors and, hence, the input for the partitioning changes from (vector-tuple) to (vector-count) pairs. We consider the count as the weight associated with each feature vector. The feature vectors and their respective count of tuples obtained are clustered to identify the partitioning blocks. The main idea for this partition mapping is to maximize the number of data tuples to be skipped in the dataset during query execution.

Clustering is a kind of technique used in machine learning to group similar objects, where the required number of clusters is defined before grouping these objects. The objects which are in the same cluster are more similar to each other as compared to the objects present in other different clusters. Different types of clustering algorithms exist, the most popular of them are *K- Means clustering, Hierarchical Clustering, Density-Based Spatial Clustering of Applications with Noise (DBSCAN).*

Sun's et. al. [7] approach uses Hierarchical Aglomerative Clustering to group similar feature vectors for efficient query data retrieval. The idea behind using hierarchical clustering for partitioning is that both K-Means and DBSCAN are susceptible to outlier data points and it is difficult to regulate the number of data points in a partition. By making a few modifications (as described in further sections), Hierarchical clustering can be used for effective partitioning in our case.

Hierarchical clustering is an unsupervised clustering algorithm. It involves creating clusters that have a predetermined creation ordering. It can be divided into two types based on a top-down(divisive) or bottom-up(agglomerative) approach. In the agglomerative or bottom-up hierarchical clustering method, each data point is regarded as a single cluster initially. Then, recursively, two most similar clusters are joined based on some similarity (e.g., distance) between each of the clusters, till a single cluster is formed.

The choice of distance measure is a critical step in clustering and it always depends on the type of data which we use. These metrics define the similarity of two elements (x,y) in the dataset. The classical methods on distance measure are Euclidean and Manhattan distances. Other dissimilarity measures are defined as the correlation-based distances which are Pearson Correlation distance, Cosine Correlation distance, etc.; all these types of metrics have a strong influence on the clustering results.

Apart from the similarity metric, there are different types of linkage methods to calculate the similarity between two clusters. Linkage refers to the strategy we use to pick the points to be compared in two clusters. Some methods are:

- Single Linkage - The distance between two clusters is defined as the shortest distance between two points in each cluster.
- Complete Linkage - The distance between two clusters is defined as the longest distance between two points in each cluster.
- Average Linkage - The distance between two clusters is defined as the average distance between each point in one cluster to every point in the other cluster.
- Ward's Method -The distance between two clusters is defined as the sum of the square of the distances between each point in one cluster to every point in the other cluster. Ward's method says that the distance between two clusters, A and B, is the increase in the sum of squares when the clusters are merged.

$$\Delta(A,B) = \sum_{i \in A \cup B} \|\vec{x}_i - \vec{m}_{A\cup B}\|^2 - \sum_{i \in A} \|\vec{x}_i - \vec{m}_A\|^2 - \sum_{i \in B} \|\vec{x}_i - \vec{m}_B\|^2 \quad (1)$$

$$\Delta(A,B) = \frac{n_A n_B}{n_A + n_B} \|\vec{m}_A - \vec{m}_B\|^2 \quad (2)$$

where $\vec{m}_j$ is the center of cluster j, and nj is the number of points in it. Δ is called the merging cost of combining the clusters A and B. For our project, Hierarchical Agglomerative Clustering i.e. bottom-up approach with a modified Ward's Method has been used as the clustering technique.
The basic algorithm for Hierarchical Agglomerative Clustering in our case can be described below in Algorithm 2.

Algorithm 2 Basic Algorithm for HAC

Input: Feature Vectors and their respective counts
Output: An indicator of group membership of feature vectors

```
0: procedure HAC
1: Each feature vector forms a group (cluster)
2: Calculate the similarity matrix between vector pairs
3: repeat
4:     Detect the two closest groups
5:     Merge them to form only one group
6: until All the clusters are merged into one cluster
```

The major modification in this algorithm is regarding the calculation of the similarity function. Since the above-discussed similarity metrics are all based on distance, these metrics are not useful in this scenario of feature vectors. Here, our clusters consist of the feature vectors. We modify Ward's method, as described above to use the improvement in the cost when merging the partitions as the similarity measure.

