Optimizing Aggregate Query Processing in Cloud Data Warehouses

Swathi Kurunji*, Tingjian Ge*, Xinwen Fu*,
Benyuan Liu*, Amrith Kumar†, and Cindy X. Chen*

University of Massachusetts Lowell, MA, USA
*{skurunji,ge,xinwenfu,bliu,cchen}@cs.uml.edu
† amrith@parelastic.com

Abstract. In this paper, we study and optimize the aggregate query processing in a highly distributed Cloud Data Warehouse, where each database stores a subset of relational data in a star-schema. Existing aggregate query processing algorithms focus on optimizing various query operations but give less importance to communication cost overhead (Two-phase algorithm). However, in cloud architectures, the communication cost overhead is an important factor in query processing. Thus, we consider communication overhead to improve the distributed query processing in such cloud data warehouses. We then design query-processing algorithms by analyzing aggregate operation and eliminating most of the sort and group-by operations with the help of integrity constraints and our proposed storage structures, PK-map and Tuple-index-map. Extensive experiments on PlanetLab cloud machines validate the effectiveness of our proposed framework in improving the response time, reducing node-to-node interdependency, minimizing communication overhead, and reducing database table access required for aggregate query.

Keywords: Aggregate Operation, Communication Cost, Read-Optimized Database, Data Warehouse, Cloud Storage, Query Optimization

1 Introduction

Data Warehouses or decision support systems use join, group-by, and aggregate operations very often in formulating analytical queries. One of the survey conducted by Oracle [15] shows that, 36% of Data Warehouse users are having performance problems. Common performance bottlenecks include loading large data volumes into a data warehouse, poor metadata scalability, running reports that involve complex table joins and aggregation, increase in the complexity of data (dimensions), and presenting time-sensitive data to business managers etc.

Efficient evaluation of complex queries (i.e. aggregate and multi-join queries) is an important issue in applications that manage and analyze multidimensional data (analytical business data, scientific data, spatial data etc.). Efficient execution of such queries in large-scale and dynamic cloud databases is a challenging problem [7]. One of the main reasons is that we need to update global indexes (such as DHTs) every time the data is moved (or changed), or new machine is added. For example, in a ring network like Cassandra architecture, new machines are added to the ring near the bottlenecked machine, and the data is redistributed between that machine and its new neighbor.

One of the important properties of cloud architecture is elastic scalability. It must therefore support scale-out, where the responsibility of query processing (and the corresponding data) is distributed among multiple nodes to achieve higher throughput. Such network needs good storage structures, which reduce the dependency of data distribution on other machines of the cluster [3].

Our proposed PK-map and Tuple-index-maps are fully decentralized, where no predefined limits are imposed on the sizes of the network or data distribution. We only store information on relationship between the data and not their locations. It is independent of the scale-out of data or the remote data distribution. So, even if data is relocated remotely anywhere in the system, we do not have to update our structures. In our previous work [17], we have showed how our proposed method decreases the interdependency of machines (containing related data) while optimizing the join operation in the query processing. In this paper we will show how we optimize the aggregate query with join operations.

Aggregate operation is one of the expensive operations in distributed query processing due to the requirement of sort and group-by operations to find the aggregated result. Most of the earlier research work considers communication cost as cheaper of all other operations of the query. [1] Hence two-way optimization algorithm is most commonly used, where query optimizer first generates the plan assuming that the query is processed in local machine and, then optimizes the plan considering the distributed architecture of query execution.

Due to virtualization nature of a cloud environment, data storage and query processing have become physically more distributed to meet the resource availability or the customers service agreement [9]. This gives rise to increase in node-to-node communication. In such scenario, even if the overall response time of distributed query is decreased, it is possible that communication cost exceeds other query operator costs. So, it is better to give more consideration to both query operators as well as communication cost while generating the plan.

In this paper, we use our map structures and integrity constraint inference to push down or pull up group-by functions while generating an aggregate query plan. During query processing, we eliminate most of the sort and group-by operations by having sorted map structures and enforcing a certain order based on the hierarchy of the tables in the schema (reference graph of [17]). We will show how we only access more relevant data from the database tables.

The remainder of this paper is organized as follows: Section 2 provides a literature review on aggregate query processing and optimization. Section 3 explains our proposed framework. Section 4 shows the performance evaluation using PlanetLab Cloud. Finally, Section 5 states the conclusion.

