Fine-grained Partitioning for Aggressive Data Skipping

Liwen Sun, Michael J. Franklin, Sanjay Krishnan, Reynold S. Xin†
AMPLab, UC Berkeley and †Databricks Inc.
{liwen, franklin, sanjay, rxin}@cs.berkeley.edu

ABSTRACT

Modern query engines are increasingly being required to process enormous datasets in near real-time. While much can be done to speed up the data access, a promising technique is to reduce the need to access data through *data skipping*. By maintaining some metadata for each block of tuples, a query may skip a data block if the metadata indicates that the block does not contain relevant data. The effectiveness of data skipping, however, depends on how well the blocking scheme matches the query filters.

In this paper, we propose a fine-grained blocking technique that reorganizes the data tuples into blocks with a goal of enabling queries to skip blocks aggressively. We first extract representative filters in a workload as *features* using frequent itemset mining. Based on these features, each data tuple can be represented as a feature vector. We then formulate the blocking problem as a optimization problem on the feature vectors, called *Balanced MaxSkip Partitioning*, which we prove is NP-hard. To find an approximate solution efficiently, we adopt the bottom-up clustering framework. We prototyped our blocking techniques on Shark, an open-source data warehouse system. Our experiments on TPC-H and a real-world workload show that our blocking technique leads to 2-5x improvement in query response time over traditional range-based blocking techniques.

Categories and Subject Descriptors

H.2.2 [**Database Management**]: Physical Design

Keywords

Data warehouse; Partitioning; Query processing; Algorithms

1. INTRODUCTION

Data analytics has been proven critical in many applications, ranging from business decision making to scientific discovery. Many of these applications require *interactively* unlocking insights from enormous data. To meet this need, designs of modern query engines (e.g., [8, 37, 32, 2, 35]) are striving to identify opportunities to shorten query response time. One dimension of this effort focuses on improving the data scan *throughput*, such as memory caching, parallelization, and data compression. Another dimension is to *reduce* the data access. For example, columnar-oriented data layout prevents the queries from accessing irrelevant columns, and sampling provides approximate answers by scanning only a small subset of data. Along these lines, there has recently been an increasing interest in reducing data access through *data skipping* [1, 6, 35, 37]. Intelligently skipping *blocks* of tuples can significantly speed up the query processing.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2610515.

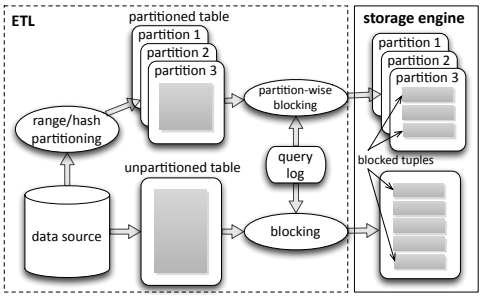


Figure 1: Blocking at Data Loading

1.1 Background

Traditionally, data skipping has long been implemented via partition pruning. In a data warehouse environment, many tables are partitioned by time and most queries have a time range filter. A query can check the time ranges of the partitions and decide which partitions to scan and which to skip. While this is an effective way to prune data, the remaining partitions can still contain a lot of tuples.

Recent systems [28, 1, 34, 7, 6, 37, 35] support skipping data *blocks*. A block in these systems is a horizontal partition that is fairly small (e.g., 1000's or 10,000's of tuples). Each block is associated with some metadata such as min and max values. Before scanning a block, data skipping first evaluates the query filter against this metadata and then decides if the block can be skipped, i.e., the block does not need to be accessed. The salient features of data skipping include avoiding random disk access and incurring minimal storage and maintenance overhead [28, 34, 6, 35]. Block skipping speeds up table scans by accessing less data, which is beneficial to both disk- and memory-resident tables.

1.2 Goals

The effectiveness of block skipping depends on how the data tuples are partitioned into blocks. We refer to this

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

(a) tuples

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

(b) features

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

(c) vectors

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

(d) blocks

Figure 2: Example of Blocking

process as *blocking*. Current systems adopt very small block sizes for data skipping. For example, IBM DB2 BLU [35] uses 1,000-tuple blocks; Google's PowerDrill [6] suggests 50,000 tuples; Shark [37] skips data on HDFS blocks, each of which is 128MB by default. These systems rely on range partitioning to generate such blocks. While range partitioning has been useful for many purposes, it may not be ideal for generating fine-grained blocks for skipping. Specifically, range partitioning lacks of a principled way of: (1) setting the fine-grained ranges on each column that matches the data skew and workload skew, (2) allocating the number of partitions for different columns and (3) capturing inter-column data correlation and filter correlation.

In this paper, we propose a workload-driven blocking technique, with a goal of (horizontally) partitioning the data into *fine-grained, balance-sized* blocks in a way that queries can maximally skip the blocks. This is an offline process that executes at data loading time and may be re-executed later to account for a more recent workload. Figure 1 depicts the use of blocking in an ETL process. Note that our blocking techniques can co-exist with traditional horizontal partitioning techniques, as these techniques may be used for a different purpose, such as load balancing and roll-in/roll-out operations. Specifically, our techniques can be applied to further segment each individual partition. As shown in Figure 1, we take as input the data tuples and a query log, and write the blocked tuples to the storage engine; if the incoming tuples are partitioned, we can block each partition individually (in parallel). A newly-inserted partition can be blocked on its own and will not affect existing partitions.

First, we extract some filter predicates as *features* from a past query workload using frequent itemset mining. We then generate feature vectors by precomputing these filter predicates on the data and solve an optimization problem to guide the data blocking. In many real-world workloads, especially the reporting and scheduled workloads, similar queries are repeatedly run on different time-ranged data. We also analyze real-world workloads in Section 2, which show that (1) a small set of representative filters are commonly used by many queries and (2) many queries use recurring filters. These findings suggest that our workload-driven approach can be effective for real query workloads.

Some previous work also utilizes workloads for physical database design, e.g., [17, 21, 12, 10]. Specifically, our work is related to materialized view selection (MVS) [16, 10]. Like MVS, we exploit precomputation. However, our partitioning techniques are at the physical-record level and are complementary to materialized views. In fact, our techniques can be applied to partition large materialized views, e.g., data cubes. As we will show shortly, we maintain concise feature-based metadata derived from precomputation. Another proposed data skipping technique involves the use of small materialized aggregates (SMAs) associated with partitions [28, 34]. These SMAs have been shown to improve query performance in range-partitioned systems. In contrast, our work is focused on constructing fine-grained partitions that more closely capture the access patterns of complex analytics workloads. Like materialized views, SMAs are also complementary to our approach and in fact could be implemented on our partitions as well. We defer the detailed discussion of related work to Section 8.

