IMPLEMENTATION OF COLUMN-ORIENTED DATABASE IN POSTGRESQL FOR OPTIMIZATION OF READ-ONLY QUERIES

Aditi D. Andurkar

Department of Computer Engineering, College of Engineering Pune, Pune, India
aditiandurkar@yahoo.co.in
andurkarad10.comp@coep.ac.in

ABSTRACT

The era of column-oriented database systems has truly begun with open source database systems like C-Store, MonetDb, LucidDb and commercial ones like Vertica. Column-oriented database stores data column-by-column which means it stores information of single attribute collectively. The need for Column-oriented database arose from the need of business intelligence for efficient decision making where traditional row-oriented database gives poor performance. PostgreSql is an open source row-oriented and most widely used relational database management system which does not have facility for storing data in column-oriented fashion. In our work we propose the best method for implementing column-store on top of row-store in PostgreSql along with successful design and implementation of the same.

KEYWORDS

DBMS, Column-store, Row-store, PostgreSql, index, benchmark, predicate, metadata, tuple construction

1. INTRODUCTION

Traditional Row-store DBMS stores data tuple by tuple i.e. all attribute values of an entity will be stored together rather sequentially one after the other. Hence, row-store should be used where information is required from DBMS on a granularity of an entity. But if we are required to access only some of the attributes of a relation then using row-store degrades the performance of these queries [11]. Whenever data is read in row-store, irrelevant attributes will also be accessed due to their fundamental structure of storing an entire entity together [3, 6]. But column-store can access only the required attribute/attributes effortlessly since they store information of an attribute separately [3] thus increasing read query performance. Due to this fundamental difference between these two type of databases, inserting, deleting, updating rows is optimized in row-stores i.e. modifying a tuple becomes easy since attribute values of a tuple are stored contiguously and selecting data is optimized in column-stores i.e. reading only required data becomes easy. Hence, row-stores are called write-optimized where as column-stores are called read optimized [1].

Using row-store or column-store for any application thus depends on the nature of type of query workloads. For usual business processing, row-stores are best considering their performance. But when it comes to analytical applications, column-stores prove to be the best. Business organizations have to handle large amount of data and extract meaningful information from that data for efficient decision making which is commonly termed as Business Intelligence. This includes finding associations between data, classifying or clustering data etc. This lead to a large

Natarajan Meghanathan, et al. (Eds): SIPM, FCST, ITCA, WSE, ACSIT, CS & IT 06, pp. 437–452, 2012.
© CS & IT-CSCP 2012 DOI : 10.5121/csit.2012.2343

area of research called data mining. It is observed that for these kinds of applications, once data warehouse is built i.e. once data is loaded, most of the operations on data are read operations. Unlike business processing, all attributes of an entity would not be required for the analysis. Row-store, if compared with column-store for these applications, has significantly slower performance as it has been shown [1] Because of this, an improvement in the performance of read/select type of queries was required [6]. A possible solution to this problem is to implement column-store by using existing row-store so that DBMS can work for both business processing as well as analytical processing with optimal performance in both. Implementing column-store from the scratch [2] will be the best solution. C-store is the best example of such DBMS developed [2 ,12].

Therefore, the aim is to implement Column-store on top of Row-store in PostgreSql [7] which is a widely used open source object-relational database management system without changing the basic structure of PostgreSql so that its consistency is maintained.

Different approaches [1] for implementing column-store are explored in Section II. These approaches have been studied as part of our literature survey. Section III will explore Column-store approach to be implemented in PostgreSql in detail, the new data structures introduced and how read/write queries are processed i.e. the internal design and working of the query for the column-store implementation. In Section IV, benefits of using our approach will be mentioned along with our implementation from end-user point of view. Section V concludes the paper and explains the future scope of the work.

2. LITERATURE SURVEY

As suggested in a paper written by Daniel J. Abadi, Samuel R. Madden and Nabel Hackem [1] there are three approaches for the implementation of Column-stores such as:

2.1. Implementing Column-store on top of row-store:

The table will be logically broken into multiple tables containing two attributes each <key, attribute>.Tuple will be formed by joining these internal tables on the basis of common table key.

2.2. Modifying the storage layer:

The columns are physically stored one by one so that positional join can be taken to form a tuple and table keys would not be required.

2.3. Modifying the storage and execution layer:

The columns are physically stored column by column. Therefore, positional join can be taken. Also, executor can process the data while it is still in columns [2].

Out of these, modifying the storage layer or execution layer or both would completely change the DBMS. Thus, different approaches for implementing column-store on top of row-store are explored next.

Now, we briefly explain various approaches [1] towards implementing column-store using row-store as suggested by prior related research work. The approaches are considered in an order such that the limitations they put on the kind of possible queries decrease.