[1] Communication cost includes costs per message, costs to transfer data and CPU costs to pack, unpack, and process messages at the sending and receiving sites.

2 Related Work

Aggregate query processing has been studied in many research works [5]. But, as per our knowledge, not many of them consider communication cost in optimizing aggregate query processing. We analyzed some of the works which optimize the aggregate query operations. Along with that knowledge, we propose our storage structures, which will not only optimize query operations, but also communication cost overhead caused in cloud data warehouses.

Some of the earlier papers, which optimize aggregate query processing, are [2] [14] and [22]. These papers provide optimizations by pushing down group-by in the query tree to improve the query response time. W.Yan [22] proposed two kinds of transformations namely, eager aggregation and lazy aggregation. In eager aggregation, group-by operation is pushed down in the query tree, while in lazy aggregation group-by is pushed up. We use the above transformations of [22] in our system along with our PK-map and Tuple-index-map to generate optimized query plan to process aggregate queries.

Order-Optimization [4], presents techniques to reduce the number of sorts needed for query processing by finding the cover set using keys, predicates and indexes. Since our proposed map structures are already sorted on keys, we eliminate most of the sort operations required for join operation on the tables.

Coloring-Away [23], proposed query plan generation using tree-coloring mechanism. This paper considers both communication cost and data re-partitioning, and uses tree coloring to generate optimal query plan. In our framework, we optimize the query operations that cause the above mentioned query performance problems such as aggregates and joins by doing sort and group-by on the fly.

Avoid-Sort-Groupby [24], proposed a query plan refining algorithm through which unnecessary sorting and grouping can be eliminated from the query plan. It uses inference strategies and order properties of the relation table to find the unnecessary sorting or grouping. T.Neumann[19], points out that it is necessary to consider both ordering and grouping to generate the query plan.

Cooperative-Sort [25], presented an evaluation technique for sorting tables. This technique is for those queries that need multiple sort orders of the same table on different attributes. This minimizes the I/O operations of successive sort operations, which reduce the overall query cost.

Pre-computing the aggregates is proposed by many other researchers [6] [16] [26], which are useful for decision support systems. Decision support systems store huge amount of historical data for analysis and decision-making. These databases are updated less frequently (once a hour/day) on batches. This made it easy to compute the aggregation ahead of time and store it as data cubes or materialized views. Recently, the interval between historic and current data has been reduced a lot. This will make it complicated and time consuming to re-compute data cubes or materialized views every time data gets updated. Recent research by companies like HP, Oracle and Teradata [8] [18] [21] shows new parallelization schemes for processing joins and aggregate operations, eliminating data cubes. So, in this paper we concentrate on optimizing aggregate queries without pre-computation.

3 Our Approach

During query execution, a query accesses data from different tables by performing join operation on primary and foreign key attributes of those tables, and then the required attribute values are filtered. This becomes more complicated in the virtual cloud environment, where the location of the data changes more often based on the resource availability or SLA (Service Level Agreement) of the customer. Increased node-to-node communication results in increased network delay and processing time, leading to the decrease in query performance. Hence, we need storage structures that help to minimize the communication between machines, minimize the maintenance during updates and minimize the increase in the storage space.

3.1 Storage Structure

PK-map Structure: We create a PK-map (i.e., Primary Key map) for each of the primary keys in the tables of the star schema. A PK-map will have one column for the primary key and one column for each of the foreign keys referencing this primary key as shown in Table 1. Primary key column contains primary key values and foreign key column contains logical record-ids. These logical record-ids are index to the Tuple-index-map of the foreign key. Tuple-index-map is explained in the next subsection. The PK-maps are sorted on primary key values, which allow us to apply run length encoding on foreign key logical record-ids and reduce the size of the map to a great extent. Thus, the overall size of the map will be proportional to the number of records in the dimension table, which are usually smaller in size than fact table.

Tuple-index-map Structure: We create a Tuple-index-map for each of the foreign key attribute, which reference the primary key attribute of the dimension table as shown in Table 2. This Tuple-index-map will store the mapping between the logical and actual record-id of the foreign keys in the foreign key table.