From the workload analysis, we obtain a m-dimensional feature vector, F = {F_1,F_2,...,F_m}. w_j denotes the weight of the features F_j, or the number of queries subsumed. Let V = {v_1, v_2, . . . , v_n} be the collection of m-dimensional bit vectors, where each vector represents a tuple. If a feature F_j is satisfied by the jth bit of vector v_i, the value is 1 and otherwise, the value is 0. Let P = {P_1,P_2,...,P_k} denote a partitioning over V , and $\overline{v}(P_i)$ be the union vector of all vectors in P_i, i.e., $\overline{v}(P_i) = \bigvee_{v_j \in P_i} v_j$. A partition P_i is pruned by a feature F_j, if none of the vectors in P_i satisfies F_j, i.e., $\overline{v}(P_i)_j$=0, and hence, prunes $|P_i|$ tuples, as each vector represents a tuple. The weight w_j of F_j represents the number of queries subsumed by F_j in the workload. So, the sum of tuples that can be skipped if F_j prunes P_i becomes $w_j \cdot |P_i|$.

The cost function $C(P_i)$ for a partition P_i can be defined as the sum of tuples in P_i that can be skipped when all the queries in the workload are executed [7].

$$C(P_i) = | P_i | \sum_{1 \le j \le m} w_j (1 - v(P_i)_j) \quad (3)$$

The cost function $\mathbb{C}(P)$ over a partitioning can be given by $\mathbb{C}(P) = \sum_{P_i \in P} C(P_i)$ and is, intuitively, the sum of tuples skipped if all the queries are executed in the workload [7].

Based on Ward's method, initially, every feature vector is considered as a partition by itself. At each iteration, two partitions which maximize $\mathbb{C}(P)$ are selected to be merged. The change in $\mathbb{C}(P)$ when merging partitions P_i and P_j is given by [7]

$$\delta(P_i, P_j) = \mathbb{C}(P \cup \{P_i \cup P_j\} - \{P_i, P_j\}) - \mathbb{C}(P) \quad (4)$$

The union vectors of P_i and P_j are ORed when merged together, i.e., $\overline{v}(P_i \cup P_j) = \overline{v}(Pi) \vee \overline{v}(P_j)$ The costs of other partitions are not affected by the merging of P_i and P_j, thus:

$$\delta(P_i, P_j) = C(P_i \cup P_j) - C(P_i) - C(P_j) \quad (5)$$

The modified algorithm is described in Algorithm 3 [7]. The output of this algorithm is a partitioning of the vectors, which is used to construct a **blocking map** returning a block id for a given feature vector. By referring to the partitioning map, a data tuple can be routed to the right block with its corresponding feature vector.

Algorithm 3 Modified Algorithm for HAC

Input: Feature Vectors and their respective counts

```
0: procedure HAC MODIFIED
1: P ← $\{\{v_1\}, \{v_2\}, ..., \{v_n\}\}$, R ← ∅
2: while P is not empty do
3:     merge the pair $P_i, P_j \in P$ with the largest $\delta(P_i, P_j)$
4:     if $| P_i \cup P_j | > minSize$ or $| P_i \cup P_j |$ is the last one in P then
5:         remove $| P_i \cup P_j |$ from P and add it to R
6:     end if
7: end while
8: return R
```

E. Query Execution

Our main goal of query execution is to determine how well the queries can be adapted to the partitioned dataset achieved after the clustering process. After clustering, we have a union or blocking vector corresponding to each feature vector. Based on the union vectors, we perform the partition on the table. All the queries for execution are already transformed into feature vectors in the parsing step. However, during the execution of the queries for data retrieval, the query feature vector is modified by inverting the feature vector bits. For example, out of a feature set F={f1,f2,f3}, if a query is subsumed by f2 and f3 present in the filter condition, after parsing, the query feature vector will be [0,1,1] but during execution, it is inverted to [1,0,0]. Assuming, the clustering resulted in three partitions with blocking vectors as {[1,1,0],[0,1,1]and [1,0,0]}, we perform a bit-wise OR operation between each of the union vectors with the inverted query vector. If the OR operation yields [1,1,1] we will scan only that block, the second block in this example, and skip all the other blocks.

V. Implementation

In this section, we describe the TPC-H dataset and its corresponding benchmark used for the implementation. Then, we will mention briefly implementation details of our workload analysis with its substeps: Parsing, Featurization and Predicate augmentation and at the end, we will clustering details. To close the section we describe relevant software and experimental setup aspects.