1.3 Example

Suppose we are given a table as shown in Figure 2(a), an example log of online events. We first look at the log of queries that were posed on this table and extract a set of features, each of which is a representative filter with possibly multiple conjunctive predicates. Suppose the features extracted are as shown in Figure 2(b). Given these features, we then transform the data tuples into feature vectors. This process can be done by scanning the table once and batch-evaluating the features on each tuple. As shown in Figure 2(c), each feature vector is (in this case) a 3-dimensional bit vector, whose i-th bit indicates whether this tuple satisfies filter F_i. In practice, the number of features can be kept small, e.g., < 50. We then partition the tuples according to these vectors. Intuitively, tuples that do not satisfy the same features should be placed in the same block such that, when a query uses one of these features as filter, this block of tuples can be skipped altogether. An example of the resulting blocked tuples is shown in Figure 2(d). For each block, we compute a *union vector* by taking a bitwise `OR` of all the feature vectors in it. If the i-th bit of the union vector is 0, then we know that no tuple in this block satisfies feature i. In this case, any query whose filter is F_i can skip this block. For example, a query on F_3 can skip the blocks P_1 and P_3. More generally, a query can skip blocks if its filter *is subsumed by* (i.e., is stricter than) some features. For example, a query with filter *event = 'buy'* $\wedge$ *product = 'jeans'* is subsumed by both features F_1 and F_2, which lead to the skipping of P_2 and P_3 respectively.

1.4 Contributions

To realize this design, we address a few technical challenges as outlined below.

Feature Selection. Indeed, selecting the right features to guide partitioning is critical. We develop a *workload analyzer* to identify representative filters as features from a query log. We consider a feature representative if it could be used to help many queries. If some filter predicates are fre-

quently used together, we should combine these predicates as one feature to skip data more effectively, e.g., feature F_3 in Figure 2. To capture both *frequency* and *co-occurrence* of filters, we model the feature selection as a frequent itemset mining problem. Due to their subsumption relations, some features can be redundant. For example, $revenue > 0$ could be redundant if there is already a feature $revenue > 100$. We develop a principled way to eliminate redundancy.

Optimal Partitioning. Given a set of features, we compute a feature bit-vector for each tuple. The problem then is to find an optimal partitioning over these vectors. This is clearly a hard problem, as different features may be conflicting, may be correlated, and may have different selectivities. We formulate the *Balanced MaxSkip* partitioning problem: given a desired number of tuples per block, find a partitioning over a collection of tuples (represented as bit vectors) that maximizes the number of tuples that can be skipped. This objective is fundamentally different from other well-known partitioning objectives, such as k-means and distance-based clustering [29]. We prove that *Balanced MaxSkip* is NP-hard, by a reduction from the hypergraph bisection problem [24]. We conjecture that *k-MaxSkip*, a variant without the balance constraint, is also NP-hard. To find an approximate solution efficiently, we adopt the classic bottom-up clustering framework, as it naturally incorporates our objective function and is a widely-understood framework with scalable implementations, e.g., [38, 5].

Scalability. It is prohibitively expensive to run a clustering algorithm on large datasets. Fortunately, we observe that the input size can be reduced from the number of tuples to the number of distinct feature vectors. The latter mostly depends on the number of features and can be small (e.g., $<10k$) in practice. As shown in Section 7, although we run a sophisticated partitioning algorithm on the vectors, the cost bottleneck of the entire blocking process is still the actual data movement instead of the partitioning algorithm.

We prototype our blocking techniques on Shark [37], an open-source data warehouse system. We conduct experiments on TPC-H benchmark and a real-world ad-hoc workload from a video streaming company. The results show that our techniques reduce the data access by a factor of 4-7 over existing skipping techniques on top of range partitioning. We also demonstrate that this reduction can directly translate to a reduction in query response time, on both disk and memory resident data. Specifically, we improve the query response time by 5-14x over full table scans without skipping and by 2-5x over existing skipping techniques.

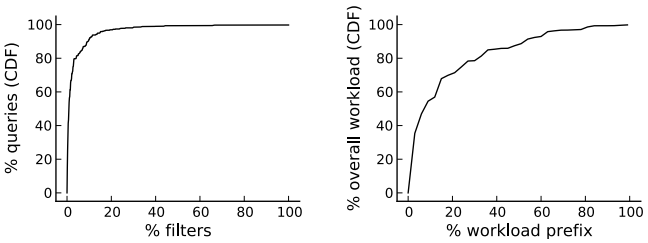
The remainder of paper is organized as follows. Section 2 gives an overview of our blocking approach. Section 3 presents the workload analyzer. We discuss the partitioner in Section 4. We show how skipping works during query execution in Section 5. Section 6 discusses the practical issues. We report our experimental results in Section 7. Section 8 reviews the related work and Section 9 concludes the paper.

2. OVERVIEW

In this section, we first discuss the assumptions for our techniques and then give an overview of the workflow.

2.1 Workload Assumptions

We extract representative filters as features from the workload to guide the blocking. In order for our approach to work well, we expect two properties of the workload:



(a) Filter Commonality (b) Filter Stability

Figure 3: Filter Analysis on Real Ad-hoc Workloads

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

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

Obviously, the commonality and stability of filters can be observed in recurring scheduled or reporting queries. Such queries are usually generated from templates and the same set of filters would be repeatedly used on the data of different time ranges. The effectiveness of using past queries to guide database design was also evidenced in many previous works, e.g., [10, 21, 17, 12].

We conducted empirical analysis on a real-world production SQL workload from a video streaming company called Conviva. These 8664 ad-hoc queries, spanning the period from 06/2010 to 01/2012, were used for problem diagnosis and data analytics over an access log of video streams. Note that each query uses possibly multiple filter predicates. For each predicate, e.g., *event = 'click'*, we count its frequency, i.e., how many queries use it. In Figure 3(a), we sort the filters by descending frequency and plot the cumulative percentage of the queries that use the filters. A point (x, y) indicates that the most frequent x% filters are used by y% of queries. We can see that the filter usage is highly skewed, e.g., 10% of the unique filters were used by 90% of the queries. This implies that using only 10% of filters as features can benefit 90% of the queries.