Table 1: NationKey-map and SuppKey-map

	$N.nationkey$	$S.nk\_index$	$C.nk\_index$
Row 0	0000000000	0	0
Row 1	0000000001	4	6
Row 2	0000000002	14	26
..	....	....	....
..	....	....	....
Row 24	0000000024	....	....

	$S.suppkey$	$PS.sk\_index$	
Row 0	0000000000	0	
....	....	....	Partition 1
Row 20000	0000020000	100000	
Row 20001	0000020001	....	Partition 2
....	....	....	
....	....	....	....
....	....	....	Partition 49
Row 980001	0000980001	....	
....	....	....	Partition 50
Row 1000000	0001000000	....	

Table 2: SuppKey Tuple-index-map

$PS.sk\_index$	$PS.sk\_t\_index$	
Row 0	0	
......	24	Partition 1
......	.....	
Row 400000	.....	
Row 400001	.....	Partition 2
......	.....	.....
......	.....	.....
......	.....	.....
......	.....	.....
......	.....	.....
......	.....	.....
......	.....	Partition 49
Row 19600000	.....	
......	.....	
......	.....	Partition 50
......	.....	
Row 20000000	.....	

If the foreign key table is sorted on the foreign key attribute value, then we do not require Tuple-index-map for that relationship. In that case, logical record-id of the PK-map will be the actual record-id of the foreign key table. For example, in Figure 1, PART and PARTSUPP tables are sorted on partkey attribute. Hence, PartKey-map does not require Tuple-index-map. Similarly, in column-oriented databases, if at least one projection of the table is sorted on foreign key attribute, then the logical record-id of the PK-map will be the actual foreign key record-id.

3.2 TPC-H Star Schema

Star and Snowflake Schema representations are commonly used in read-optimized Data Warehouses. In the rest of this paper we use the star schema from TPC BENCHMARK H Standard Specification Revision 2.15.0 (Figure 1) for analysis and performance study. Figure 1 is the schema of an industry, which must manage, sell and distribute its products worldwide.

With the TPC-H schema of Figure 1, we need to create 6 PK-maps NationKey-map, SuppKey-map, PartKey-map, PartsuppKey-map, CustKey-map and OrderKeymap. Structure of NationKey-map and SuppKey-map are shown in Table 1. We also need 5 Tuple-index-maps (like Table 2) for relationships between each of the following tables: nation $\leftrightarrow$ supplier, nation $\leftrightarrow$ customer, supplier $\leftrightarrow$ partsupp, customer $\leftrightarrow$ orders table and partsupp $\leftrightarrow$ supplier.

Through performance study of our previous work [17], we know that the size of the maps would usually be around 10% to 12% of the actual data size. Size can be calculated using below formulas.

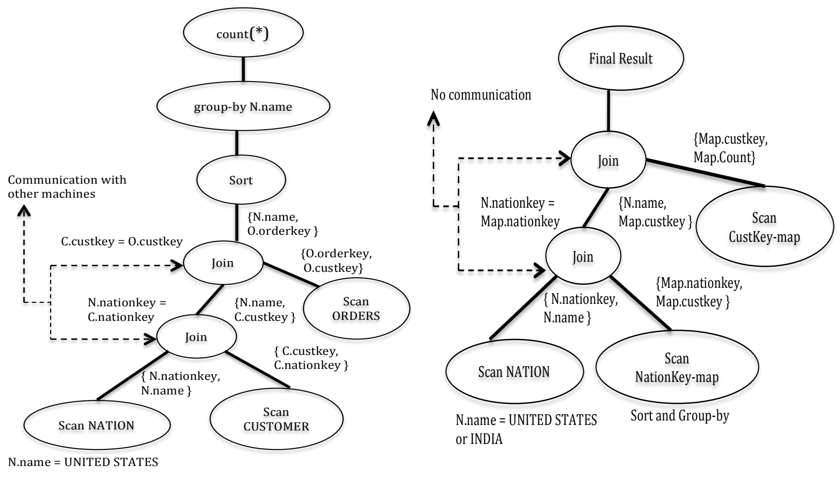
$$Size\ of\ PK\ map = S_1 + \sum_{i=0}^{n} S_2[i] + c \tag{1}$$

$$Size\ of\ Tuple\_index\_map = (Number\ of\ rows\ in\ FK\ Table)\ *\ S_2 \tag{2}$$

where, S_1 is the size of the primary key and S_2 is the size of logical record id (32/64 bit depending on the type of processor)