A. Dataset and Benchmark

TPC-H Dataset: The TPC-H Benchmark is a transaction processing and database benchmark which is specific to decision support. The decision support is basically used to examine and analyze large volumes of data, execute queries that have high complexity and provide answers to several critical questions.

TPC-H provides two basic tools which we have also used for our project:

DBGEN: DBGEN is a Tool which helps in generating dataset used in the TPC-H benchmark. It is used to create a set of tables, which can be done at different scale factors or storage size. For our project, we focus on the Line Item table, as it is the largest in the dataset. We use a scale factor of 1.

QGEN: It is an executable query text generation program used in the TPC-H benchmark. TPC-H also provides some basic queries and out of which only a few are targeting the line-item table and in order to generate new queries, QGEN uses those template queries provided by TPC-H. We used the query generator which gave us very simple set of predicates.

Using the tools that are provided by the TPC-H Benchmark we generated tables and queries.

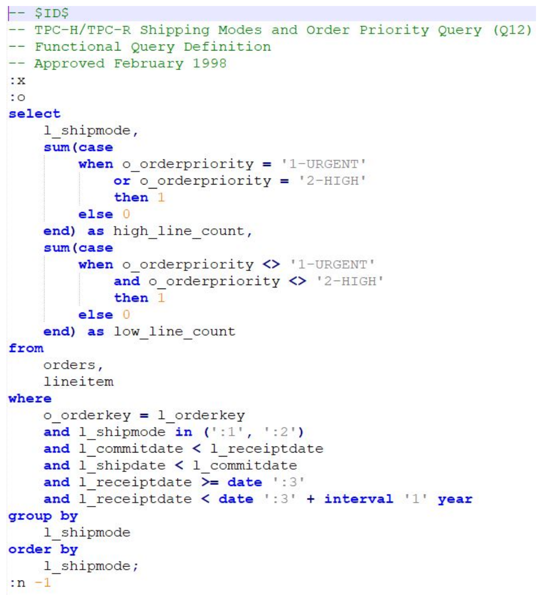
For our task, we have considered only the Line-item table and there were a total of 22 template SQL queries that are provided by the TPC-H benchmark and we generated some set of queries out of those 22 queries using QGEN.

Custom QGEN: In the featurization step, we found out that the generated queries from QGEN use only a small set of the available 16 columns of the Line-Item table while varying the values only from this small set of columns. This caused the feature space to be very small meaning that we would end up with a small variety of feature vectors in the end which would make the clustering impractical.

Due to this issue, we decided to generate other queries which are based on the column metadata from the "line-item" table. For this, we used a procedure that takes the list of column names and metadata about each column and generates queries by randomly selecting predicates for the "where" clause and the metadata is used for assigning the values for each predicate.

The metadata collected for the columns is dependent on the nature of the column itself. For the sake of this project, we only needed 2 groups, but this can be extended to include more if needed. The first group included the column name, it's data type and the minimum and maximum values for this column. The second group included the column name, it's data type and a list of the unique values the column holds. For numerical columns of datatypes "int" or "decimal" and "date" columns, we used the first group. While for categorical columns of "string" type is used in the second group.

At run time, the script with the help of the metadata can choose either a random number between the minimum or maximum, if the column is from group one, or one of the values from the values list, if the column is from group two.



```
-- $ID$
-- TPC-H/TPC-R Shipping Modes and Order Priority Query (Q12)
-- Functional Query Definition
-- Approved February 1998
:x
:o
select
    l_shipmode,
    sum(case
        when o_orderpriority = '1-URGENT'
            or o_orderpriority = '2-HIGH'
            then 1
        else 0
    end) as high_line_count,
    sum(case
        when o_orderpriority <> '1-URGENT'
            and o_orderpriority <> '2-HIGH'
            then 1
        else 0
    end) as low_line_count
from
    orders,
    lineitem
where
    o_orderkey = l_orderkey
    and l_shipmode in (':1', ':2')
    and l_commitdate < l_receiptdate
    and l_shipdate < l_commitdate
    and l_receiptdate >= date ':3'
    and l_receiptdate < date ':3' + interval '1' year
group by
    l_shipmode
order by
    l_shipmode;
:n -1
```

Fig. 4: A simple query generated using qgen

We excluded 3 columns from our list "l_discount", "l_tax" and "l_comment". Columns "l_discount" and "l_tax" were decimal columns that included a very narrow range, while "l_comment" is an open text column that is used for operational purposes.