We then examine the queries in the order as they arrive. To prevent our analysis from being biased towards a particular starting point, we divided the 8664 queries into 5 disjoint time windows and plotted five curves. Figure 3(b) shows an average over these five curves. A point (x, y) means that, as we have seen x% of the queries (or x% workload prefix), the filters in these x% are used by y% of all the queries in the workload. If every query used completely new filters, i.e., there is no recurring filter, this curve would be a function of $y = x$. The plot, however, shows that many queries use recurring filters. In particular, 80% of the entire workload uses the filters that already occur in the 30% prefix. Since the filters are recurring, we can infer that most of the future filters are predictable based on a past workload.

2.2 The Blocking Workflow

Having described our workload assumptions, we now overview the blocking overflow, as depicted in Figure 4. This workflow consists of three standard data marshaling steps (shaded arrows) and two important modules named the workload analyzer and the partitioner.

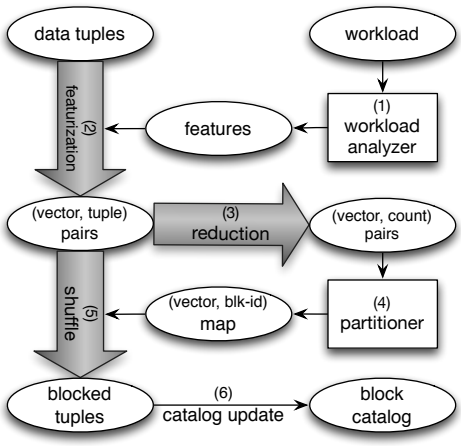


Figure 4: The Blocking Workflow

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

(1) Workload Analyzer. The workload analyzer (Section 3) extracts a set of features from the query workload. A feature is a representative filter with possibly multiple predicates.

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

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

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

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

(6) Catalog Update. We add the features and one union vector (e.g., Figure 2(d)) for each block to the *block catalog* (Section 5). After this step, we can drop the feature vectors, and the blocked tuples are ready to serve queries.

The above workflow can be executed as an offline process at data loading time and may be re-executed later to account for workload changes. In the event that the data arrival rate is high or that the new data needs to be queried immediately, the blocking process can be postponed. As stated in Section 1, in many data warehouse applications, tables are partitioned by time ranges, and new tuples are typically batch-inserted as a new partition. We can consider our blocking as a "secondary" partitioning scheme under each such coarse-grained partition. For example, when a new partition `dt='1994-11-03'` is added to the table `logs`:

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

we can invoke our blocking process on this newly inserted partition without affecting existing data. To re-block a horizontally partitioned table, we can block each partition individually in parallel.

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

3. WORKLOAD ANALYSIS

The goal of workload analysis is to extract features from the query trace. We follow two intuitions. First, we should use representative filters as features to guide the blocking. A feature is representative if it can potentially help a large number of queries skip data. Second, the filter predicates that are frequently used together should be considered as one single feature, e.g., F_3 in Figure 2, as these predicates will likely be used together in the future queries and combining them can greatly maximize block skipping. To take into account the counting and co-occurrence of predicates, we model the workload analysis as a *frequent itemset mining* problem [22]. In this section, we first formulate the problem and then present a principled approach for feature selection.

3.1 Workload Model

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

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

The workload is thus modeled as a transactional database [9], where a predicate is an item and a query is a transaction. For example, Q_1 can be viewed as a transaction of only one item *product='shoes'*. Let F be the set of all predicates that occurred in $\mathcal{Q}$. We call any set of conjunctive predicates $F_i \subseteq F$ a *predicate set* (analogous to *itemset*). By definition, each Q_i is a predicate set.

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

3.2 Predicate Augmentation

We want to select representative predicate sets as features. As discussed in Section 1 and Section 5, a feature can be used to skip data blocks, but only for the queries it subsumes. Let us show this using Example 1. Suppose we use {*product* in ('*shoes*', '*shirts*')} as a feature. We can see that this feature subsumes both Q_1 and Q_2. If there is a block that does not contain any tuple that satisfies {*product* in ('*shoes*', '*shirts*')} (indicated by the union vector, as discussed), we can then infer that no tuple in this block satisfies Q_1 or Q_2, because these queries are subsumed by (i.e., even stricter

than) this feature. In this case, Q_1 and Q_2 can safely skip this block.

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

---

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

After this augmentation step, the number of occurrences of a predicate set equals to the number of queries it subsumes. For example, the workload in Example 1 becomes:

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

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

3.3 Redundant Predicate Set Elimination

We now have obtained a set of predicate sets that subsume at least T queries. Unfortunately, some of these predicate sets may be redundant. For example, if there is a predicate set {*publisher*='*google*', *revenue*<0} in the result, both its subsets {*revenue*<0} and {*publisher*= '*google*'} would also be present, due to the apriori property [9]; but the predicate set {*publisher*='*google*', *revenue*<0} is only useful for the queries that cannot be subsumed by either of the subsets. In addition, the filter augmentation step may also introduce redundant results. For example, if {*revenue*> 32} is present, then {*revenue*>21} is also present due to the augmentation. Again, {*revenue*>21} is only helpful for the queries that cannot be subsumed by {*revenue*>32}. Existing approaches that keep only *maximal* or *closed* frequent itemsets [22] do not capture the subsumption relations in our specific problem.

We develop a principled way to remove redundancy from the frequent predicate sets. Notice that the frequency threshold T used in Section 3.2 indicates that we are only interested in the predicate sets that subsume at least T queries. We will use the same principle for redundancy removal. Specifically, let $\mathcal{F}$ be the frequent predicate sets, we consider a predicate set F_i in $\mathcal{F}$ to be redundant and remove it, if the number of queries F_i *additionally* subsumes, given the predicate sets that are already in $\mathcal{F}$, is less than the threshold T. We denote by $subsume(F_i, \mathcal{Q}) \subseteq \mathcal{Q}$ the set of queries in $\mathcal{Q}$ that are subsumed by F_i. We use the following procedure to eliminate the redundant predicate sets:

---

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

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

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

4. THE PARTITIONING PROBLEM

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

4.1 Problem Definition

Suppose we have a set of m features obtained from the workload analysis, denoted by $\mathcal{F} = \{F_1, F_2, \ldots, F_m\}$. We denote by w_j the weight of the features F_j, i.e., the number of queries it subsumes. Based on these features, the data tuples are augmented with binary vectors (Step (1) in Figure 4). We now formulate the partitioning problem on the m-dimensional bit vectors.