REGION(regionkey, name, comment)
NATION(nationkey, name, regionkey, comment)
SUPPLIER(suppkey, name, address, nationkey, phone, acctbal, comment)
CUSTOMER(custkey, name, address, nationkey, phone, acctbal, mktsegment, comment)
PART(partkey, name, mfgr, brand, type, size, container, retailprice, comment)
PARTSUPP(partkey, suppkey, availqty, supplycost, comment)
ORDERS(orderkey, custkey, orderstatus, totalprice, orderdate, order-priority, clerk, ship-priority, comment)
LINEITEM(orderkey, partkey, suppkey, linenumber, quantity, extendedprice, discount, tax, returnflag, linestatus, shipdate, commitdate, receiptdate, shipinstruct, shipmode, comment)

Fig. 1: TPC-H Benchmark Schema [20]



(a) MySQL Plan (b) PK-map Plan

Fig. 2: Query Processing Plan for Query1 of Table 3

3.3 Aggregate Query Processing

Inferring on referential integrity constraints and functional dependencies, we have classified aggregate operations into two general categories based on the type of attribute on which the aggregate operation is applied. We then generate plan for query processing accordingly. [2]

Aggregate on Primary or Foreign Key: When there is an aggregate operation on primary key (PK) or foreign key (FK), we do not scan the database table (unless there is a filtering constraint on non-PK or non-FK), instead we scan our proposed map structures and perform aggregate operation on the fly. We can do this because, our maps are sorted on keys, and has required information on number of tuples of foreign key table that are mapped to primary key of dimension table.

For example, Query 1 in Table 3 has count(*) operation. This query is trying to find the total number of orders placed by the specified nations. Here, we only require the number of rows in orders table that belong to each group in the group-by clause (i.e., N.name). By inference, we can say that, this is an aggregate operation on PK of the orders table. Hence, we can eliminate the scan of orders table and get this count by scanning CustKey-map. We push down group-by on N.name to step "scan NationKey-map" as shown in Figure 2b.

Figure 2a is the plan generated by MySQL for processing Query 1 of Table 3. As shown in Figure 2a, it requires three tables scan (Nation, Customer and Orders), two joins, a sort and a group by operation. Here, query processor needs to communicate the data for each join operation, which increases the communication overhead and response time.

[2] We do not consider "having" clause in our analysis, because having clause can be converted into "where" clause constraints by rewriting the query [22].

```
select N.name, SUM(O.totalprice)
from ORDERS O, CUSTOMER C, NATION N
where N.name = "UNITED STATES"
        or N.name = "INDIA"
        and N.nationkey = C.nationkey
        and C.custkey = O.custkey
group by N.name;
```

Fig. 3: Modified Query 1 of Table 3

On the other hand, Figure 2b is the plan generated for the same Query 1 using our framework. Here, instead of scanning Customer and Orders table, we scan two PK-maps (NationKey-map and CustKey-map), and its associated Tuple-index-maps. These maps are very small compared to scanning tables. The join operation in Figure 2b is not equivalent to join operation in Figure 2a. Instead, it is performed by scanning map and applying associated filtering constraint. Algorithms are shown in Figure 4a - 4b and comparison results in Section 4.

Aggregate on Non-Primary or Non-Foreign Key: When there is an aggregate operation on Non-Primary Key (NPK) or Non-Foreign Key (NFK) of the table, then we need to scan the table to find the correct result. But, we use inference to push down the group-by in the query tree so that we can scan only the required portion of table. We also replace the scanning table with maps wherever possible.

For example, suppose if we have SUM (O.totalprice) instead of COUNT (*) of Query 1 as shown in Figure 3. We push down the group-by N.name (as in Figure 2b) to reduce the number of rows to be scanned from the orders table. This reduces the input rows of final aggregate operation.

Group-by Optimization: When there are multiple attributes in the group-by clause, we associate certain order to those attributes and push down group-by in the query tree. As we know that the group-by is a set operation and join result will carry the sort order, we can change the sequence of attributes in group-by clause. So, we change the sequence of attributes in group-by clause corresponding to the sequence of table processing in our algorithm. By doing this we can eliminate non-relevant data in the early stage of query processing as well as achieve same result as we perform group-by in the final stage.