B. Workload Analysis:

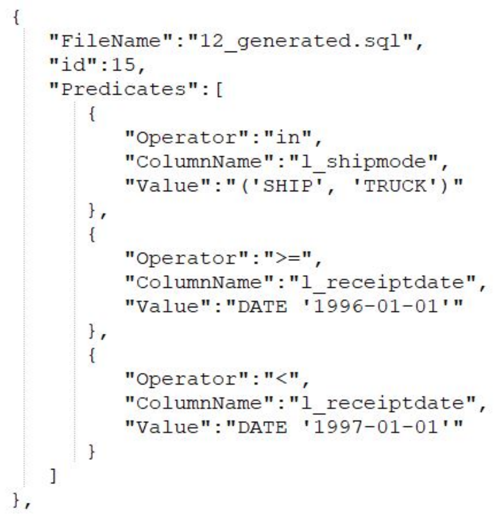
1) Parsing

This is the first step of workload analysis where we are trying to extract only the important information out of the queries, for instance, Predicates. Here, we are only dealing with WHERE clause which contains all the predicates and these predicates are quite important to determine how important and useful a query is. Using the WHERE clause, we can filter and fetch all the necessary records which are required for the implementation. As input, we used SQL "select" scripts and then to parse a SQL query for further processing we have taken some constraints into consideration:

We have removed all the predicates that correspond to another table and discarded the Predicates that contain Joins. Secondly, we have only considered filter predicates that contain operators, for instance, *in, Range*$(>, <, >=, <=)$ *and equal(=)*.

All the predicates which are from Line-item table and contains Range, in, and equal operators are extracted. After this, we store them in a JSON structure to help for further processing.

An example of a query is presented below to show how predicates are stored in a JSON structure.



```
{
   "FileName":"12_generated.sql",
   "id":15,
   "Predicates":[
      {
         "Operator":"in",
         "ColumnName":"l_shipmode",
         "Value":"('SHIP', 'TRUCK')"
      },
      {
         "Operator":">=",
         "ColumnName":"l_receiptdate",
         "Value":"DATE '1996-01-01'"
      },
      {
         "Operator":"<",
         "ColumnName":"l_receiptdate",
         "Value":"DATE '1997-01-01'"
      }
   ]
},
```

Fig. 5: JSON structure for storing predicates

The structure contains the filename and the id so that we can keep track of the query we are using. All the predicates are stored as simple predicates with “operator”, “columnName” and the “value”. Predicates can have AND, OR and sometimes nested OR queries. Finally, all the predicates are stored as a JSON object inside a big JSON array.

The main reason why we chose JSON is that it is a way of storing information in an organized way so that one can easily interpret what the information is all about. It has a straightforward structure that is easily readable. JSON is a lightweight and easy to use structure when compared to other data interchange formats. It is also widely used for its more compact style.

2) Featurization

Given the JSON format mentioned in the previous section which represents the parsed workload, we now move to the featurization step. The goal of this step is to select and combine the most representative predicates that are present in the workload. During our implementation, we made some minor changes to the algorithms described in introduce design section. One modification was to the Apriori mining algorithm. For the workload we used, we experienced a high number of combinations of predicates that we were not able to prune using the apriori property. To overcome this issue, we modified the classic algorithm by introducing a stop early-constraint. We concluded that in this specific case the apriori-algorithm can be stopped after a few iterations, more specifically when we reach a certain length for the sets that are generated. This can be done because those sets will be always removed in the next step, the redundant predicate removal step. At the end of the featurization process, we save all the representative features and their frequency in a JSON file which will in turn used for the data augmentation step.

3) Data Augmentation

For this step, we take as input the JSON file with the features from the previous step and parse it. Next, we create a new column (of type string) to hold the feature vector and we loop through the JSON file to check whether tuples satisfy each feature or not. For one feature, we create a subset with all tuples that satisfy the feature and we append "1" to the corresponding tuple string in the features column. Similarly, we create a subset for all tuples that do not meet the feature, in other words, the remaining tuples and append "0" to their feature vector. Lastly, we employ the union operator to combine both datasets into one and use this combined set for the next iteration.

