Industrial and Applications Paper

Herding the elephants: Workload-level optimization strategies for Hadoop

Sandeep Akinapelli
Cloudera, Palo Alto, CA
sakinapelli@cloudera.com

Ravi Shetye
Cloudera, Palo Alto, CA
ravi@cloudera.com

Sangeeta T.
Cloudera, Palo Alto, CA
sangeeta@cloudera.com

ABSTRACT

With the growing maturity of SQL-on-Hadoop engines such as Hive, Impala, and Spark SQL, many enterprise customers are deploying new and legacy SQL applications on them to reduce costs and exploit the storage and computing power of large Hadoop clusters. On the enterprise data warehouse (EDW) front, customers want to reduce operational overhead of their legacy applications by processing portions of SQL workloads better suited to Hadoop on these SQL-on-Hadoop platforms - while retaining operational queries on their existing EDW systems. Once they identify the SQL queries to offload, deploying them to Hadoop as-is may not be prudent or even possible, given the disparities in the underlying architectures and the different levels of SQL support on EDW and the SQL-on-Hadoop platforms. The scale at which these SQL applications operate on Hadoop is sometimes factors larger than what traditional relational databases handle, calling for new workload level analytics mechanisms, optimized data models and in some instances query rewrites in order to best exploit Hadoop.

An example is aggregate tables (also known as materialized tables) that reporting and analytical workloads heavily depend on. These tables need to be crafted carefully to benefit significant portions of the SQL workload. Another is the handling of UPDATEs - in ETL workloads where a table may require updating; or in slowly changing dimension tables. Both these SQL features are not fully supported and hence have been underutilized in the Hadoop context, largely because UPDATEs are difficult to support given the immutable properties of the underlying HDFS.

In this paper we elaborate on techniques to take advantage of these important SQL features at scale. First, we propose extensions and optimizations to scale existing techniques that discover the most appropriate aggregate tables to create. Our approach uses advanced analytics over SQL queries in an entire workload to identify clusters of similar queries; each cluster then serves as a targeted query set for discovering the best-suited aggregate tables. We compare the performance and quality of the aggregate tables created with and without this clustering approach. Next, we describe an algorithm to consolidate similar UPDATEs together to reduce the number of UPDATEs to be applied to a given table.

While our implementation is discussed in the context of Hadoop, the underlying concepts are generic and can be adopted by EDW and BI systems to optimize aggregate table creation and consolidate UPDATEs.

CCS Concepts

•Information systems → Database utilities and tools; Relational database model;

Keywords

Query optimization; Hadoop; Hive; Impala; BI reporting

1. INTRODUCTION

Large customer deployments on Hadoop often include several thousand tables many of which are very wide. For example, in the retail sector, we have observed customer workloads that issue over 500K queries a day over a million tables some of which have 50,000 columns. Many of these queries share some common clustering characteristics; i.e. in a BI or reporting workload we may find clusters of queries that perform highly similar operations on a common set of columns over a common set of tables. Or in an ETL workload, UPDATEs on a certain set of columns over a common set of tables may be highly prevalent. But at such large scales, detecting common characteristics, identifying the set of queries that exhibit these characteristics and using this knowledge to choose the right data models to optimize these queries is a challenging task. Automated workload level optimization strategies that analyze these large volumes of queries and offer the most relevant optimization recommendations can go a long way in easing this task. Thus for the BI or reporting workload, creating a set of aggregate tables that benefit performance of a set of queries is a useful recommendation; while for the ETL case detecting UPDATEs that can be consolidated together can help overall performance of the UPDATEs.

Aggregate tables are an important feature for Business Intelligence (BI) workloads and various forms of it are supported by many EDW vendors including Oracle [8], Microsoft SQL Server [7] and IBM DB2 [15] as well as BI tools such as Microstrategy [13] and IBM Cognos [9]. Here, data required by several different user or application queries are

joined and aggregated apriori and materialized into an aggregate table. Reporting and analytic queries then query these aggregate tables, which reduces processing time during query execution resulting in improved performance of the queries. This is an example aggregate table over the TPC-H workload schema:

```
CREATE TABLE aggtable_888026409 AS
 SELECT lineitem.l_quantity
  , lineitem.l_discount
  , lineitem.l_shipinstruct
  , lineitem.l_commitdate
  , lineitem.l_shipmode
  , orders.o_orderpriority
  , orders.o_orderdate
  , orders.o_orderstatus
  , supplier.s_name
  , supplier.s_comment
  , Sum (orders.o_totalprice)
  , Sum (lineitem.l_extendedprice)
 FROM   lineitem
  , orders
  , supplier
 WHERE  lineitem.l_orderkey = orders.o_orderkey
  AND lineitem.l_suppkey = supplier.s_suppkey
 GROUP  BY lineitem.l_quantity
  , lineitem.l_discount
  , lineitem.l_shipinstruct
  , lineitem.l_commitdate
  , lineitem.l_shipmode
  , orders.o_orderdate
  , orders.o_orderpriority
  , orders.o_orderstatus
  , supplier.s_name
  , supplier.s_comment
```

The aggregate table above can be used to answer queries which refer the same set of tables(or more), joined on same condition and refer columns which are projected in aggregated table. Few sample queries which can benefit from the above aggregate table are:

```
SELECT Concat(supplier.s_name,
 orders.o_orderdate) supp_namedate
, lineitem.l_quantity
, lineitem.l_discount
, Sum(lineitem.l_extendedprice) sum_price
, Sum(orders.o_totalprice)      total_price
FROM   lineitem
 JOIN  part
  ON ( lineitem.l_partkey = part.p_partkey )
 JOIN  orders
  ON ( lineitem.l_orderkey = orders.o_orderkey )
 JOIN  supplier
  ON ( lineitem.l_suppkey = supplier.s_suppkey )
WHERE  lineitem.l_quantity BETWEEN 10 AND 150
 AND lineitem.l_shipinstruct <> 'deliver IN person'
 AND lineitem.commitdate BETWEEN '11/01/2014'
          AND '11/30/2014'
 AND lineitem.l_shipmode NOT IN ('AIR', 'air reg')
 AND orders.o_orderpriority IN ('1-URGENT', '2-high')
GROUP BY Concat(supplier.s_name, orders.o_orderdate)
 , lineitem.l_quantity
 , lineitem.l_discount
```

or

```
SELECT lineitem.l_shipmode
 , Sum(orders.o_totalprice)
 , Sum (lineitem.l_extendedprice)
FROM lineitem
 JOIN orders
  ON ( lineitem.l_orderkey = orders.o_orderkey )
 JOIN supplier
  ON ( lineitem.l_suppkey = supplier.s_suppkey )
WHERE ( lineitem.l_quantity BETWEEN 10 AND 150
 AND lineitem.l_shipinstruct <> 'DELIVER IN PERSON'
 AND lineitem.commitdate BETWEEN
 '11/01/2014' AND '11/30/2014'
 AND supplier.s_comment Like '\%customer\%complaints\%'
 AND orders.o_orderstatus ='f'
GROUP BY
  lineitem.l_shipmode
```