Query Processing Using Proposed Method: In Algorithm 1 shown in Figure 4, we first sort the tables referenced in the query according to their relationship in the schema. We sort tables starting from the table that does not have any foreign keys (depth 0) [17]. For example, Query 1's order of processing will be nation $\rightarrow$ customer $\rightarrow$ orders. In addition, we consider all the constraints in *where* clause and *group-by* while deciding on tables processing order.

We then consider aggregate operation to decide whether to scan the table from the database or to scan our map structure (Line 3-15). If there are PK

Algorithm 1 Aggregate Query Processing Algorithm

```
Input: query Q, reference graph G [17]
Output: Result of query Q
 1: Let T be an array of tables referenced in Query
 2:    Sort T based on d value of G and filtering constraints
          //d value is shown in reference graph of [17]
 3: for each table t ε T do
 4:     Perform predicate/join processing using Algorithm 2
 5:     if there is an aggregate on PK or FK then
 6:         Scan PK-map
 7:         //sort and group-by is inferred
 8:         Update the current result set
 9:     else if there is an aggregate on NPK or NFK then
10:         Scan t and apply aggregate operation
11:          //sort and group-by is applied if necessary
12:        Update the current result set
13:        Communicate if necessary with other machines
14:    end if
15: end for
16: Scan tables of T if necessary to get the value of other attributes
     in the select statement of Q
17: return Result
```

Algorithm 2 Predicate/Join Processing Algorithm

```
Input: Table t ∈ T, predicates of Q, required attributes in result
Output: Result of t
 1:   if there is a predicate on non-PK/non-FK then
 2:       if d == 0 for t then
 3:             Apply predicate on t to get the record ids
 4:           Store the record-id mapping in the format
 5:           (rec-id_1, rec-id_2,....)
 6:           Communicate if necessary with other nodes
 7:       else if any table t_1 with d_1 <= d referenced by t then
 8:          Apply predicate on t
 9:          Update the mapping with rec-ids of t
10:           Perform line 9
11:           Eliminate mappings which has no match for t
12:       else
13:           Perform similar to line 6, 9 and 14
14:       end if
15:   else if there is a predicate on PK or FK then
16:       if d == 0 for t then
17:           Scan PK-map and tuple-index-map
18:           Perform line 6 to 8
19:       else
20:           Scan PK-map and tuple-index-map for those rec-ids stored
                 for table t_1 with d_1 <= d that is referenced by t
21:           Perform 12 and 14
22:       end if
23:   end if
24: Scan tables of T for final mappings (rec-id_1,.......) to get the value of
        other attributes in the select statement of Q
25: return Result
```

(a) Aggregate Query Processing Algorithm (b) Join Processing Algorithm [17]

Fig. 4: Query Processing Algorithms

or FK constraints, we scan our maps. As all PK to FK mapping information is available in maps, data movement between nodes is eliminated in lines 5-8. Also, our maps are already sorted on keys which further eliminates most of the sort operations. At last we retrieve remaining attributes required for the result.

4 Performance Evaluation

In this section, we present the performance study to show the effectiveness of our proposed PK-map and Tuple-index-map structures while processing aggregate queries (using Algorithms in Figure 4a and Figure 4b). We will compare the performance between MySQL and our proposed framework on a large-scale cloud network called PlanetLab with 150GB of TPC-H star schema data.

PlanetLab [12] [13] is a geographically distributed computing platform available as a testbed for deploying, evaluating, and accessing planetary-scale network services. It is currently composed of around 1050 nodes (servers) at 400 sites (location) worldwide.

For performance study of this paper we chose 50 PlanetLab machines worldwide running Red Hat 4.1 Operating System. Each machine has 2.33GHz Intel Core 2 Duo processor, 4GB RAM and 10GB disk space. We installed regular MySQL on all of the machines to perform experiments.

We generated 150GB of data using the data generator "dbgen", provided by TPC-H benchmark and distributed it to 50 PlanetLab machines. Each of these machines store around 3GB data fragments of TPC-H schema relations. We generated PK-maps and Tuple-index-maps, and then horizontally partitioned them using the same partition key that was used to partition the data. We then distributed these partitions into all 50 machines that contained corresponding data. All the map structures are loaded into the main memory before each experiment begins. Here we can take advantage of main memory databases if the data size is huge (in Petabyte or Exabyte) [1] [10] [11].