4) Partitioning, Clustering

The output of the data augmentation step i.e. the feature vector and their tuple count is used as input for the clustering and partitioning. Algorithm 3 as described in the section IV-D is used as-it-is without any further modifications. The (vector-count) pairs are stored as *CurrentPartitions*, which is initially a set of all feature vectors and their counts and recursively gets updated with the merged partitions. Two data structures are used to store the [vector: union vector] mapping globally and for the current partitions. The global [vector: union vector]

Algorithm 4 Data Augmentation

```
1: for t in T do
2:     CREATE s(t)=" "
3:     for f in F do
4:         if f(t) then
5:             $s(t) += 1$
6:         else
7:             return strings
8:         end if
9:     end for
10: end for=0
```

mapping is initialized as [vector: vector] and is updated after each iteration to [vector: union vector] pairs, where union vector is the OR-ed vector of all the feature vectors in the partition. After each iteration, the current partition mapping also gets updated with the [unionvector: unionvector] of the merged partitions, where the key is now regarded as the feature vector to be merged. The merged partitions only remain in the current partition set, if its size is less than or equal to a predefined max partition size, otherwise, it is removed and regarded as a final cluster. The algorithm returns a mapping of all initial feature vectors with their updated union vectors after clustering.

This can be illustrated with the help of an example. Let's assume we have (vector-count) pairs as {[(0,1,0),12], [(0,0,1),11], [(0,0,0),19], [(1,1,0),10]}. The user-defined weights or importance give to the features are assumed to be [50, 20, 10]. The global mapping at this point is {[(0,1,0):(0,1,0)], [(0,0,1):(0,0,1)], [(0,0,0):(0,0,0)], [(1,1,0):(1,1,0)} and the current partitions is {[(0,1,0):(0,1,0),12], [(0,0,1):(0,0,1),11], [(0,0,0):(0,0,0),19], [(1,1,0):(1,1,0),10}. Table I shows the improvement in cost matrix for the vector pairs after the first iteration.

Vectors	V1	V2	V3	V4
V1	0	-340	-380	-600
V2	-340	0	**-190**	-870
V3	-380	**-190**	0	-1330
V4	-600	-870	-1330	0

TABLE I: Improvement in cost matrix

The Max Cost Change is -190 for the Vectors V2 and V3, so these vectors are merged. The mapping now gets updated to {[(0,1,0):(0,1,0)], [(0,0,1):(0,0,1)], [(0,0,0):(0,0,1)], [(1,1,0):(1,1,0)} and the current partitions gets updated to {[(0,1,0):(0,1,0),12], [(0,0,1):(0,0,1),30], [(1,1,0):(1,1,0),10}.

C. *Query Execution*

How we are adapting the queries on our clustered results is discussed in the design section. To achieve this in our implementation we have added two new columns in the line-item table i.e. Union vector and corresponding block-ids. We have partitioned the line-item table based on the union vectors. We perform the same steps as mentioned in the design section, inverse the feature vector for the query but instead of performing bit-wise operation on all the union vectors we pass some other filter in the where condition of the query i.e. If a query vector is [1,0,0,1] the necessary union vectors which are required to perform the bit-wise operation so that it yields [1,1,1,1] could be in the partition set ([0,1,1,1],[0,1,1,0],[1,1,1,0]) which implies less computation. Where condition of the query would now consist of another filter i.e.partition in([0,1,1,1],[0,1,1,0],[1,1,1,0]).Hence the query vector would scan only on this set to achieve [1,1,1,1] and use those union vectors to scan the database.

D. *Software Details and Experimental Setup*

In the following subsection, we give a brief overview of the software architecture and mention what tools and libraries were used. We implemented the prototype using Java 1.8 and deployed it as a web REST server. All experiments have been conducted in our Spark YARN cluster with Spark version 2.4.4. We use HDP to deploy our Spark YARN cluster, which is an open-source Apache Hadoop distribution based on a centralized architecture (YARN). The cluster has ten Spark clients, which includes one node to access the cluster, two nodes as HDFS NameNodes (one active and one standby) and seven executor nodes as HDFS DataNodes, which can also be used to run Spark applications. Each of them runs on virtual machines, with VMWare ESXi as hyper-visor. Each executor node has four CPU cores, 16 GB RAM and 150 GB hard disk space.

In order to retrieve data and partition the used tables, we used Spark jobs that were submitted through an Apache Livy web-server, version 0.6.0. We include in our implementation a web interface that communicates with the Java web server and can be used for any step of the process. The communication is done through HTTP requests which are then propagated to the Apache Livy server.