Similarly, UPDATE statements in various flavors that modify rows in a table via a direct UPDATE or with the query results from another query, have been supported in relational DBMS offerings for several decades. In some scenarios, consolidating UPDATEs can produce performance benefits. For example combining the following two simple statements:

```
UPDATE customer
 SET   customer.email_id='bob.johnson@edbt.org'
WHERE  customer.firstname='Bob'
 AND   customer.last_name='Johnson'

UPDATE customer
 SET   customer.organization='Engineering'
WHERE  customer.firstname='Bob'
 AND   customer.last_name='Johnson'
```

into a single UPDATE statement as follows:

```
UPDATE customer
 SET   customer.email_id='bob.johnson@edbt.org',
       customer.organization='Engineering'
WHERE  customer.firstname='Bob'
 AND   customer.last_name='Johnson'
```

Such consolidation reduces the number of UPDATE queries on the source table 'customer' and minimizes the I/O on the table.

Existing commercial offerings also support the 'REFRESH' option to propagate changes to aggregate tables whenever the underlying source tables are updated. Generally, this requires a mechanism to UPDATE rows in the aggregate tables. However, the immutable properties of HDFS in Hadoop, which is highly optimized for write-once-read-many data operations, poses problems for implementing the 'REFRESH' option in Hive and Impala. This hampers developing functionality for UPDATE statements - which has largely lead Hadoop-based SQL vendors to shy away from or offer limited support for UPDATE-related features.

Based on our learnings from several customer engagements, some important observations surface:

1. In Hadoop, highly parallelized processing and optimized execution engines on systems such as Hive and Impala enable rebuilding aggregate tables from scratch very quickly, making UPDATEs unnecessary and mitigating the HDFS related immutability issues in many EDW workloads.

2. Many aggregate tables are temporal in nature. For example, quarterly financial reports that require data from only three months, in which case the aggregate tables that feed these reports can be data partitioned on a monthly basis on Hive and Impala. Smaller portions of giant source tables need to be queried to populate these aggregate tables. Only the impacted partitions of the aggregate tables need to be written, making modifications to aggregate tables less expensive. Hence, instead of using UPDATES to modify them, new time-based partitions (by month or day) can be added and older ones discarded. SQL constructs such as INSERT with OVERWRITE supported on Hive and Impala, can be used to mimic this REFRESH functionality. And SQL views can be used to allow easy switching between an older and newer version of the same data.

3. With the introduction of new Hadoop features such as the Apache Kudu integration [12], a viable alternative to using HDFS is now available. Hence UPDATEs can now be supported for certain workloads.

UPDATE statements used to perform tasks such as address cleanup in ETL workloads or modify slowly-changing dimension tables in BI and Analytic workloads are different in nature from highly concurrent OLTP style UPDATEs present in traditional operational systems. In this case, UPDATEs are concentrated on certain tables and are less frequent. If the temporal nature of data mentioned above can be exploited, partitioning techniques to mimic UPDATEs are possible as are some other SQL join-based techniques discussed later in this paper.

Given the importance of these two SQL features, BI Users and Hadoop developers are adopting one of the above mentioned strategies; and require recommendations on which aggregate tables to create, and how to consolidate UPDATE statements, to optimize the performance of their queries on Hadoop. In the following sections, we describe our algorithm for aggregate table creation. And compare the efficiency and quality of the aggregate tables generated when the input to the algorithm is all queries in a workload versus targeted sets of highly similar queries derived from the workload. A clustering algorithm performs advanced analytics over all the queries in a workload, to extract these highly similar query sets. We also discuss techniques and algorithms for consolidation of UPDATE statements, prior to applying them on Hadoop.

2. BACKGROUND AND RELATED WORK

Aggregate table advisors are available in several commercial EDW and BI offerings. In some of these offerings, the onus is on the user to provide a representative workload - i.e. a sample set of queries to use for deriving aggregate tables. Others require query execution plans to provide recommendations. The DB2 Design Advisor in [15] discusses the issue of reducing the size of the sample workload to reduce the search space for aggregate table recommendations, while the Microsoft paper [3] details specific mechanisms to compress SQL workloads. Our approach takes a SQL query log as an input workload (all queries executed over a period of time in a EDW system) and identifies semantically unique queries discarding duplicates. We use the structure of the SQL query when identifying the duplicates which means the changes in the literal values result in identifying these queries as duplicates. Advanced analytics are then deployed on the SQL structures of these semantically unique queries to discover clusters of similar query sets. This enables quicker and more relevant aggregate table definitions because the set of queries that serve as input to aggregate table recommendations are highly similar.

After aggregate tables are set up, some DBMS and BI tools offerings are further capable of rewriting queries internally to use aggregate tables versus the base tables to optimize performance of queries. This feature also known as materialized views is not addressed in our paper. An example Hadoop implementation of materialized views is described in [14]. In our experience, BI tools are frequently used in reporting and analytic workloads deployed atop Hadoop and necessarily support materialized views. Hence we provide recommendations and the DDL definitions for the aggregate tables that users can create, using the BI tools of their choice.

On the Hadoop side, the [5] features revolve around using materialized views to better exploit Hadoop clusters. And in the Hive community, explicit support for materialized views is under development [10]. Again, these efforts are orthogonal to the aggregate table recommendations we provide. [11] seeks to solve the HDFS immutability issue and lift UPDATE restrictions. The techniques we propose in this paper are orthogonal and applicable at the SQL level - and seek to boost performance of SQL queries on Hadoop. Thus they can benefit both HDFS and Kudu-based Hadoop deployments.