We used 5 queries for the performance analysis as shown in Table 3. Comparison of time taken by these queries is shown in Figure 5a to 5e. In graphs of these figures, PlanetLab machines are on the x-axis and time taken by the query (in seconds) is on the y-axis.

On each graph, top line shows the result of processing aggregate queries on MySQL, and the bottom line shows the result of our framework. For simplicity of explanation, we do not consider pipelined approach to count the number of communications between the machines. Instead we use inter query operation communications.

Figure 5a compares the time taken for processing Query 1. As explained in Section 3.3.1, Query 1 has an aggregate operation on primary key of table ORDERS, and a Non-Primary Key/Non-Foreign Key (NPK/NFK) constraint on table NATION. Hence, it is sufficient to scan NATION table, NationKey-map, and CustomerKey-map to answer this query.

Figure 5b compares the time taken for processing Query 2. In Query 2, we have an aggregate operation COUNT(*), and a NPK constraint on attribute O.orderstatus of table ORDERS. Hence, we need to scan ORDERS table along

Table 3: Performance Evaluation Queries

Query 1*: (COUNT) Find the total number of orders placed by the 'UNITED STATES' and 'INDIA' customers (aggregation on primary key attribute)*

```
select N.name, count(*)
from ORDERS O, CUSTOMER C, NATION N
where N.name = "USA" or N.name = "INDIA" and
        N.nationkey = C.nationkey and C.custkey = O.custkey
group by N.name;
```

Query2*: (COUNT) Find the total number of failed orders placed by customers of each nation. I.e, orders whose ORDERSTATUS = 'F' (aggregation on NON - Primary Key attribute)*

```
select N.name, count(*)
from ORDERS O, CUSTOMER C, NATION N
where N.nationkey = C.nationkey and C.custkey = O.custkey and O.orderstatus = 'F'
group by N.name;
```

Query 3*: (MIN) Find the minimum supply cost for all the parts supplied by 'UNITED STATES' suppliers*

```
select PS.partkey, min(PS.supplycost)
from PARTSUPP PS, SUPPLIER S, NATION N
where N.name = "USA" and N.nationkey = S.nationkey and S.suppkey = PS.suppkey
group by PS.partkey;
```

Query 4*: (SUM) Find the revenue generated by customers of each nation in the year 1995. I.e., revenue is equal to the totalprice from the orders table*

```
select N.name, sum(O.totalprice)
from ORDERS O, CUSTOMER C, NATION N
where N.nationkey = C.nationkey and C.custkey = O.custkey and O.orderdate like '1995%'
group by N.name;
```

Query5*: (AVG) Find the average revenue generated by customers of each nation in year 1995. I.e., revenue is equal to the totalprice from the orders table*

```
select N.name, avg(O.totalprice)
from ORDERS O, CUSTOMER C, NATION N
where N.nationkey = C.nationkey and C.custkey = O.custkey and O.orderdate like '1995%'
group by N.name;
```

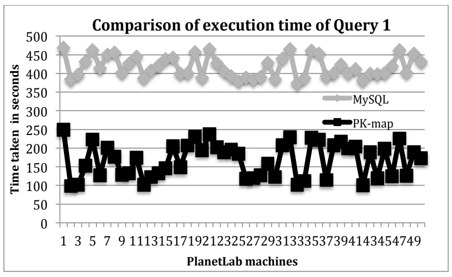
with NationKey-map and CustomerKey-map to process this query.

Figure 5c compares the time taken for processing Query 3. To answer this query, we need to scan PARTSUPP table, NATION table, NationKey-map, and SuppKey-map.

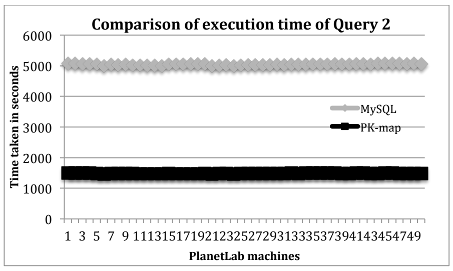
Figure 5d compares the time taken for processing Query 4. This query has an aggregate operation and a filtering constraint on NPK of table ORDERS. Hence, we scan NationKey-map, CustKey-map, and ORDERS table to process the query.