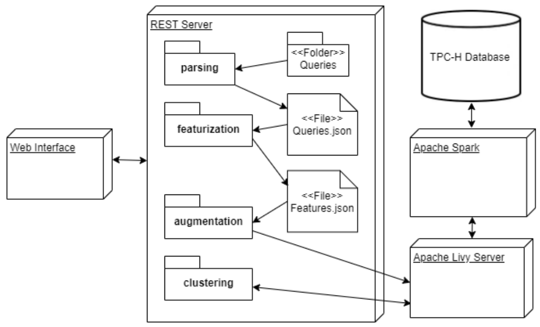


Fig. 6: Deployment diagram of our prototype implementation

VI. Evaluation

A. Baselines and Evaluation metrics

In this section, we elaborate on the baseline models and the metrics that we use for our evaluation. We are comparing our approach with two baseline models. The first baseline that we use is a simple, non-partitioned approach which we call *non-partitioned*. As a second baseline we use a partitioned table obtained using the hierarchical clustering approach. When we evaluate on this table we use the initial queries. We name this baseline model *initial-queries*. Finally, both baselines are compared with our approach, where we perform all the steps, including the query augmentation. The main difference between this approach and the *initial-queries* baseline is that the queries include an "in" condition with a list of all potential block vectors. These vectors tell us which partitions we have to scan.

Below are the metrics used in our evaluation. The first metric we use is the average execution time of queries. As a second metric, since we are using the data skipping approach, we were also interested in knowing the absolute number of tuples being scanned. To achieve this we calculate the total number of records actually being accessed for each query, and use the average as a metric.

B. Research Questions

1) Can we reproduce the performance improvements of Sun et al. using hierarchical agglomerative clustering for aggressive data skipping, compared to random hash partitioning?

Though this is a single research question, to be evaluated experimentally, it demanded from us to find solutions to many problems not clearly explained in the work of Sun et al.

C. Experiments

For our experiments, we picked out a total of 330 queries. After applying the frequent itemset mining algorithm, with a minimum support of 25, we were able to extract a total of 15 high-support features. While taking into consideration the constraints on partition size, we set the maximum limit on the number of records per partition to 100K. Partitions that will include more than 100k records would have to be divided into two smaller ones. In total our proposed model divided the data into 46 separate partitions.

For simplicity we used the smallest TPC-H version. All the experiments were conducted on the lineitem table, which contained approximately 6 million records.
Figure 7 shows that the non-partitioned baseline has the best average query execution time. If we look at the comparison between the initial-queries baseline and our approach, *augmented-queries*, we can conclude that modifying the queries has a positive impact on the average query execution time. For this comparison, we have exactly the same setting, the only different aspect is the modified queries. This shows that the aggressive data skipping approach has generally a positive impact in a partitioned setting.

We believe that the non-partitioned baseline achieved the best result for this metric because of a poorly selected partitioning scheme. This may be because of an unsuitable, artificial workload which fails to meet the necessary assumptions that the SOP approach makes. Because of this, the features that we generate and use do not lead to a good partitioning scheme.

Figure 8 shows that for the second evaluation metric, the augmented-queries model has the best results, scanning on average around 4 million records. This means that on average, one third of the records are skipped. The two baselines, *non-partitioned* and *initial-queries* do not perform any skipping.

