A Partitioning Framework for Aggressive Data Skipping

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

ABSTRACT

We propose to demonstrate a fine-grained partitioning framework that reorganizes the data tuples into small *blocks* at data loading time. The goal is to enable queries to maximally skip scanning data blocks. The partition framework consists of four steps: (1) workload analysis, which extracts *features* from a query workload, (2) augmentation, which augments each data tuple with a feature vector, (3) reduce, which succinctly represents a set of data tuples using a set of feature vectors, and (4) partitioning, which performs a clustering algorithm to partition the feature vectors and uses the clustering result to guide the actual data partitioning. Our experiments show that our techniques result in a 3-7x query response time improvement over traditional range partitioning due to more effective data skipping.

1. INTRODUCTION

A rapidly increasing number of applications require *interactive* data analysis on enormous datasets. This necessitates the capability of low-latency query processing for large-scale query engines. Among others, an effective way to improve query latency is to reduce the unnecessary data access. For example, column store techniques have been widely adopted to avoid touching irrelevant columns. To reduce the scan of unwanted tuples, there is an increasing interest in *data skipping* in recent systems [4, 5, 2]. By partitioning the data into small *blocks*, these systems associate each block with some metadata, such as the min and max values of each column. A query can first evaluate its filter against these block-level metadata and decide which blocks can be safely skipped.

Data skipping can be viewed as a generalization of partition pruning. On a horizontally partitioned table, partition pruning allows a query to prune partitions based on their associated partition key ranges. Data skipping extends the idea of partition pruning in several ways. First, storing the min and max values of each column enables skipping based on non-partition-key columns, especially on the columns that are naturally clustered with the partition keys. Second, for better skipping chances, data skipping is often considered on very small blocks, e.g., partitions consisting of 1,000's to 10,000's of tuples [4, 2] or simply HDFS blocks [5].

The effectiveness of data skipping depends on the interplay between query filters and the partitioning scheme. In a data warehouse environment, for example, a time-range partitioning scheme offers great opportunities for partition pruning, as most queries are observed to have time-range filters. While range partitioning is simple and useful for data warehouse operations, it may not be ideal for generating a large number of small blocks for effective data skipping. Specifically, range partitioning lacks of a principled way of: (1) setting the fine-grained value ranges on each column that matches the data skew and workload skew, (2) allocating the number of partitions for different partitioning columns and (3) capturing inter-column data correlation and filter correlation.

We propose to demonstrate a fine-grained partitioning framework that, at data loading time, partitions data tuples into small, balanced blocks with a goal of maximizing data skipping. We call this framework WARP, based on its a four-step workflow: **W**orkload analysis, **A**ugmentation, **R**educe, and **P**artitioning. We first analyze a query log offline and extract a set of representative query filters as *features*. Intuitively, we want a small set of features that can *subsume* as many queries in the workload as possible. Given these features, a set of tuples can be succinctly represented as a (much smaller) set of feature vectors. We then partition these feature vectors by solving an optimization problem. Finally, the partitioning scheme of these feature vectors will be used to guide the partition of actual data tuples. After partitioning the tuples into data blocks, we store the features and some concise metadata for each block in the system catalog. When a query comes, we first check if this query can be *subsumed* by any of the features and then decide which blocks can be skipped.

Instead of specifying the partitioning columns and value ranges as in range partitioning, WARP factors in the *common interests* of the workload as *features* and, based on these common interests, finds a partitioning scheme by solving an optimization problem. The fine-grained tuple-level partitioning decision output by WARP offers greater flexibility and better chances for data skipping. Since WARP produces very small blocks, it can be used to further segment traditional range partitions. In fact, as data is often batch inserted in a data warehouse environment, it is a good practice to apply WARP within each individual time-range (e.g., date) partition, instead of moving tuples across different

Proceedings of the VLDB Endowment, Vol. 7, No. 13
Copyright 2014 VLDB Endowment 2150-8097/14/08.

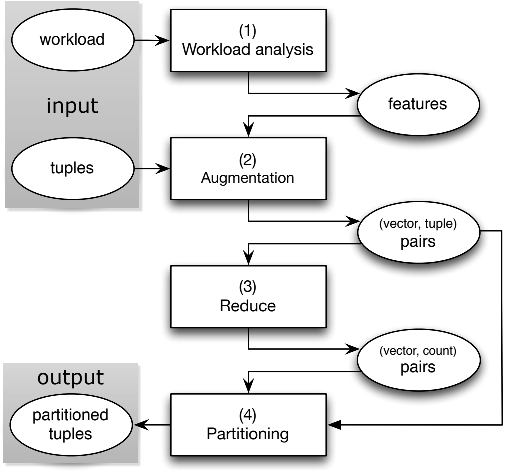


Figure 1: The WARP Workflow

time-range partitions.