Figure 5e compares the time taken for processing Query 5. This query is similar to Query 4, but needs to keep track of "count" in order to find the "AVG".

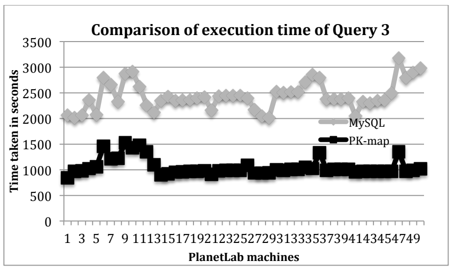
In the above experiments, MySQL takes 3 inter-machine communications, while our framework takes only one communication. In MySQL, each node has



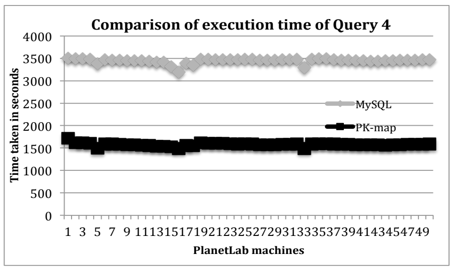
(a) Query 1



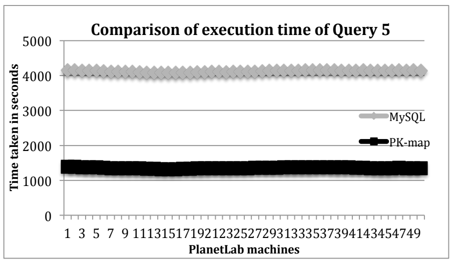
(b) Query 2



(c) Query 3



(d) Query 4



(e) Query 5

Fig. 5: Performance Comparison of Query 1 to Query 5 of Table 3

to communicate partial result or the join attribute values with its peers for every join predicate present in the query. Thus each node has to do additional message processing. This will effect the overall time taken by the query. But, with our map structures we do not communicate for every join operation reducing the communication delay. This less interaction between the database machines in the cluster reduces the communication overhead and effect of poor load balancing. Based on the data in Figure 5a to 5e, it is clear that, the query execution time required by the state-of-the-art MySQL is more than our framework.

5 Conclusion

In this paper, we studied distributed aggregate query processing on cloud data warehouses. We proposed storage structures PK-map and tuple-index-map, and designed query-processing algorithm for processing these aggregate queries. We analyzed the query processing, and advantages of using maps in eliminating sort and group-by operations required by the aggregate operation. Our framework not only reduces the communication cost but also makes minimum access to relation tables depending on the type of aggregate operation. Results of our extensive performance study demonstrate that the proposed approach improves the performance of the ad-hoc aggregate query in Cloud Data Warehouses and reduces communication overhead.

References

1. A., Kemper, T., Neumann: Hyper: A hybrid OLTP and OLAP main memory database system based on virtual memory snapshots. In: IEEE 27th International Conference on Data Engineering (ICDE), pp. 195–206. Hannover, Germany (2011)
2. A., Vlachou, C., Doulkeridis, K., Norvag, Y., Kotidis: Peer-to-Peer Query Processing over Multidimensional Data. Springer (2012)
3. C., Curino, Evan, P., C., Jones, Raluca, A., Popa, N., Malviya, E., Wu, S., Madden, H., Balakrishnan, N., Zeldovich: Relational Cloud: A Database Service for the cloud. In: 5th Biennial Conference on Innovative Data Systems Research (CIDR), pp. 235–240. California, USA (2011)
4. D., Simmen, E., Shekita, T., Malkemus: Fundamental Techniques for Order Optimization. In: ACM SIGMOD international conference on Management of data, vol. 25, pages 57–67. Montreal, Canada (1996)
5. D., Kossmann: The state of the art in distributed query processing. ACM Computing Surveys (CSUR), vol. 32(4), pp. 422–469. December (2000)
6. D., Xin, J., Han, X., Li, Benjamin, W., Wah: Star-Cubing: Computing Iceberg Cubes by Top-Down and Bottom-Up Integration. In: 29th International Conference on Very Large Data Bases (VLDB), vol. 29, pp. 476–487. Berlin, Germany (2003)
7. D., Boukhelef, H., Kitagawa: Efficient Management of Multidimensional Data in Structured Peer-to-Peer Overlays. In: 35th International Conference on Very Large Data Bases (VLDB), vol. 35. Lyon, France (2009)
8. G. Graefe: New algorithms for join and grouping operations. In Journal Computer Science - Research and Development, vol. 27(1), pp. 3–27. February (2012)