Let $V = \{v_1, v_2, \ldots, v_n\}$ be a collection of m-dimensional bit vectors, where each vector corresponds to a tuple. We will focus on our discussion on the vectors, finding as a partitioning on the vectors is equivalent to finding a blocking on the tuples. The *j-th* bit of v_i, denoted by v_{ij}, indicates whether v_i satisfies feature F_j. Let $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be a *partitioning* over V, i.e., $\mathcal{P}$ is a set of disjoint subsets of V whose union is V. Let $\overline{v}(P_i)$ be the union vector of all vectors in P_i, i.e., $\overline{v}(P_i) = \bigvee_{v_j \in P_i} v_j$. We say a feature F_j *prunes* a partition P_i if none of the vectors in P_i satisfies F_j, i.e., the j-th bit of $\overline{v}(P_i)$ is 0, denoted by $\overline{v}(P_i)_j = 0$. If F_j prunes P_i, then F_j prunes $|P_i|$ tuples, as each vector corresponds to a tuple. Note that the weight w_j of F_j is the number of queries F_j subsumes in the workload. If F_j prunes P_i, when we run all the queries in the workload, the sum of tuples that can be skipped would be $w_j \cdot |P_i|$. Given a partition P_i, we define the cost function $C(P_i)$ as the sum of tuples in P_i that can be skipped when we execute all the queries in the workload, we have:

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

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

$$\mathbb{C}(\mathcal{P}) = \sum_{P_i \in \mathcal{P}} C(P_i) \tag{2}$$

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

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

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

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

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

In this paper, we will focus on Problem 1.

4.2 NP-Hardness

We now prove that Problem 1 is NP-hard even in the special case where $p = |V|/2$, by reduction from hypergraph bisection [18]. Given a hypergraph instance $H = (V, E)$, where V is a set of vectices and E is a set of hyper edges. Since each hyper edge connects a set of vertices, we denote by $v_i \in E_j$ if v_i is in E_j. The hypergraph bisection problem finds a balanced 2-partitioning on the vectices such that the number of hyperedges across the two partitions is minimum.

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

$$\mathbb{C}(\mathcal{P}) = n_1 m_1 + n_2 m_2 + n m_c \tag{3}$$

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

$$\mathbb{C}(\mathcal{P}) = \frac{1}{2} n m_1 + \frac{1}{2} n m_2 + n m_c = n m + n m_c \tag{4}$$

Since n and m are constant, minimizing Equation 4 is equivalent to minimizing m_c. Hence, an optimal solution to our problem solves the hypergraph bisection problem on H.

From Equation 3, we can see that our objective function involves the product of number of vertex and number of edges for each vertex partition. We conjecture that Problem 2, the variant that does not have the balance constraint, is also NP-hard.

4.3 Reduction Step

The partitioning problem defined in Section 4.1 is solely based on the vectors, not the actual tuples. As an optimization step, we *group-by* the vectors into (vector, count)-pairs. This step does not affect the quality of partitioning, since the same vectors should go to the same partition anyway. Each vector now is associated with a weight, denoting the number of tuples it represents. To compute Equation 1, we simply replace $|P_i|$ with the weighted sum of all the vectors in P_i.

Note that this step is the key to scale our partitioning techniques to large datasets. Theoretically, the number of distinct vectors is upper-bounded by $min(2^m, n)$. In practice, however, this number is usually very small, e.g., $<10k$, as many tuples may have the same feature vectors. We will empirically examine this number in Section 7.

4.4 The Bottom Up Framework

Since Problem 1 is NP-hard, we will turn to Ward's method as a heuristic algorithm to find an approximate solution efficiently. Ward's method [36], originally proposed for minimizing the error sum of squares, is a general bottom-up clustering framework and has been used for various objective functions [29].

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

$$\delta(P_i, P_j) = \mathbb{C}(\mathcal{P} \cup \{P_i \cup P_j\} - \{P_i, P_j\}) - \mathbb{C}(\mathcal{P}) \tag{5}$$

When we merge P_i and P_j, their union vectors are `OR`ed, i.e.,

$$\overline{v}(P_i \cup P_j) = \overline{v}(P_i) \vee \overline{v}(P_j) \tag{6}$$

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

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

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

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

Lemma 1 guarantees that our objective function can fit in Ward's method correctly. The objective $\mathbb{C}(\mathcal{P})$ has the maximal value when every vector is a partition by itself.

We iteratively find a pair of points whose merge hurts $\mathbb{C}(\mathcal{P})$ the least.

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

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

A straightforward implementation of this algorithm has a complexity of $O(n^2 \log n)$ [29]. As bottom-up clustering is a classic algorithmic framework, scalable implementations have been proposed, e.g., [38, 5]. Due to the reduction step, the input size becomes much smaller than the number of tuples. Our results in Section 7 indicate that the partitioning algorithm can run efficiently even with a naïve implementation, and the cost bottleneck of the entire blocking process is still on the actual data movement, not on the partitioning algorithm.

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

5. FEATURE-BASED DATA SKIPPING

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

As mentioned earlier, many query engines support data skipping using the value ranges or bloom filters. We could import our block data to these systems and rely on the built-in skipping mechanism. This, however, may miss some data skipping opportunities. For example, the features we used to guide partitioning can involve multiple columns (e.g.,F_3 in Figure 2); checking the aggregates of the two columns individually may not be able to prune the block, while maintaining multi-dimensional statistics is expensive. In addition, it is unclear how the aggregates can be used to prune string matching and general user-define functions. To fully exploit our blocking scheme for data skipping, we propose a feature-based skipping mechanism, which can be used in conjunction with existing data skipping mechanisms based on the aggregates.

Block Catalog. After the blocking, we obtain a union vector for each block. If the i-th bit of the union vector is 0, then we know that no tuple in this block satisfies feature i. Therefore, the queries that are subsumed by feature i can safely skip this partition. To realize this, we store the features used and one union vector for each block in a *block catalog*, which can be part of the system catalog in a database system. Figure 5 shows the partition catalog for the example blocking in Figure 2. Note that the block catalog is very compact: we store the features once and then one bit vector for each block.

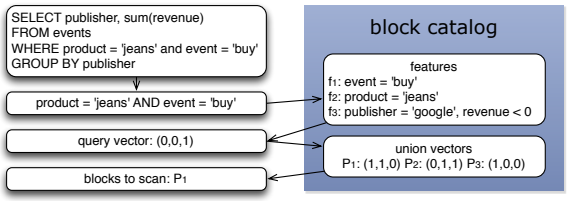


Figure 5: Data Skipping in Query Execution

Query Execution. Let us see how a query interacts with the block catalog in Figure 5. When a query comes, we first extract the filter operator from the query. We check which features in the block catalog can subsume this query. The subsumption check can be implemented as hard-coded rules. We find that F_1 and F_2 can subsume the query. We represent this information by a *query vector* $(0, 0, 1)$, the i-th bit of which is 0 if i-th feature subsumes the query. We then compute, for each block, a bitwise `OR` between the query vector and the union vector. For any block, if the resulting vector of the `OR` operation has at least one 0 bit, then this block can be skipped. In Figure 5, the `OR`ed vectors for P_1, P_2 and P_3 are $(1, 1, 1)$, $(0, 1, 1)$ and $(1, 0, 1)$ respectively. Thus, we know that we only need to scan P_1. This information is then passed to table scan operator.