The remainder of this demonstration proposal is organized as follows. Section 2 outlines the workflow and challenges of WARP and Section 3 discusses how queries can utilize the metadata generated by WARP to skip data. In Section 4, we present our WARP prototype on Shark [5], an open-source data warehouse system running on Spark [6]. Section 5 reports performance results. Finally, we propose the demonstration details in Section 6.

A full technical paper describing our partitioning framework can be found in [3].

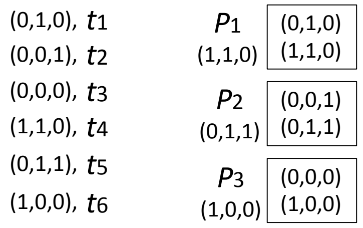
2. THE WARP WORKFLOW

The input is a collection of tuples, which can be a table or a table partition such as a date partition, and a workload represented as a collection of queries. The query workload can be obtained from the query logs. We now walk through the four steps of the workflow, as depicted in Figure 1.

2.1 Workload Analysis

Workload analysis is an offline process that extracts a set of features from the query workload. Its core is a frequent itemset mining problem. We model as an item each predicate or each disjunction of predicates and model each query as a set of (conjunctive) items. We run a frequent itemset mining algorithm to find frequent *predicate sets*. Note that our goal here is to identify a set of features that *subsume* as many queries as possible. Therefore, in the mining process, we count the number of queries a predicate set subsumes, instead of the number of queries in which a predicate set occurs as in a traditional sense. Some predicate sets may be redundant due to the subsumption relations. For instance, *revenue* < 0 may be redundant if *revenue* < 10 is also in the result. Thus, we finally select a set of features by removing redundant frequent predicate sets. As shown in Example 1, the output of workload analysis is a set of features, each of which is associated with an integer indicating how many queries it subsumes.

EXAMPLE 1. *Features output by workload analyzer:*
F_1*: event = 'buy', 60*
F_2*: product = 'jeans', 20*
F_3*: publisher = 'google'* $\wedge$ *revenue* < 0*, 10*



(a) (vector, tuple)-pairs (b) partitions

Figure 2: A Partitioning Example

2.2 Augmentation

Note that the features from Step 1 are essentially filters. Let us say there are m features. We evaluate each tuple against these m features and generate a m-bit vector, where the i-bit indicates whether the tuple satisfies the i-th feature. When a set of tuples are batch-loaded, e.g., as a date partition, we scan these tuples once and batch evaluate these filters on each tuple. This step transforms each tuple to a (vector, tuple)-pair, as shown in Figure 2(a).

2.3 Reduce

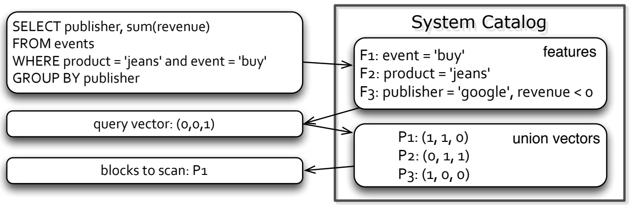
The partitioning algorithm will be solely based on the feature vectors, not the actual tuples. As many different tuples can have the same bit vector, we *group-by* the (vector, tuple)-pairs from Step 2 into (vector, count)-pairs. Note that the number of distinct vectors can be much smaller than the number of data tuples. Thus, this is a critical step that reduces the input size of the partitioning algorithm.

2.4 Partitioning

The partitioning step runs a clustering algorithm to partition the (vector, count)-pairs. Figure 2(b) illustrates an example partitioning scheme over the vectors. Note that the partitioning of the vectors governs the partitioning of the tuples. For example, all the tuples with vector $(0, 1, 0)$ as key will be routed to partition P_1. For each partition of vectors, we derive a *union vector*, which is the union of all vectors in it. For example, the union vector of partition P_1 is $(1, 1, 0)$. Since the third bit of this vector is 0, we know that no vector in P_1 has the third bit on, and hence, no tuple in P_1 satisfies feature F_3. In this case, all the queries subsumed by F_3 can safely skip scanning P_1.

Intuitively, our objective is to find a partitioning that maximizes the (weighted) sum of zeros in the union vectors. This objective is quite different from traditional clustering objectives such as k-means and distance-based metrics. We proved that finding an optimal partitioning under this objective is NP-hard [3]. We adopt the classic bottom-up clustering framework as a heuristic: every vector starts as a partition by itself, and at each iteration, we merge the two vectors that hurtsa the objective the least. To make the partitions almost balanced, we remove a partition from further merging once its size reaches a parameter *minSize*. By consulting the partitioning result of vectors, each of the (vector, tuple)-pairs (from the Augmentation step) is routed to its destination partition.