3. THE SYSTEM

Our system is a workload-level optimization tool that analyzes SQL queries (from many popular RDBMS vendors) from sources such as query logs. It breaks down the individual SQL constructs in these queries and employs advanced analytics to

- identify semantically unique queries, thus eliminating duplicates
- discover top tables and queries in a workload as shown in Figure 1
- surface popular patterns like joins, filters and other SQL constructs used in the workload.

This analysis is further used to alert users to SQL syntax compatibility issues and other potential risks such as many-table joins that these queries could encounter on Hive or Impala providing recommendations on data model changes and query rewrites that can benefit performance of the queries on Hadoop.

The tool operates directly on SQL queries so does not require access to the underlying data in tables or to the Hadoop clusters that the workload may be deployed on. However, information such as the elapsed time for a query and statistics such as table volumes and number of distinct values (NDV) in columns, help improve the quality of our recommendations.

The recommendations include candidates for partitioning keys, denormalization, inline view materialization, aggregate tables and update consolidation. The last two recommendations are the focus of this paper.

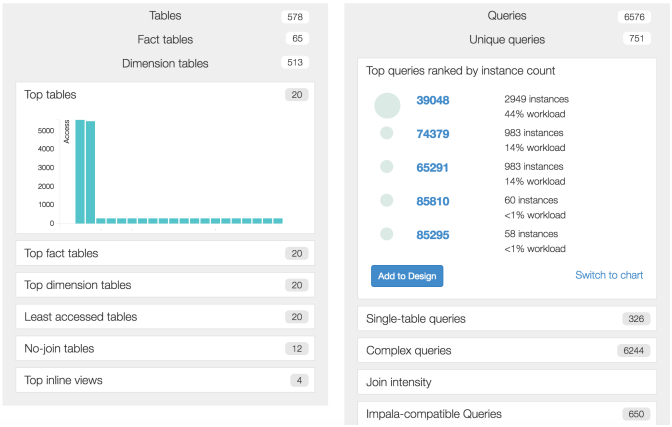

Figure 1: Workload Insights: Popular Queries and Patterns.

3.1 Aggregate Table Recommendation

Our algorithm to determine aggregate tables, is similar to [2] with a few important modifications, which are elaborated in the subsequent sections. The first step in determining the aggregate tables is to find a set of interesting table subsets. A table-subset T is interesting if materializing one or more views on T has the potential to reduce the cost of the workload significantly, i.e., above a given threshold.

In BI workloads, joins over 30 tables in a single query is not an infrequent scenario. Such workload characteristics could incur exponential costs while enumerating all interesting subsets. The enumeration of all interesting subsets of 30 tables is not practical hence we need a mechanism to reduce the overall number of interesting subsets. [1] presents efficient algorithms to enumerate all frequent sets and [4] presents a compact way of representing all the frequent sets. However generating aggregate tables on a subset of tables may be more beneficial than generating it over supersets. Since enumerating all subsets can be exponential, we need to select the subsets which are still a good representation of all the interesting subsets.

3.1.1 Merge and Prune

We address the problem of exponential subsets by constraining the size of the items at every step. During each step in subset formation, we merge some of the subsets early and then prune some of these subsets, without compromising on the quality of the output. We use the notations mentioned in Table 1 to describe the various concepts used in our mergeAndPrune algorithm. The detailed steps are outlined in the algorithm 1. The algorithm takes a set of sets of tables of a given size and returns the new set with some elements merged and removed from the input.

The metric TS-Cost(T) we use is the same as that mentioned in [2] which is the total cost of all queries in the workload where table-subset T occurs. After we enumerate all 2-subsets (subsets of size 2) we execute the algorithm in each step for merging and pruning the sets early. We start with a given element and collect the list of all candidates that it can be merged with. The merges are performed as long as the merged set is within a certain threshold. Experimental results indicated that a value of .85 to 0.95 is a good candidate for this threshold. We maintain a merge list and add the elements from the merge list to the prune list, only if there is no potential for the elements to form further combinations of tables.

Table 1: Notations used in mergeAndPrune

input	Set of sets of a given size formed by tables in the queries.
pruneSet	A subset of *input* that holds the list of elements that will be pruned from the *input* at the end of the iteration.
M	Holds the current table set that is considered for meging.
$MList$	Holds all the sets that can be merged with M.
mergedSets	Set of sets of tables that are formed by merging some elements from *input*.

Algorithm 1 Algorithm for merging and pruning interesting table subsets

function MERGEANDPRUNE
  for each $i \in$ input and $i \notin$ pruneSet **do**
    $M \leftarrow i$
    $MList \leftarrow \{i\}$
    for each $c \in$ input **do**
      if $c \subset M$ **then**
        $MList \leftarrow MList \cup c$
        continue
      end if
      ▷ determine if the merge item is effective and not too far off from the original
      if TS-COST(M $\cup$ c)/TS-COST(M) $>$ MERGE_THRESHOLD **then**
        $M \leftarrow M \cup c$
        $MList \leftarrow MList \cup c$
      end if
    end for
    for each $m \in MList$ **do**
      ▷ retain candidates that we should not be pruning.
      if $\nexists$ s| s $\in$ input and s $\notin MList$ and s $\cap$ m $\neq \phi$ **then**
        ▷ find the candidates for pruning from input in the later step.
        $pruneSet \leftarrow pruneSet \cup m$
      end if
    end for
    $mergedSets \leftarrow mergedSets \cup M$
  end for
  $input \leftarrow input - pruneSet$
  return mergedSet
end function

3.1.2 Aggregate table creation using query clustering

BI reporting workloads and analytical workloads typically generate queries against the same star/snowflake schema, but these queries select different sets of columns and invoke different sets of aggregate functions i.e SUM, COUNT etc. The clustering algorithm compares the similarity of each clause in the SQL query (i.e. SELECT list, FROM, WHERE, GROUPBY, etc.) to pull together highly similar