The above procedure happens before the table scan operator and is independent of the rest of query execution. As we can see, we only need to maintain minimal metadata and add an simple module between query compilation and the start of the query execution. We have implemented this functionality in less than 100 lines of code in Shark [37].

6. DISCUSSION

In this section, we discuss the practical issues of using our techniques.

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

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

Parameter Selection. Two key parameters were used in our blocking process, namely, $numFeat$, the number of features, and $minSize$, the minimum number of tuples per block. For a real-world workload, we expect $numFeat$ to be small. We can use cross validations to choose an appropriate number of features that prevents under- and over-fitting.

The choice of $minSize$ can depend on the underlying system. In our prototype on Shark, we set $minSize$ to be $50,000$, which results in 64-128MB blocks. Each block nicely fits with a Spark/HDFS block. Intuitively, if a block is bigger than 128MB, the system will break it down into many HDFS blocks anyway. If the blocks are too small, the metadata storage and skip checking overhead may become significant.

Overhead of Skip Checking. In many large-scale query engines, the query optimizer runs in a single node. We can easily perform the skip checking (Section 5) in a single-node optimizer. This might seem counter-intuitive, as a large table can have many blocks. To see if this makes sense, we can do some back-of-the-envelope analysis. Assuming an average block size of 128MB, 100TB of data would have less than 1 million blocks. It only takes 13MB to store the bit vectors for all these partitions if each vector is 128-bit, i.e., 128 features used. The block catalog can easily fit in the memory of a single machine. In a modern CPU, a bitwise operation of two 128-bit vectors can be computed using only a single instruction. The block checking incurs minimal overhead even for interactive queries.

Objective Function. Our objective function for partitioning (Equation 2) is defined as the sum of query costs in the workload, where the cost of a query is quantified as the number of tuples scanned. We chose the current objective function as it is simple, commonly used in database designs (e.g., [20]) and works reasonably well in practice. In some applications, we may want to define the objective differently, e.g., we can maximize the number of queries in the workload that can finish in 30 seconds. In this case, we may need to estimate the query costs and weight the queries based on their costs. Incorporating these in our framework would be an interesting avenue of future work.

7. EXPERIMENTAL EVALUATION

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

7.1 System Prototype

We prototype our blocking techniques on Shark [37], a fully Apache Hive [8]-compatible data warehousing system using Apache Spark [25] as runtime. Shark parses and compiles HiveQL (SQL-like) queries to a query plan, which are then translated to Spark tasks. A Shark table is stored as a Spark data abstraction called Resilient Distributed Dataset (RDD), which is physically stored as a list of data blocks, each of which can be either memory- or disk-resident. Each block in Spark is a data processing unit and has a default size of 128MB. Shark supports data skipping over such data blocks. At data import time, Shark collects the data statistics for each block. These statistics are maintained in the system catalog. Before a table scan, the query filter is applied on these statistics, and then the block ids to be scanned are passed to the table scan operator.

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

Workload Analyzer. We can collect a query trace from the query logging system of Shark or from an external source. We used Shark's query parser to convert each query string into a set of conjunctive filter operators. We implemented a `isSubsume`(f_1, f_2) function using a set of rules to check if filter f_1 subsumes filter f_2. A module was added in Shark to implement the techniques in Section 3.

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

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

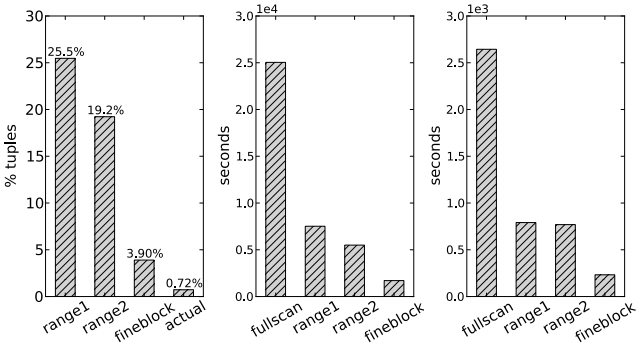
Catalog Update. We added the metadata described in Section 5 to the Shark system catalog. A table scan now can utilize two tiers of skipping mechanisms: our feature-based skipping (Section 5) and the existing skipping.

7.2 Datasets

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

TPC-H Skewed. The TPC-H query generator assumes a uniform distribution for the predicate constants. In many real-world workloads, the constant distributions are skewed. For example, *region='North America'* may be queried much more often than the other regions for a U.S. company. To test a skewed distribution, we modified the TPC-H query generator to follow a Zipf distribution. As most parameters in the query templates have a small number of possible values, we chose a high skew factor of 3. For example, `REGION` only has 5 possible values, and by using a skew factor of 3, the most frequent 20% values occur in 84.3% of the generated queries. Note that we only added the skewness to the non-date filters, while [DATE] is still uniformly distributed. We similarly generate 800 train queries and 80 test queries under this Zipf distribution.

Conviva. The Conviva data is an anonymized user access log of video streams. The data consists of a single large fact table with 104 columns, such as customer ID, city, media URL, genre, date, time, browser type and request response time. We also obtained an in-production SQL query trace from Conviva, which has 735 queries for problem diagnosis and data analytics issued on the log data. The queries were between 08/01/2012 and 11/30/2012. We split the query



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

Figure 6: Query Performance on TPC-H

trace at the date of 11/24/2012, which results in 674 training queries (before 11/24/2012) and 61 testing queries (on and after 11/24/2012). Based on the training queries, we partition the log data from 11/24/2012 to 11/30/2012. This snapshot of the log has 680 million tuples and is roughly 1 TB in size when stored as text. We evaluate the performance of running the test queries on the blocked data.

7.3 TPC-H Results

7.3.1 Query Performance

We evaluate the effect of our blocking techniques on query performance. We measure the number of tuples scanned and end-to-end query response time for different blocking and skipping schemes in Figure 6. Specifically, we compare the following alternatives:

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

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

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

`fineblock`: This is our approach. We first partition by month on *o_orderdate*. We use the workload analyzer to extract 15 features from the 800 training queries (by setting $numFeat$=15). Note that we do not consider any date filters as features and will rely on the month partitions to prune data. Using these features, we run a partitioner instance on each month partition in parallel. An average month partition has 7.7 million tuples. By setting $minSize$= $50k$, a month partition has around 100 blocks. The total number of blocks is roughly 8000. We used both our feature-based skipping (Section 5) and Shark's existing skipping.