9. G., Soundararajan, D., Lupei, S., Ghanbari, Adrian, D., Popescu, J., Chen, C., Amza: Dynamic Resource Allocation for Database Servers Running on Virtual Storage. In: 7th USENIX Conference on File and Storage Technologies (FAST), pp. 71–84. San Francisco, California, USA (2009)
10. H., Garcia-Molina, K., Salem: Main Memory Database Systems. IEEE Transactions on Knowledge and Data Engineering (TKDE), vol. 4(6), pp. 509–516. December (1992)
11. H., Plattner: A common database approach for OLTP and OLAP using an in-memory column database. In: ACM SIGMOD International Conference on Management of data, pp. 1-2. Providence, USA (2009)
12. L., Peterson, T., Roscoe: The design principles of PlanetLab. In: ACM SIGOPS Operating Systems Review, pp. 11–16. New York, USA (2006)
13. Planetlab Cloud, `http://www.planet-lab.org/`
14. S., Chaudhuri, K., Shim: Including Group-By in Query Optimization. In: 20th International Conference on Very Large Data Bases (VLDB), pp. 354–366. Santiago, Chile (1994)
15. Survey, `http://www.oracle.com/us/products/database/high-performance-data-warehousing-1869944.pdf`
16. S., Agarwal, R., Agrawal, Prasad, M., Deshpande, A., Gupta, Jeffrey, F., Naughton, R., Ramakrishnan, S., Sarawagi: On the Computation of Multidimentional Aggregates. In: 22nd International Conference on Very Large Data Bases (VLDB), vol. 22, pp. 506–521. Mumbai(Bombay), India (1996)
17. S., Kurunji, T., Ge, B., Liu, Cindy X., Chen: Communication Cost Optimization for Cloud Data Warehouse Queries. In: 4th IEEE International Conference on Cloud Computing Technology and Science Proceedings (CloudCom), pp. 512–519. Taipei, Taiwan (2012)
18. Srikanth, B., H., Li, Unmesh J., Y., Zhu, Vince, L., Thierry C.: Adaptive and Big Data Scale Parallel Execution in Oracle. International Journal on Very Large Data Bases (VLDB), vol. 6(11), pp. 1102–1113. August (2013)
19. T., Neumann, G., Moerkotte: A Combined Framework for Grouping and Order Optimization. In: 30th International Conference on Very Large Data Bases (VLDB), volume 30, pp. 960–971. Toronto, Canada (2004)
20. TPC-H benchmark, `http://www.tpc.org/tpch/spec/tpch2.14.4.pdf`
21. Teradata, `http://www.teradata.com/white-papers/Teradata-Aggregate-Designer-eb6110`
22. Weipeng, P., Yan, Per-Ake, Larson: Eager Aggregation and Lazy Aggregation. In: 21st International Conference on Very Large Data Bases (VLDB), pp. 345–357. Zurich, Switzerland (1995)
23. W., Hasan, R., Motwani: Coloring Away Communication in Parallel Query Optimization. In: 21st International Conference on Very Large Data Bases (VLDB), pp. 239–250. Zurich, Switzerland (1995)
24. X., Wang, M., Cherniack: Avoiding Sorting and Grouping in Processing Queries. In: 29th International Conference on Very Large Data Bases (VLDB), vol. 29, pp. 826–837. Berlin, Germany (2003)
25. Y., Cao, R., Bramandia, C.-Y., Chan, K.-L., Tan: Sort-Sharing-Aware Query Processing. International Journal on Very Large Data Bases (VLDB), vol. 21(3), pp. 411–436. June (2012)
26. Y., Lin, D., Agrawal, C. Chen: Llama- Leveraging Columnar Storage for Scalable Join Processing in the MapReduce Framework. In: ACM SIGMOD International Conference on Management of data, pp. 961-972. Athens, Greece (2011)