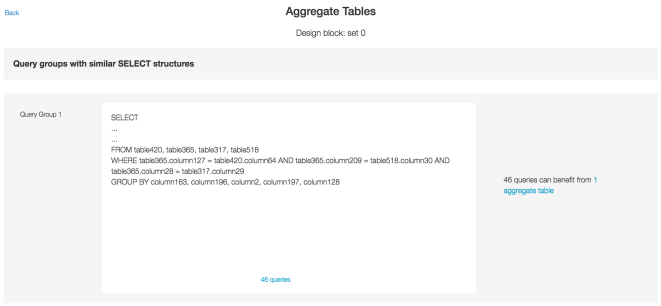

Figure 2: Aggregate Table Candidate Queries

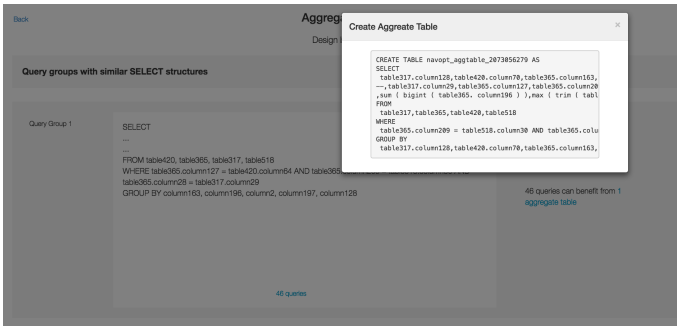

Figure 3: Aggregate Table DDL Generation

queries. Clustering queries in the workload based on the similarity of the SQL query structure collects together queries that access the same or almost similar table sets. Figure 2 and figure 3, depicts our implementation. Figure 2 shows the aggregate query that is beneficial to the given cluster of queries.The number of queries in the cluster are shown on the right side. As shown in figure 3, users can also generate the DDL that creates the specified aggregate table.

3.2 Update Consolidation

Several customers have legacy applications that encapsulate ETL logic in SQL stored procedures and SQL scripting languages (such as Oracle PL/SQL or Teradata BTEQ). Neither Hive nor Impala support stored procedures, so the individual queries in these stored procedures need to be executed on Hive/Impala. This ETL logic many times includes UPDATE queries.

In this section, we focus on such UPDATE queries, specifically the issue of converting a sequence of UPDATE queries in a workflow into a smaller set of UPDATE queries. We call this UPDATE consolidation.

Most UPDATE queries in ETL workflows are not complex and largely following the patterns like:

```
UPDATE employee emp
 SET   salary = salary * 1.1
WHERE  emp.title = 'Engineer';

UPDATE emp
 FROM  employee emp ,
       department dept
 SET   emp.deptid = dept.deptid
WHERE  emp.deptid = dept.deptid
 AND   dept.deptno = 1
 AND   emp.title = 'Engineer'
 AND   emp.status = 'active';
```

Table 2: Notations used in update consolidation

Q_i	ith query in the sequence.
SOURCETABLES(Q_i)	All the tables that the query reads from.
TARGETTABLE(Q_i)	The table that is updated as part of the given INSERT/ UPDATE/ DELETE query.
C_i	consolidation set i containing one or more queries.
READCOLS(e)	Set of all the columns that are read by the given query e. For a consolidated set e, this is the union of all the columns belonging to every query in the set.
WRITECOLS(e)	Set of all the columns that will be written by the given query e. For a consolidated set e, this is the union of all the columns belonging to every query in the set.
TYPE(Q_i)	type of the UPDATE query, 1 if it is a single table UPDATE; and 2 if more than one table is referenced in the query.
TYPE(C)	UPDATE type of all the queries contained in the set. A set only contains queries of same type. Hence its a single value indicating the update type of all queries.
SETEXPREQUAL(Q_i, C)	returns true if the set expression in the UPDATE query Q_i is same as one of the set expression in consolidate set C all other columns except those in set expression are not write conflicted

We classify these UPDATE queries into two categories: Type 1 and Type 2 UPDATEs:

- Type 1 UPDATEs are single table UPDATE queries with an optional WHERE clause.
- Type 2 UPDATEs involve updates to a single table based on querying multiple tables.

This distinction between UPDATE queries is important, because Type 1 and Type 2 UPDATE queries can never be consolidated together. To execute UPDATE queries on Hadoop, the typical process is to use the CREATE-JOIN-RENAME conversion mechanism. The three steps of the CREATE-JOIN-RENAME conversion mechanism are:

1. Create a temporary table by converting the UPDATE query into CREATE+SELECT query, containing the primary key and updated columns of the target table.
2. Create a new table by performing a LEFT OUTER JOIN of the original table with the temporary table.Non

null values in the temporary table get priority over the original table.

3. Drop the original table and RENAME the newly created table to that of the original table.

If these 3 steps are needed to process each UPDATE, executing a sequence of UPDATEs can become very expensive if the steps are repeated for each UPDATE query individually. An efficient way of executing sequential UPDATEs on Hadoop is to first consolidate the UPDATEs into a smaller set of queries. However, it is very important to attempt consolidation only when we can guarantee that the end state of the data in the tables remains exactly the same with both approaches - i.e. when applying one UPDATE at a time versus a consolidated UPDATE. Therefore, the algorithm has to check for interleaved INSERT/UPDATE/DELETE queries, be mindful of transactional boundaries, etc. - and only perform consolidation when it is safe to do so.

Partitioned tables can be updated using the PARTITION OVERWRITE functionality. If the UPDATE statement contains a WHERE clause on the partitioning column, then we can convert the corresponding UPDATE query into an INSERT OVERWRITE query along with the required partition specification. If the query is modifying a selected subset of rows in the partition, we still have to follow the above approach to compute the new rows for the partition, including the modified rows. In this case too, since a join is involved, it is beneficial to look at consolidation options.

Another commonly used workaround to mitigate UPDATE issues is to use database views, i.e. users access data pointed to by a normal table or in the Hadoop context a partitioned table through a view. After UPDATEs to the table are propagated to Hadoop by adding a new partition that contains updated data to the existing table or re-building the entire table that now reflects UPDATEs, the view definition is changed to now point at the newly available data. This way users have access to the 'old' data till the point of the switch. A similar approach, (but for the compaction use case) is discussed here [6]. Even with this mechanism, consolidating updates to a particular partition or table prior to applying the updates, can minimize IO costs.

3.2.1 Update Consolidation Algorithm

We use the notations mentioned in Table 2 to describe the various concepts used in our UPDATE consolidation algorithm.

The *findConsolidatedSets* algorithm to consolidate UPDATE queries is shown in Algorithm 4. The algorithm starts with an empty set and adds the first UPDATE query it finds into the current consolidation set. Then it checks subsequent queries to see if there are any potential conflicts with the group in hand. Query Q_i conflicts with Q_j if Q_j is either reading or writing a table that Q_i writes to. We use the procedure *isReadWriteConfict* in Algorithm 2 to determine the same.The UPDATE queries Q_i and Q_j that are reading from the same set of tables and writing to the same table can conflict if one of the queries is writing to a column, which the other query is reading from. We use the procedure *isColumnConflict* in Algorithm 3 to determine the conflict.