We evaluate the performance of running 80 test queries (as mentioned in Section 7.2) using the above alternatives. Figure 6(a) shows the percentage of tuples scanned for the 80 queries relative to `fullscan`. As we can see, the existing data skipping with `range1` and `range2` only scan 25% and 19% of the tuples scanned by `fullscan` respectively. The tuples scanned by `fineblock` is only 3.9% of `fullscan`, a 5x improvement over the manual range partitioning scheme `range2`. As a reference, the bar `actual` shows the percentage of tuples that must be scanned, i.e., the tuples that actually satisfy the filters, which is 0.7% of `fullscan`.

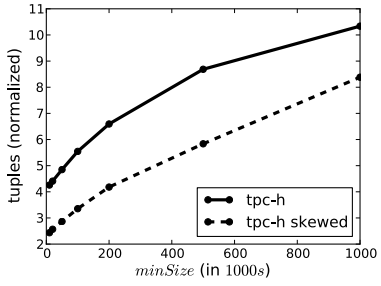


Figure 7: Effect of $minSize$ on TPC-H

To test the end-to-end query response time, we consider two scenarios: when the table is entirely on disk and when the data is entirely in memory. Figure 6(b) shows the query response times for on-disk data. We run the 80 test queries in a sequence and record the sum of their response time. We cleared the OS cache before running each query. As shown, the query response time for `range1` and `range2` are 30% and 21% of that for `fullscan`. `fineblock` only took 7% of the time for *fullscan*, 23% for `range1` and 30% for `range2`. This is an 3-4x improvement over the range partitioning schemes. Figure 6 shows the query response time for memory-resident data. In Shark, we can simply cache a table in memory by executing:

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

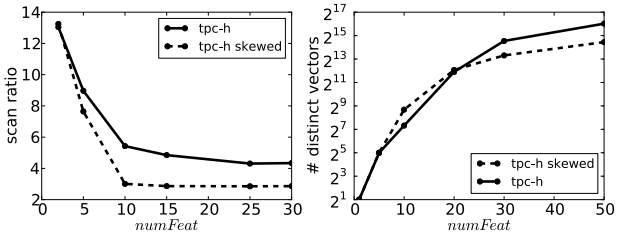
We configured our distributed RAM cache to be large enough to hold the entire table. As we can see, `fineblock` only took 30% and 32% of the time taken for `range1` and `range2`, respectively. This is roughly a 3x improvement. It is interesting to note that the end-to-end improvement for in-memory data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

We also tested the end-to-end performance for TPC-H Skewed. The amount of tuples scanned by `fineblock` is only 8% and 10% of that by `range1` and `range2` respectively. For the on-disk data, `fineblock` took 20% and 23% of the time for `range1` and `range2` respectively; for the in-memory data, `fineblock` took 22% and 29% of the time for `range1` and `range2` respectively. Note that our improvement for TPC-H Skewed is better than for TPC-H. The skewness in the filter distribution allows a small number of features to subsume even more queries, and thus makes our techniques even more effective for TPC-H Skewed.

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

7.3.2 Effect of $minSize$

Intuitively, the smaller the block size is, the more chance we can skip data. In Figure 7, we plot the total number of tuples scanned using our approach for answering the 80 test queries in TPC-H and TPC-H Skewed by varing $minSize$, with $numFeat$=15. Since the two curves represent two different workloads, for fair comparison, we plot the ratio of the number of tuples scanned to the number of tuples that have to be scanned. Thus, a y-value of 5 in the curve means we scanned 5 times as many tuples as necessary.



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

Figure 8: Effect of $numFeat$ on TPC-H

We make several observations. First, for both curves, we scan more data as the block size increases. Second, data skipping is even more effective on TPC-H skewed. Recall our workload assumptions in Section 2, many real workloads tend to have skewed predicate distribution. If the filter predicates are skewed, a small set of features can cover more queries. Third, the number of tuples scanned is not sensitive to the block size. In particular, increasing $minSize$ from $5k$ to $200k$ only make the scan twice as much for both workloads. This gives us a wide range of choosing $minSize$. For example, in our experiment on Shark, we set $minSize$=$50k$, which make each block nicely fit in a Spark/HDFS block file (128MB by default).

7.3.3 Effect of $numFeat$

Figure 8 plots the number of tuples scanned by varying $numFeat$. The numbers are also normalized as in Figure 7. As we can see, when using too few features, e.g., < 5, we have to scan a lot of more tuples, as these features are not representative enough for the workload. As we add more features, the effectiveness of skipping quickly stabilizes and TPC-H Skewed consistently benefits more from our blocking. We can see that, for both TPC-H and TPC-H Skewed, a small set of features is sufficient. Even though the predicates in TPC-H are uniformly distributed, the fixed filters play an important role as features. For example, the feature $l\_returnflag$ = 'R' in template q_{10} can subsume 100 out of 800 training queries (and also 10 out of 80 testing queries), since all the queries generated from template q_{10} have this filter. TPC-H Skewed performs even better due to the skewness in the parametric predicates. This curve also suggests that adding more features will not significantly hurt the effectiveness of skipping, though this may hurt the blocking efficiency. This is because the highly weighted features will dominate and adding more features with small weights will not dramatically change the blocking solution.

As discussed, one important effect of $numFeat$ is on the number of distinct feature vectors. We cannot afford to run a bottom-up clustering algorithm if the number of distinct vectors is too large, e.g., close to the number of tuples. Theoretically, with m features and n tuples, the number of distinct vectors is upper-bounded $min(2^m, n)$. In Figure 8, we plot the actual number of distinct vectors by varying $numFeat$ in base-2 log scale. The plot shows that the number of distinct vectors is much smaller than either 2^m or n. This is because these features have low selectivities and can be correlated.