The WARP workflow can be executed 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, WARP can be postponed.



```
SELECT publisher, sum(revenue)
FROM events
WHERE product = 'jeans' and event = 'buy'
GROUP BY publisher
```

Figure 3: Skip Checking in Query Processing

Having presented the WARP workflow, we next discuss how the queries can perform aggressive data skipping on our partitioned data.

3. SKIP-AWARE QUERY PROCESSING

At the end of the WARP workflow, we add the features used in WARP to the system catalog. In addition, we store one union vector (e.g., Figure 2(b)) for each block in the system catalog. After this step, we can safely drop the feature vectors associated with every tuple.

The i-th bit of a union vector is 0 only if no tuple in this block satisfies feature i. In this case, the queries that are subsumed by feature i can safely skip this partition. To make use of this information during query processing, we need a two-step skip checking mechanism, as illustrated in Figure 3:

(1) Feature matching. Given the features we store in the system catalog. When a query comes, we first check which features *subsume* this query. We then encode this information in a *query vector*. The i-th bit of the query vector is 0 if this query can be subsumed by the i-th feature. For example, in Figure 3, we find that feature F_1 and F_2 can subsume the query, but not F_3, and thus we construct a *query vector* $(0, 0, 1)$.

(2) Union vector checking. Given that we store a union vector for each block, we now check the query vector against these union vectors and decide which blocks can be skipped. Specifically, we perform a bitwise `OR` operation between the query vector and each union vector. For any block, if the result of the `OR` operation has at least one 0 bit in it, then this block can be safely skipped. In Figure 3, 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. Finally, this information is passed to the table scan operator.

The above procedure happens before the actual query execution. As we can see, we only need to maintain a set of features used and one bit vector for each block. This metadata is very small and can be stored in main memory. The skip checking only involves a feature matching step and, for each block, a bit-OR operation. This incurs little overhead to query latency. Note that this skipping mechanism can be used in conjunction with existing skipping mechanisms based on per-column value ranges.

4. SYSTEM PROTOTYPE

We prototype our blocking techniques on Shark [5], a fully Apache Hive-compatible data warehousing system using Apache Spark [6] as runtime.

Shark Background. Shark uses Hive's query parser to parse and compiles HiveQL (SQL-like) queries to a query plan, and then translates it 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 Spark block is a task processing unit and has a default size of 128MB. Shark can skip some of these blocks during a table scan. At data import time, Shark collects the data statistics for each block, such as the min and max values of each column. These block-level statistics are maintained in the system catalog. The table scan operator first fetches the query filter operators and applies them on these statistics to prune blocks. Only the blocks that are not pruned in this step are actually scanned.

We now briefly discuss how our techniques were implemented on Shark. First, we collect a query trace from the query logging system of Shark or from an external source. We use Shark's query parser (through Hive) 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 workload analysis module was added in Shark to compute frequent itemsets and remove redundant results based on subsumption, as discussed in Section 2.

Given the features, represented as filter operators, we simply wrote a Spark map function to transform each data tuple into a (vector, tuple) key-value pairs. Then we use a Spark reduce function to group-by these key value pairs. We implemented a bottom-up clustering algorithm as an independent module in Shark. Note that external libraries could be used here for an optimized implementation. We then built a customized partitioner class which implements the *Partitioner* interface in Spark. This partitioner stores a (vector, partition-id) map in memory and routes each (vector, tuple)-pair to its corresponding destination block. We then re-partition original table (an RDD) using this partitioner.

In the end, we added the metadata described in Section 3 to the Shark system catalog. A table scan in Shark with WARP can utilize the conjunction of two block skipping mechanisms: our feature-based skipping (Section 3) and the existing skipping based on value ranges.

5. PERFORMANCE

We conducted extensive evaluation using TPC-H and a real-world analytical workload. The experiments were run on an Amazon Spark EC2 cluster of 25 m2.4xlarge instances, each with 8 2.66 GHz CPU cores, 68.4 GB of RAM and 800 GB of disk. Here we present some performance highlight from TPC-H. For full experimental results, refer to [3].

Dataset. We denormalize the TPC-H tables against the lineitem table. With a scale factor of 100, the resulting table has roughly 600 million rows and is 700 GB in size. We select eight TPC-H query templates (q_3, q_5, q_6, q_8, q_{10}, q_{12}, q_{14}, q_{19}) that involve the *lineitem* table and have selective filters. The FROM clauses in these templates were all changed to be the denormalized table. 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.