When we encounter a conflicting query, we stop the consolidation process. When two UPDATE queries Q_i and Q_j are in sequence with no conflicting queries in between, they can be considered for consolidation. So we check Q_i and Q_j for compatibility. Q_i and Q_j can be consolidated into one group if all the following conditions are met:

1. Q_i and Q_j are of the same UPDATE types - i.e. either both are Type 1 or both are Type 2.
2. For Type 1 UPDATEs, the target table is the same for Q_i and Q_j and there are no columns of Q_i and Q_j that are write conflicted.
3. For Type 2 UPDATEs, the source and target tables are the same for Q_i and Q_j (along with same join predicate) and there are no columns of Q_i and Q_j that are write conflicted.

Finally, we maintain a visited flag with each UPDATE query so that if there are interleaved UPDATEs between totally different UPDATE queries in the same stored procedure, they can be considered for consolidation.

Once we have identified a group of all consolidated sets, the conversion to the equivalent CREATE-JOIN-RENAME queries follows these steps. Here, without loss of generality, we assume that all WHERE predicates are in Conjunctive Normal Form.

1. We convert each of the
'SET <col> = <colexpression> WHERE <predicates>'
into
'CASE WHEN <predicates> THEN <colexpression>
ELSE <col> END as <col>'
2. For queries with same SET expression and different WHERE predicates, we create an OR clause for each of the WHERE predicates in the CASE block.
3. We take the WHERE predicates of all the queries and combine them using disjunction with the OR operator. If there is a common subexpression among WHERE predicates, we promote the common subexpression outwards.

Algorithm 2 Procedure to detect conflicting queries

function isReadWriteConfict(e_1,e_2)
  if targetTable(e_1) ∩ sourceTables(e_2) = ϕ && targetTable(e_2) ∩ sourceTables(e_1) = ϕ && targetTable(e_2) ∩ targetTable(e_1) = ϕ **then**
    return True
  else
    return False
  end if
end function

Algorithm 3 Procedure to detect conflicting read/write columns

function isColumnConflict(e_1,e_2)
  if writeCols(e_1) ∩ readCols(e_2) = ϕ && writeCols(e_2) ∩ readCols(e_1) = ϕ && writeCols(e_2) ∩ writeCols(e_1) = ϕ **then**
    return True
  else
    return False
  end if
end function

Algorithm 4 Procedure to find and consolidate queries

```
function findConsolidatedSets
    C ← {}                                                          ▷ current consolidated set
    Q ← setOfInputQueries
    while ∃ update query with visited(q) = False do
        for i←1 to |Q| do
            if Q_i ≠ Update Query then                                          ▷ insert or delete query
                if ¬ isReadWriteConflict(C,Q_i) then        ▷ conclude the current consolidated set and start a new set
                    output ← output ∪ C
                end if
                visited(Q_i)← True ; continue
            end if
            if |C| = 0 and Q_i is Update Query and visited(Q_i) = False  then
                visited(Q_i)← True ; continue
            end if
            if  type(Q_i) ≠ type(C) then
                output ← output ∪ C
                if  visited(Q_i) = False then
                    C ← {Q_i}
                else
                    C ← φ
                end if
                visited(Q_i)← True ; continue
            end if
            if  type(Q_i) = 1 and type(C) = 1 then                  ▷ Type 1 : Single table update query
                if  targetTable(Q_i) = targetTable(C) then
                    if  isColumnConflict(C,Q_i) or setExprEqual(Q_i,C) then
                        if  visited(Q_i) = False then
                            C ← C ∪ Q_i
                        end if
                    else
                        output ← output ∪ C
                        if  visited(Q_i) = False then
                            C ← {Q_i}
                        else
                            C ← φ
                        end if
                    end if
                    visited(Q_i)← True ; continue
                end if
            end if
            if  type(Q_i) = 2 and type(C) = 2 then                  ▷ Type 2 : Multi table update query
                if  targetTable(Q_i) = targetTable(C) and sourceTable(Q_i) = sourceTable(C) then
                    if  isColumnConflict(C,Q_i) or setExprEqual(Q_i,C) then
                        if  visited(Q_i) = False then
                            C ← C ∪ Q_i
                        end if
                        visited(Q_i)← True ; continue
                    end if
                end if
                if  ¬ isReadWriteConflict(C,Q_i) then
                    output ← output ∪ C
                    if  visited(Q_i) = False then
                        C ← {Q_i}
                    else
                        C ← φ
                    end if
                    visited(Q_i)← True ; continue
                end if
            end if
        end for
    end while
    return output
end function
```

Here are some examples of consolidations. The following Type 1 UPDATE queries that modify the table 'lineitem' - with or without filtering conditions:

```
UPDATE lineitem
 SET   l_receiptdate = Date_add(l_commitdate, 1)

UPDATE lineitem
 SET   l_shipmode = concat(l_shipmode,'-usps'),
WHERE  l_shipmode = 'MAIL'

UPDATE lineitem
 SET   l_discount = 0.2
WHERE  l_quantity > 20
```

can be consolidated and converted into a CREATE-JOIN-RENAME flow as follows:

```
CREATE table lineitem_tmp AS
SELECT Date_add(l_commitdate, 1) AS l_receiptdate
 , CASE
    WHEN l_shipmode = 'MAIL'
    THEN concat(l_shipmode,'-usps')
    ELSE l_shipmode
   END AS l_shipmode
 , CASE
    WHEN l_quantity > 20
    THEN 0.2
    ELSE l_discount 0
   END AS l_discount
 , l_orderkey
 , l_linenumber
FROM lineitem;

CREATE TABLE lineitem_updated AS
 SELECT orig.l_orderkey
  , orig.l_linenumber
  , Nvl(tmp.l_receiptdate, orig.l_receiptdate)
     AS l_receiptdate
  , Nvl(tmp.l_shipmode, orig.l_shipmode)
     AS l_shipmode
  , Nvl(tmp.l_discount, orig.l_discount)
     AS l_discount
  , l_partkey, l_suppkey, l_quantity, l_extendedprice
  , l_tax, l_returnflag, l_linestatus, l_shipdate
  , l_commitdate, l_shipinstruct, l_comment
FROM  lineitem orig
 LEFT OUTER JOIN lineitem_tmp tmp
 -- lineitem table primary key
  ON ( orig.l_orderkey = tmp.l_orderkey
   AND orig.l_linenumber = tmp.l_linenumber )

DROP TABLE lineitem;

ALTER TABLE lineitem_updated RENAME TO lineitem;
```