7.3.4 Blocking Time

Figure 9 shows the breakdown cost of blocking a month partition in TPC-H. An average month partition has 7.7 million tuples and 8G in size and can be divided into roughly

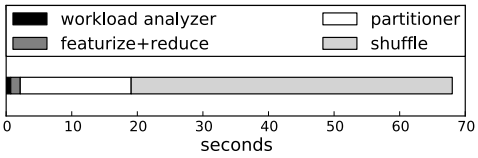


Figure 9: Breakdown of blocking time

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

7.4 Conviva Results

7.4.1 Query Performance

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

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

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

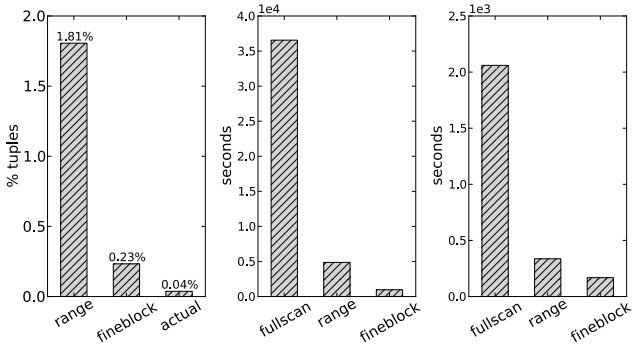
`fineblock`: We first partition by date. After extracting 40 features from the training queries ($numFeat$=40), we block each date partition with $minSize$ =$50k$. We use both our feature-based and Shark's existing skipping mechanisms.

Figure 10(a) shows the percentage of tuples scanned for evaluating the test queries, relative to `fullscan`. As we can see, `range` already reduced the scan to be 1.81% of `fullscan`, `fineblock` only scans 0.23% of the data. The average selectivity of these queries (i.e., `actual` is 0.03% of `fullscan`. Figure 10(b) shows the query response times for on-disk data. We can find that `range` spent 13% the time as `fullscan`, and `fineblock` further reduced the time to be 2.6% of `fullscan`. This is a 5x improvement over `range`. Figure 10(c) shows the query time for in memory data. We find that `range` and `fineblock` used 16% and 8.1% of the time taken by `fullscan`. The improvement of `fineblock` over the range partitioning schemes is 2x.

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

7.4.2 Effect of $numFeat$

We now study the effect of $numFeat$ on the Conviva workload. Figure 11(a) plots the scan ratio by varying $numFeat$. The number of tuples scanned is dramatically reduced as $numFeat$ is increased from 2 to 20. As we continue to add more features, however, the curve is relative flat. This is



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

Figure 10: Query Performance on Conviva

inline with the results on TPC-H (Figure 8), except that this curves starts to stabilize at 20 instead of 10. This result confirms that a small number of features is sufficient for a real workload. In Figure 11(b), we can see that the number of distinct vectors is small.

We can conclude that, in a real-world workload, (1) our blocking can effectively help queries skip data and in turn reduce the query response time significantly for both on-disk and in-memory data, and (2) the blocking can be done effectively and efficiently with a small number of features.

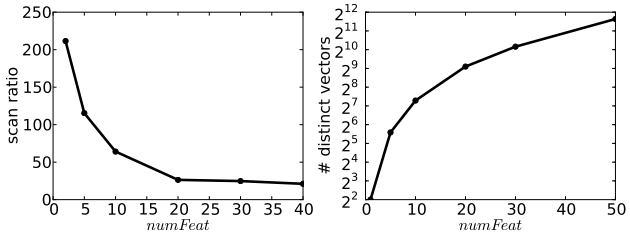
8. RELATED WORK

In this section, we review the related work.

Horizontal Partitioning. Range and hash partitioning are the most widely used horizontal partitioning techniques and serve for many purposes, such as load balancing. Advanced and automated partitioning techniques have also been extensively studied [15, 39, 30, 11], but they were built on top of range or hash partitioning. Although the blocking problem we study is a form of horizontal partitioning, we generate a tuple-level partitioning map by solving an optimization problem instead of using explicit range constraints. Our techniques can be used to further segment a date-range partition into finer blocks for skipping purposes. While Schism [17] also used fine-grained tuple-level partitioning, they had a different objective, which is reducing cross-machine transactions for OLTP workloads.

Materialized Aggregates and Skipping. Many databases utilize range partitions to enhance query performance. Partition pruning (e.g., Oracle [3] and Postgres [4]) allows queries to skip partitions based on partition key ranges (e.g., date). Extending this idea, other works [28, 34] have proposed maintaining small materialized aggregates (SMAs) for each range-partitioned block, such as min, max, count, sum and histograms for each column. For a given query, these SMAs can be used to classify the data blocks into three categories: (C1) *irrelevant* blocks, the ones in which no tuple satisfies the query, (C2) *relevant* blocks, the ones in which all the tuples satisfy the query, and (C3) *suspect* blocks, the ones in which some tuples may satisfy the query.

Obviously, the blocks in C1 can be safely skipped. The C3 blocks can also be skipped when the requested aggregates can be answered by SMAs. We note that the opportunity of identifying and skipping C3 blocks can be rather limited in practice, as it requires that 1) all the (conjunctive) filters of the query subsume the block's min and max ranges and 2) all the requested aggregates can be answerable by the chosen



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

Figure 11: Effect of $numFeat$ on Conviva

SMAs and if the query contains a group-by, the SMAs must be stored for all of the potentially relevant groups. A number of systems, e.g., [7, 37, 6, 35], use a simplified version of SMAs, which only skip C1 blocks and do not distinguish C2 from C3 blocks. Similar to these systems, our techniques only consider skipping C1 blocks. While these previous approaches are built on top of range partitioning, our main contribution is to develop a novel fine-grained partitioning technique based on workload analysis, which can turn more blocks into C1 blocks than a range partitioning. Nevertheless, the idea of using SMAs to skip C3 blocks can also be implemented on top of our partitioning scheme.

Materialized View Selection. Our work is related to the well studied problem of materialized view selection (MVS), since both exploit pre-computations for query performance improvement. These two problems, however, differ fundamentally in several ways. First, our partitioning is at the file-organization level, while MVS is at the application level; in fact, our techniques can be used to partition large materialized views. Second, we utilize pre-computation to guide the tuple re-arrangement and only need to maintain minimal metadata (i.e., a bit vector per block), while MVS does not change the original data but store the precomputed results, which can incur significant space overhead, e.g., data cubes. Third, MVS is an optimization problem constrained on space [20, 10] or maintenance cost [19], while ours is constrained on the number of partitions.

Similar to our approach, some MVS approaches also exploit workload information. Most of these focus on the group-by columns of the queries (e.g., [16, 14]) for deciding which columns to pre-aggregate. Others (e.g., [10]) also consider which columns are filtered on for selecting indexes and materialized views in an integrated manner. Different from this work, our workload analysis aims to identify representative filters, including both filter columns and constants, and their subsumption relations for skipping purposes. We consider all kinds of filters, such as equality/range conditions, string matching and user defined functions.