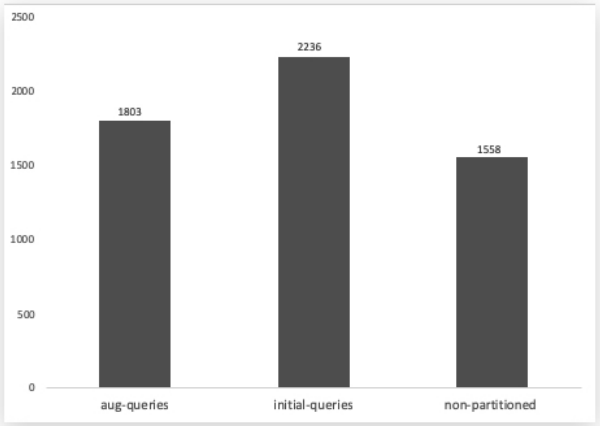


Fig. 7: Metric 1: Average execution time of queries

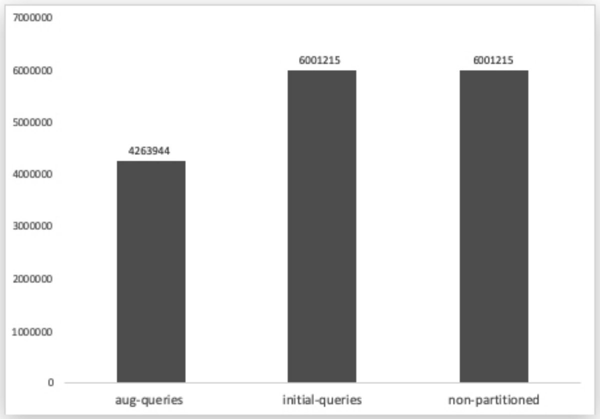


Fig. 8: Metric 2: Number of tuples scanned

VII. Conclusion and Future Work

As data volumes continue to grow, it becomes crucial to develop new strategies to improve the efficiency of large-scale query processing. With the concept of Big Data being on the rise, the existing traditional techniques in data management may not be suitable. In this paper, we presented the concepts replicating the work of Sun et. al. on skipping-oriented partitioning (SOP), a fine-grained data partitioning framework to maximize the number of data skipped during query execution. The Sun et. al. paper had a private implementation. We provide an open-source implementation of our work. We describe the primary steps involved in implementation i.e, (1) *Parsing* to find the most relevant information from the workload, (2) *Featurization* to find the most representative predicates, (3) *Data Augmentation* to generate final feature vectors representative of the workload and (4) *Clustering*, which uses a hierarchical agglomerative approach to cluster these feature vectors and generate the partition clusters. We implement our framework with a modern version of SPARK, instead of SHARK (used by Sun et.al), which has a better interface with SQL and HDFS, being thus potentially better for horizontal partitioning. As a result, we provide a more up-to-date solution.

We evaluated the effectiveness of our framework using the Line-Item table of the TPC-H workload. Our framework is evaluated on the basis of average execution time and the number of data tuples scanned against two baselines of (1) *a non-partitioned* table and (2) *a partitioned table using normal queries.* Although, the non-partitioned table proved best in terms of the average query execution time, our framework works relatively better in a partitioned setting and shows a prominent decrease of approximately 33% in the number of data tuples scanned.

We also faced a few limitations during the implementation. The parsing step is difficult for nested queries and thus such queries have been avoided for simplification. The featurization step is greatly dependent on the available workload to generate a sufficient number of features for clustering and is, hence, not very robust. Another challenge that we faced during the implementation process has been the query execution step. For the aggressive data skipping approach, each query needs to be modified in order to skip data and this cannot be trivially implemented in a data management system.

This paper is open to future improvements and refinements such as a better adaptation of query workload and evaluating the effectiveness of this framework on even larger datasets. Furthermore, important areas of work are to extend our implementation to include replication and hybrid partitioning, as well as efforts in documenting well our code such that our solution could be used by anyone interested in efficient data partitioning for Spark SQL-based workloads.

Our work could also be used as a strong baseline for novel partitioning approaches, such as proposals using deep reinforcement learning [3], [4].

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, page 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[2] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: A workload-driven approach to database replication and partitioning. *Proceedings of the VLDB Endowment*, 3(1):48–57, 2010.

[3] Gabriel Campero Durand, Marcus Pinnecke, Rufat Piriyev, Mahmoud Mohsen, David Broneske, Gunter Saake, Maya S Sekeran, Fabián Rodriguez, and Laxmi Balami. Gridformation: towards self-driven online data partitioning using reinforcement learning. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–7, 2018.

[4] Milena Malysheva and Ivan Prymak. Evaluation of Unsupervised and Reinforcement Learning approaches for Horizontal Fragmentation. 2019.

[5] M. Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems, third edition.* 2011.

[6] Andrew Pavlo, Carlo Curino, and Stanley Zdonik. Skew-aware automatic database partitioning in shared-nothing, parallel OLTP systems. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2012.

[7] Liwen Sun. *Skipping-oriented Data Design for Large-Scale Analytics.* PhD thesis, EECS Department, University of California, Berkeley, Dec 2017.

[8] Liwen Sun, Sanjay Krishnan, Reynold S. Xin, and Michael J. Franklin. A partitioning framework for aggressive data skipping. *Proceedings of the VLDB Endowment*, 7(13):1617–1620, 2014.

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