As another example consider the following Type 2 UPDATE queries, that modify the 'lineitem' table based on the results of a join with the 'orders' table:

```
UPDATE lineitem
 FROM lineitem l
    , orders o
 SET l.l_tax = 0.1
WHERE l.l_orderkey = o.o_orderkey
 AND o.o_totalprice BETWEEN 0 AND 50000
 AND o.o_orderpriority = '2-HIGH'
 AND o.o_orderstatus = 'F';

UPDATE lineitem
 FROM lineitem l
    , orders o
SET l_shipmode = 'AIR'
WHERE l.l_orderkey = o.o_orderkey
 AND o.o_totalprice BETWEEN 50001 AND 100000
 AND o.o_orderpriority = '2-HIGH'
 AND o.o_orderstatus = 'F';
```

can be consolidated and converted into a CREATE-JOIN-RENAME flow as follows:

```
CREATE TABLE lineitem_tmp AS
 SELECT CASE
     WHEN o.o_totalprice BETWEEN 0 AND 50000
     THEN 0.1 ELSE l_tax END AS l_tax
  , CASE
     WHEN o.o_totalprice BETWEEN 50001 AND 100000
     THEN 'AIR' ELSE l_shipmode
    END AS l_shipmode
  , l_orderkey
  , l_linenumber
 FROM lineitem l
  , orders o
 WHERE l.l_orderkey = o.o_orderkey
  AND o.o_totalprice BETWEEN 0 and 100000
  AND o.o_orderpriority = '2-HIGH'
  AND o.o_orderstatus = 'F';


CREATE TABLE lineitem_updated AS
 SELECT orig.l_shipdate
  , orig.l_commitdate
  , orig.l_receiptdate
  , orig.l_orderkey
  , orig.l_partkey
  , orig.l_suppkey
  , orig.l_linenumber
  , orig.l_extendedprice
  , Nvl(tmp.l_tax, orig.l_tax)  AS l_tax
  , orig.l_returnflag
  , orig.l_linestatus
  , Nvl(tmp.l_shipmode, orig.l_shipmode) AS l_shipmode
  , orig.l_shipinstruct
  , orig.l_discount
  , orig.l_comment
 FROM lineitem orig
  LEFT OUTER JOIN lineitem_tmp tmp
   ON ( orig.l_orderkey = tmp.l_orderkey
    AND orig.l_linenumber = tmp.l_linenumber )

DROP TABLE lineitem;

ALTER TABLE lineitem_updated RENAME TO lineitem;
```

We also looked at the problem of constructing a control flow graph of the stored procedure and performed a static analysis on this graph. If the number of different flows are manageably finite, we can generate a consolidation sequence

for each of the different flows independently thus enabling the user to script these flows independently. However we omit all the implementation details here, as it is beyond the scope of this paper.

4. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of the various recommendations discussed in the previous sections using two different workloads. The first workload is TPC-H at the 100 GB scale, which we call TPCH-100. Our second workload belongs to a customer in the financial sector. This customer has 578 tables with 3038 number of columns. The table sizes vary from 500 GB to 5TB. We call this workload CUST-1.

We have setup representative clusters to measure the performance of the system. This cluster has 21 nodes with 1 master and 20 data nodes. The data nodes are the AWS m3.xlarge kind, with 4 core vCpu, 2.6 GHZ, 15GB of main memory and 2 X 40GB SSD storage. In all the experiments 'time' refers to the wall clock time as reported by the executing Hive query. There are no other queries running on the system. For simplicity, we ignore the HDFS and other OS caches. The experiments presented should be interpreted as directional rather than exhaustive empirical validation.

4.1 Aggregate Table Recommendation

For aggregate table generation we ran our experiments on the CUST-1 setup.

4.1.1 Clustering similar queries

In our first set of experiments we evaluate the quality of aggregate tables generated with and without clustering similar queries together. We divided a workload with 6597 queries into set of clusters using the clustering algorithm, thus reducing the number of queries to a group of smaller workloads. We empirically show how this approach provides aggregate table recommendations with better run time benefits. The first four smaller workloads are comprised of similar queries detected by a clustering algorithm that is run over the 6597 queries. In the fifth workload we bundle all the 6597 queries together. Figure 4 displays how the workloads vary in size from 18 to 6597 queries.

Figure 5 and figure 6 show the results of executing the aggregate table recommendation algorithm on these 5 workloads. As demonstrated in these results, the time taken for the algorithm does not have a direct correlation to the input workload size. The algorithm converges to a solution when it reaches a locally optimum solution. When similar queries are clustered together the chances of the locally optimum solution being globally optimum are high. In our experiments when the algorithm is run on all the queries it converges to a globally sub-optimum solution, recommending an aggregate table that benefits fewer queries - and hence has a lower estimated cost saving. The estimated cost savings for each cluster is computed as the sum of the estimated cost savings for each query in that cluster. The estimated cost of each query is derived by computing the IO scans required for each table and then propagating these up the join ladder to get the final estimated cost of the query. The cost savings is the difference in estimated cost when a query runs on base tables versus the aggregated table.

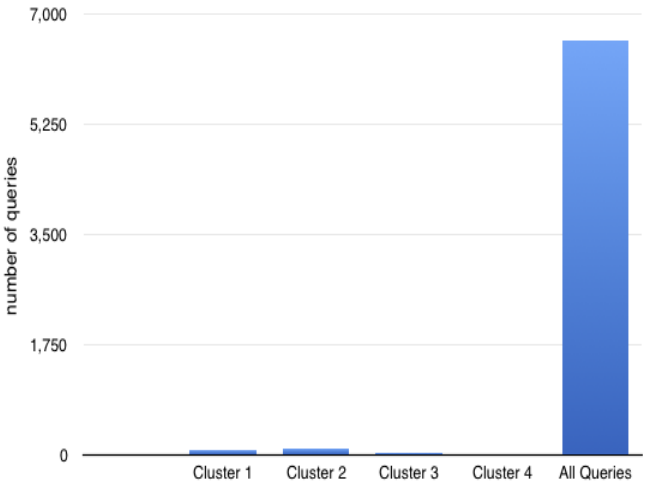

Figure 4: Number of queries per workload.