We compare WARP against Shark's existing data skipping on top of range partitioning for running the 80 test queries. We manually devise a composite range partitioning scheme on {*o_orderdate*, *r_name*, *c_mktsegmt*, *quantity*} by identifying the frequently queried columns from the training

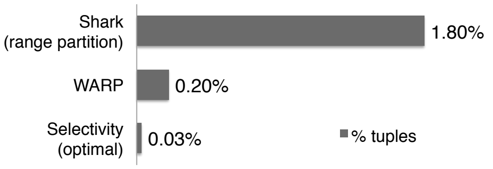


Figure 4: Number of Tuples Scanned

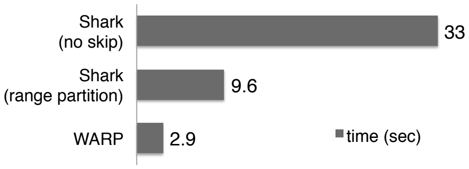


Figure 5: End-to-End Query Response Time

queries. For WARP, we first partition the data by month on *o_orderdate* and apply WARP on each month partition. We set the number of features as 15 and the partition size to be 50k tuples. We used both our feature-based skipping and Shark's existing skipping for WARP.

Figure 4 shows the number of tuples scanned by these two approaches. Shark's data skipping mechanism on range partitioning is decent, as it only scans 1.8% percent of the table for an average query. WARP brings down this number to be 0.2%. As a reference, we also plot the average selectivity of these queries, which is the minimum number of tuples that have to be scanned for answering these queries.

We compare the end-to-end query performance in Figure 5. Note that the table is memory resident. Without any skipping, Shark scans the whole table for every query, which takes 30 seconds on average. By switching on the skipping, the average query response time in Shark becomes 9.6 seconds. With WARP, it only needs 2.9 seconds, a 3× improvement.

To summarize, by deploying WARP on Shark, we significantly reduce the number of tuples scanned and this reduction effectively translates to a significant end-to-end query response time improvement.

6. DEMONSTRATION DETAILS

We will demonstrate our WARP prototype on Spark deployed on a Amazon EC2 cluster [1]. We will use the denormalized TPC-H dataset.

6.1 Demonstration Scenario

Our demonstration will put the conference attendees in the position of a database administrator (DBA) or a performance engineer. She would like to consider working with WARP to analyze her workload characteristics, load the data, and observe the query performance improvement over existing partitioning methods. We will walk through the following three steps:

Workload Analysis. To make the best use of WARP, we illustrate and visualize the key characteristics of the workload. The questions we aim to address in this part of demonstration are: (1) how predictable are future queries from a past-workload analysis? (2) how many features do we need? and (3) how representative are these features? At a web console, the attendees can tune some parameters, such as the number of features, and interactively observe the actual features extracted by WARP and graphs that summarize the statistics of workload.

Data Loading. After selecting the features, we use WARP to load data live from HDFS to Shark. The only parameter here is the size of each partition. The attendees get to tune this parameter and observe its effect on the breakdown cost and running statistics as WARP progresses.

Skipping in Action. At this stage, we have several copies of the table, including copies loaded with range partitioning and with WARP using different parameter choices. The attendees can choose a query from the workload or issue an ad-hoc query. For each query, we show how the posed query interacts with the metadata. We also demonstrate how the reduction of data scan translates to the improvement of end-to-end query response time, for both on-disk and in-memory data. In addition to the performance numbers, this part of demonstration helps attendees understand why some queries benefit more from the WARP partitioning than others. For comparison, we also print out block-level min and max values and show how they help queries skip blocks in existing systems.

6.2 Takeaway

This demonstration illustrates the reason behind WARP's significant performance benefits and its ease-of-use features, e.g., only two parameters to tune. Using WARP's buit-in workload analysis helps users understand their workload characteristics before setting up WARP. To deploy WARP on a query engine requires little effort, as it needs minimal modification on the table scan operator and is independent of the other parts of query execution. For instance, we modified Shark's table scan operator for WARP using less than 100 lines of Scala code.

7. REFERENCES

[1] Running Spark on Amazon EC2. https://spark.apache.org/docs/0.9.0/ec2-scripts.html.

[2] A. Hall, O. Bachmann, R. Büssow, S. Gănceanu, and M. Nunkesser. Processing a trillion cells per mouse click. *PVLDB*, 5(11):1436–1446, 2012.

[3] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *SIGMOD Conference*, pages 1115–1126, 2014.

[4] V. Raman *et al.* DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[5] 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.

[6] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2, 2012.