Workload-driven Physical Design. Many research efforts have been devoted to utilizing workload information for automating database design. For example, the AutoAdmin project [10, 11] integrates many physical design problems, such as selecting indexes and materialized views; Database Cracking [21] reorders the data columns as a byproduct of query processing to benefit future queries; ARF [12] tunes a range-based filter for skipping cold data; BlinkDB [31] prepares samples offline based on the workload.

Optimization Problems for Partitioning. Finding an optimal partitioning over a set of data points is an important problem in many applications, such as data mining, computer vision [33], gene expression analysis [23] and VLSI design [24]. The partitioning problem is NP-hard for many

objective functions, e.g., [13]. To the best of our knowledge, no existing work has formulated the k-MaxSkip problem before, although we found k-MaxSkip and $BalancedMaxSkip$ are closely related to several partitioning problems, such as hypergraph cut [24], discrete basis partitioning problem [27] and row-exclusive biclustering [26].

9. CONCLUSION

We presented fine-grained data blocking techniques, which partition the data tuples into blocks in a way that could help queries skip data. The key components were: (1) a workload analyzer, which generates a set of features from a query log, (2) a partitioner, which computes a blocking scheme by solving a optimization problem, (3) a feature-based block skipping framework used in query execution. We prototyped our techniques on Shark, which showed that our blocking modules can be easily added to an existing query engine and the data flow can be executed using standard data marshalling steps, such as map and reduce.

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

Acknowledgements. We thank Ion Stoica for providing the Conviva dataset. We also thank Sameer Agarwal, Ameet Talwalkar, Di Wang, Jiannan Wang and the reviewers for their insightful feedback. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Apple, Inc., Cisco, Cloudera, EMC, Ericsson, Facebook, GameOn-Talis, Guavus, Hortonworks, Huawei, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata, VMware, WANdisco and Yahoo!.

10. REFERENCES

[1] IBM Netezza. http://www.netezza.com.
[2] Impala. https://github.com/cloudera/impala.
[3] Oracle. http://docs.oracle.com/.
[4] Postgres. http://www.postgresql.org/docs/.
[5] H. Koga *et al.* Fast agglomerative hierarchical clustering algorithm using locality-sensitive hashing. *Knowledge and Information Systems*, 12(1), 2007.
[6] A. Hall *et al.* Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, 2012.
[7] A. Lamb *et al.* The Vertica analytic database: C-Store 7 years later. *VLDB*, 5(12):1790–1801, 2012.
[8] A. Thusoo *et al.* Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2):1626–1629, 2009.
[9] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *VLDB*, pages 487–499, 1994.
[10] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505, 2000.
[11] S. Agrawal, V. Narasayya, and B. Yang. Integrating vertical and horizontal partitioning into automated physical database design. In *SIGMOD*, pages 359–370, 2004.
[12] K. Alexiou, D. Kossmann, and P.-A. Larson. Adaptive range filters for cold data: Avoiding trips to Siberia. *PVLDB*, 6(14):1714–1725, 2013.
[13] D. Aloise, A. Deshpande, P. Hansen, and P. Popat. NP-hardness of euclidean sum-of-squares clustering. *Machine Learning*, 75(2):245–248, 2009.
[14] K. Aouiche, P.-E. Jouve, and J. Darmont. Clustering-based materialized view selection in data warehouses. In *Advances in Databases and Information Systems*, pages 81–95, 2006.
[15] B. Bhattacharjee *et al.* Efficient query processing for multi-dimensionally clustered tables in DB2. In *VLDB*, pages 963–974, 2003.
[16] E. Baralis, S. Paraboschi, and E. Teniente. Materialized views selection in a multidimensional database. In *VLDB*, volume 97, pages 156–165, 1997.
[17] C. Curino, E. Jones, Y. Zhang, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. *PVLDB*, 3:48–57, 2010.
[18] M. R. Garey and D. S. Johnson. *Computers and Intractability*. 1990.
[19] H. Gupta and I. Mumick. Selection of views to materialize under a maintenance cost constraint. In *ICDT*, pages 453–470, 1999.
[20] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD Record*, volume 25, pages 205–216, 1996.
[21] S. Idreos, M. L. Kersten, and S. Manegold. Database cracking. In *CIDR*, pages 68–78, 2007.
[22] J. Han *et al.* Frequent pattern mining: current status and future directions. *DMKD*, 15(1):55–86, 2007.
[23] D. Jiang, C. Tang, and A. Zhang. Cluster analysis for gene expression data: a survey. *IEEE TKDE*, 16(11), 2004.
[24] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar. Multilevel hypergraph partitioning: applications in VLSI domain. *IEEE Trans. on VLSI*, 7(1):69–79, 1999.
[25] M. Zaharia *et al.* Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, 2012.
[26] S. C. Madeira and A. L. Oliveira. Biclustering algorithms for biological data analysis: A survey. *IEEE Trans. on Computational Biology and Bioinformatics*, 1(1):24–45, 2004.
[27] P. Miettinen, T. Mielikainen, A. Gionis, G. Das, and H. Mannila. The discrete basis problem. *IEEE TKDE*, 20(10):1348–1362, 2008.
[28] G. Moerkotte. Small materialized aggregates: A light weight index structure for data warehousing. In *VLDB*, pages 476–487, 1998.
[29] A. Rajaraman and J. D. Ullman. *Mining of massive datasets*. Cambridge, 2012.
[30] J. Rao, C. Zhang, N. Megiddo, and G. Lohman. Automating physical database design in a parallel database. In *SIGMOD*, pages 558–569, 2002.
[31] S. Agarwal *et al.* BlinkDB: queries with bounded errors and bounded response times on very large data. In *EuroSys*, pages 29–42, 2013.
[32] S. Melnik *et al.* Dremel: interactive analysis of webale datasets. *PVLDB*, 3(1-2):330–339, 2010.
[33] J. Shi and J. Malik. Normalized cuts and image segmentation. *IEEE TPAMI*, 22:888–905, 1997.
[34] D. Ślęzak, J. Wróblewski, V. Eastwood, and P. Synak. Brighthouse: An analytic data warehouse for ad-hoc queries. *PVLDB*, 1(2):1337–1345, 2008.
[35] V. Raman *et al.* DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.
[36] J. Ward. Hierarchical grouping to optimize an objective function. *J. American statistical association*, (301):236–244.
[37] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and Rich Analytics at Scale. In *SIGMOD*, pages 13–24, 2013.
[38] T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *SIGMOD*, pages 103–114, 1996.
[39] J. Zhou, N. Bruno, and W. Lin. Advanced partitioning techniques for massively distributed computation. In *SIGMOD*, pages 13–24, 2012.