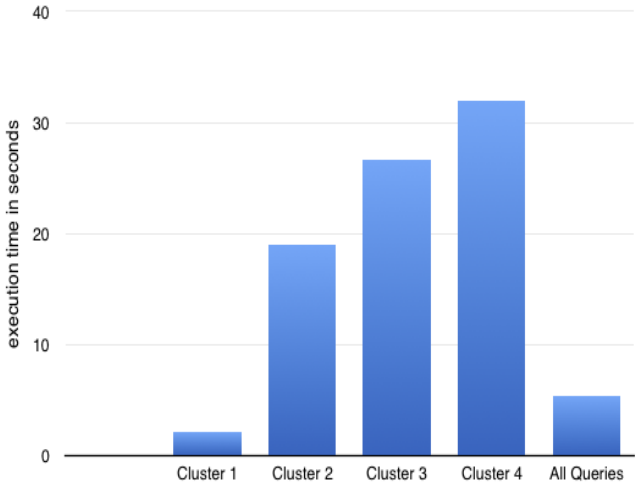

Figure 5: Execution time of aggregate table algorithm.

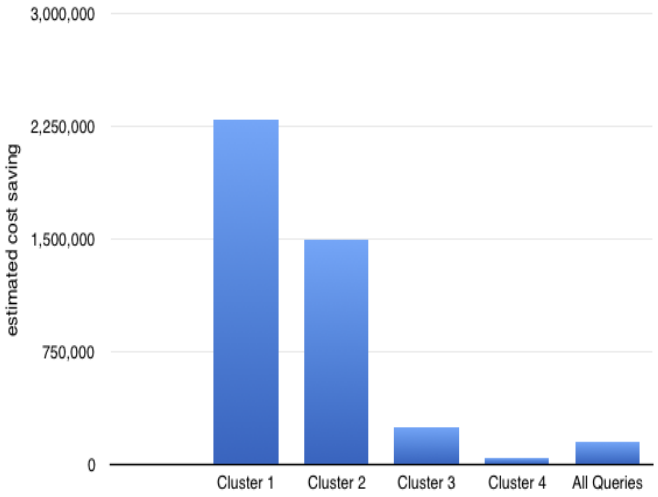

Figure 6: Estimated Cost savings per workload.

Table 3: Merge and Prune

	Execution Time in milli seconds	
Workload Name	With merge and prune	Without merge and prune
Cluster 1	2.092	2.107
Cluster 2	18.919	> 4hrs.
Cluster 3	26.567	> 4hrs.
Cluster 4	31.972	> 4hrs.
Entire Workload	5.279	5.160

4.1.2 *Merge and prune*

In this set of experiments we evaluate the run time of the algorithm with and without the merge and prune enhancement. We run the algorithm on the same workloads we created for the earlier experiment. We terminated the execution of the algorithm after 4 hours. In the case where the algorithm converges to a solution early on, removal of merge and prune has no effect. But in the other cases the algorithm without merge and prune enhancements takes more than 4 hours to complete and so was terminated. When the algorithm ran to completion without merge and prune, we found no change in the definition of the output aggregate table. The results are tabulated in Table 3.

4.2 Update Consolidation

For update consolidation, we ran our experiments on the TPCH-100 setup. We hand-crafted 2 stored procedures atop TPC-H data inspired from a real world customer workload. The number of consolidations we found in the stored procedure are shown in Table 4. Column 2 shows the number of queries in each stored procedure. Column 3 shows the groups of consolidated queries represented by the index of the query in the stored procedure. We see that sometime there are as many as 14 queries that are consolidated into a single group. We also observed that with templatized code generation, there is a lot of scope for consolidating queries.

For comparison purposes, we take the entire stored procedure and convert the queries inside it to equivalent INSERT/UPDATE queries. Any loops in the stored procedures are expanded to evaluate all updated columns - and consider each one for consolidation. Two-way IF/ELSE conditions are simplified to take all the IF logic in one run, and ELSE logic in the other run. N-way IF/ELSE conditions were ignored.

From our results, we observe that on consolidating 5 queries into a single query, the performance impact is not 5x, for the following reasons:

1. The consolidated CREATE query might have more columns than the individual queries, therefore the amount of data it writes will be larger.
2. If there are few or no common subexpressions, then we re-write the whole table or a significantly higher portion of the table.

But our performance results indicate that even with these caveats, UPDATE consolidation is very efficient compared to individually mapped queries. In all our cases, we found that consolidating even two queries is better than individually executing these queries.

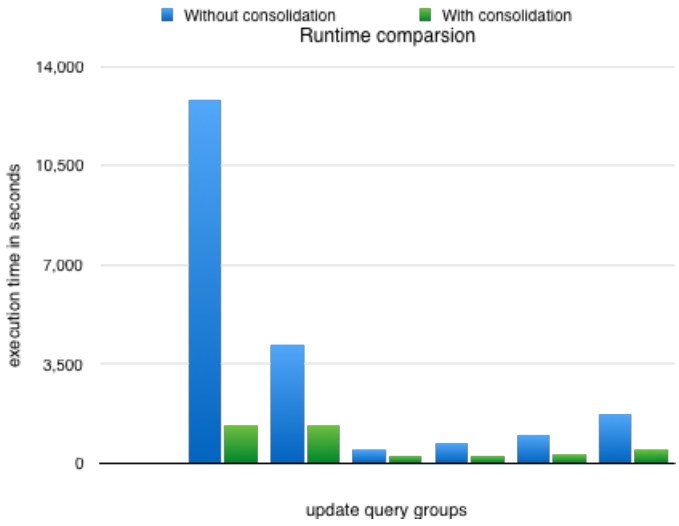

Figure 7: Execution time of consolidated vs non-consolidated queries.

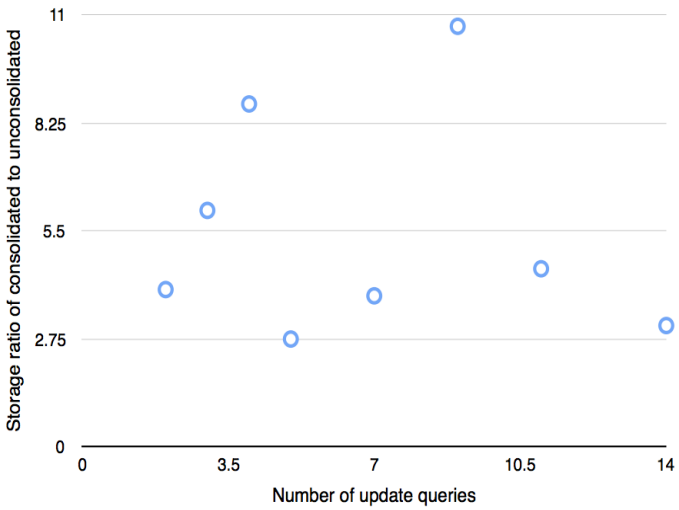

Figure 8: Storage requirements of update queries.

Table 4: Update Consolidation groups

Stored procedure	Number of queries	Consolidation groups
1	38	{6,7,9} ,{10,11} ,{12,14,16,18,20,22,24,26,28} ,{30,32,34,36}
2	219	{113,119,125,131} ,{173,175,177,179,181,183,185,187,189,191,193,195,197,199}

The time taken for detecting UPDATE consolidations is less than a second; hence it was ignored in these measurements. We assume that the database has no triggers, all the tables are independent and updating a table does not incur UPDATEs to tables that are not part of the query. We plot the execution time of non-consolidated queries to consolidated queries in figure 7. The largest group with 14 queries shows a performance improvement of 10x. Even for a group of 2 queries, we see a minimum performance improvement of 80%. The baseline update performance which is spanning few minutes is not an uncommon scenario in SQL-on-Hadoop engines.

The graph in figure 8 shows the storage ratio for consolidated and non-consolidated queries for the size of the consolidation group. If there are multiple groups with the same size, we take the harmonic average of all the groups of the given size. The intermediate storage required for consolidation varies from approximately 2x to as large as 10x when compared to the average storage requirement for individual non-consolidated queries. However in many cases the size of the intermediate table is also significantly less than the original table. In the Hadoop ecosystem, storage is considered a cheap resource and if performance of the UPDATE queries is important, it is certainly worth the trade-off.

These stored procedures are part of a daily workflow that the customer executes, hence the time savings obtained by update consolidation are not only significant but also extremely useful in the big data environment.

5. CONCLUSION & FUTURE WORK

As large scale new and legacy applications are deployed on Hadoop, automated workload-level optimization strategies can greatly help improve the performance of SQL queries. In this paper, we propose creating aggregate tables after first deriving clusters of similar queries from SQL workloads, and demonstrate that in some cases execution time and efficiency improvements of about 1500% can be achieved by using clustered set of queries versus a disparate set of queries as input to the aggregate table creation algorithm. We have also shown empirically that a merge and prune optimization strategy helps the aggregate table creation algorithm converge to a solution, even in cases where it could not converge to a solution. We also showed that 2 to 10X execution time savings can be realized in some cases, when UPDATE queries can be consolidated. Both these optimizations are critical in the Hadoop environment when tables, columns and queries are at very large scales and query response times are of significance.

Advisors [8] [7] [15] [13] [9] rightly emphasize the need for an integrated strategy that evaluates and recommends aggregate table and indexing candidates, together. In the Hadoop ecosystem, partitioning features are the closest logical equivalent to indexes. Currently, if statistical information on a table (such as table volume and column NDVs) is provided, our tool recommends partitioning key candidates for a given table based on the analysis of filter and join patterns most heavily used by queries on the table. We plan to extend this logic to discover partitioning keys for the aggregate tables, thus providing an integrated recommendation strategy.

A further area of focus for the UPDATE consolidation optimization is to explore opportunities to coalesce operations. For example, operations on the temporary table generated in our algorithm can be consolidated to reduce the size of these tables and improve the efficiency of UPDATEs. We are also investigating UPDATE consolidation techniques when UPDATEs are interleaved with control-flow-logic.

Apart from the applicability of our work to adopters of Hive, Impala and Spark SQL who want to optimize their workloads, EDW and BI tools can also use these techniques to improve the efficiency of their workloads.

6. ACKNOWLEDGEMENTS

We would like to thank Prithviraj Pandian for developing the clustering algorithm used in our experiments for Aggregate tables. We would also like to thank our colleagues Justin K, Parna Agarwal, Anupam Singh and Laurel Hale for their insightful review comments.

7. REFERENCES

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

[2] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[3] S. Chaudhuri, A. K. Gupta, and V. Narasayya. Compressing sql workloads. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 488–499, New York, NY, USA, 2002. ACM.

[4] G. Grahne and J. Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE Trans. on Knowl. and Data Eng.*, 17(10):1347–1362, Oct. 2005.

[5] J. Hyde. Discardable memory and materialized queries, May 2014. Available at http://hortonworks.com/blog/dmmq/.

[6] How-to: Ingest and query "fast data" with impala. Available at http://blog.cloudera.com/blog/2015/11/how-to-ingest-and-query-fast-data-with-impala-without-kudu/.

[7] Database engine tuning advisor overview. Available at https://technet.microsoft.com/en-us/library/ms173494(v=sql.105).aspx.

[8] SQL tuning advisor in oracle SQL developer 3.0. Available at http://www.oracle.com/webfolder/

technetwork/tutorials/obe/db/sqldev/r30/TuningAdvisor/TuningAdvisor.htm.

[9] IBM cognos. Available at http://www.ibm.com/support/knowledgecenter/SSEP7J_10.2.2/com.ibm.swg.ba.cognos.cbi.doc/welcome.html.

[10] JIRA:Add materialized views to HIVE. Available at https://issues.apache.org/jira/browse/HIVE-10459.

[11] Apache kudu. Available at http://kudu.apache.org/.

[12] Kudu impala integration. Available at http://kudu.apache.org/docs/kudu_impala_integration.html.

[13] Microstrategy product documentation. Available at https://microstrategyhelp.atlassian.net/wiki/display/MSTRDOCS/MicroStrategy+Product+Documentation.

[14] Qubole quark. Available at http://qubole-quark.readthedocs.io/en/latest/.

[15] D. C. Zilio, J. Rao, S. Lightstone, G. Lohman, A. Storm, C. Garcia-Arellano, and S. Fadden. Db2 design advisor: Integrated automatic physical database design. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pages 1087–1097. VLDB Endowment, 2004.