A. Materialized views:

Under this approach, an optimal set of materialized views is formed. Every materialized view of this set contains unique set of columns required to answer a distinct query. In this manner, for every possible query, materialized view will be present. So, whenever a select query is fired, relevant materialized view will be accessed giving us quick results. But, for that all the possible queries should be known in advance. In analytical applications specifically, for which we are

looking forward, queries cannot be known prior to execution. This is the biggest disadvantage of this approach which makes it impractical.

B. Index-only plans:

This approach allows storing tables in usual row-oriented form. But, it uses indexing for column-store implementation such that for every column of the table an index is formed which does not follow the physical order in which values are stored. Thus, it forms an un-clustered B+-tree [9] index for each column. Using these indices query can be answered without ever going to the underlying row-store tables. When a predicate is present on any column in the query, the B+-tree index formed can easily give us the result since it divides domain values of that column into continuous ranges. On the other hand, if the predicate is absent, entire index will be searched which will be an overhead. Also this concept of forming index can be problematic because indices are formed on every distinct column. So, for simultaneously considering one predicate and one non-predicate attribute or more, composite index has to be formed. There will always be a limitation as to how many composite indices can be formed.

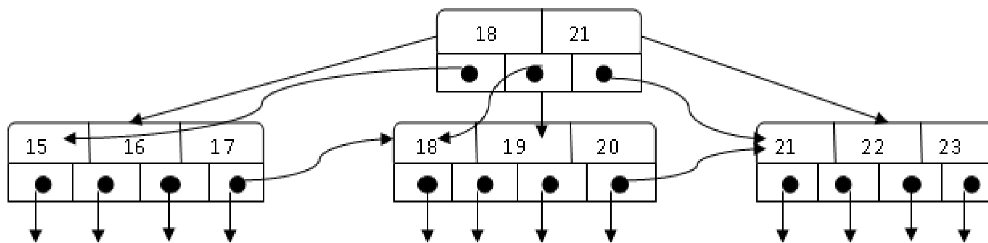

Figure 1.B+-tree index for a column

C. Vertical partitioning:

The forthright approach for designing Column-stores is to partition a relation vertically into various internal tables such that an internal table will be formed for each attribute of the relation.

Since the attribute values of an entity will be stored in different internal tables, there has to be some way to link them for tuple construction. This is because ultimately result of read query will be entity oriented. Therefore, one more column of table key will be included in every internal table so that join [10] of various columns of the relation could be taken on the basis of this common key to construct entire tuple [1]. The required common key can be formed by adding integer position of tuple in the relation table. Primary key can also be used as a common key in all internal tables but primary key can be bulky or composite as well. Therefore, position number is preferred over these. In this fashion, tables will be physically created in the logical schema of the relation.

When a single column is to be fetched, joining will not be required but when multiple columns are to be fetched, joining will become necessary. When a query is fired, only those internal tables will be accessed which correspond to mentioned attribute and remaining ones will be neglected. Then joining will be taken and then will be processed further.

Although simple, there are some advantages and disadvantages of this approach. Most obvious disadvantage is that for each internal table common key column is required, which occupies memory. For every relation, number of tables to be created will be large which again increases work for table creation query.

But on the positive side, there are no limitations on the queries which can be fired on Column-store unlike materialized views. Also, indices need not be formed for the attributes as in index-only plans.

Out of these, the approach of vertical partitioning is best since space overhead is the only disadvantage caused.

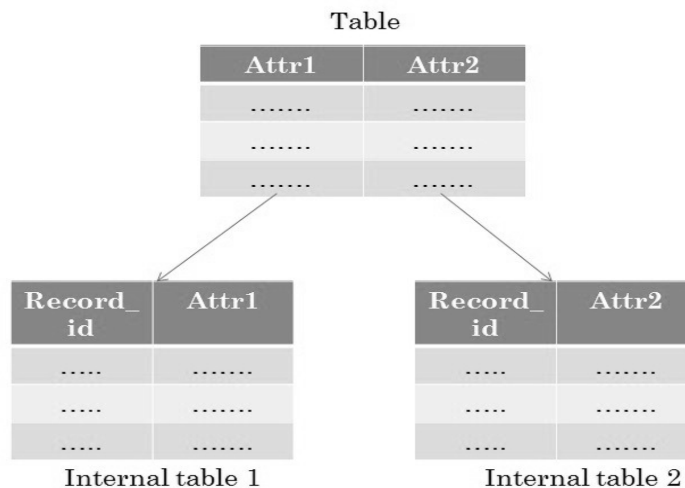

Figure 2.Vertical partitioning

3. POSTGRESQL COLUMN-STORE DESIGN

In this section we give a brief idea about how Vertical partitioning approach [1] is implemented in PostgreSql for having Column-store feature. Design modifications into PostgreSql are proposed as follows:

3.1. Creating a Column-store Relation:

For every column-store type of relation a number of internal relations will be created equal to number of attributes present in the relation. Hence, a unique identifier (Oid) will be allotted for every internal table created. No table is created by the name of mentioned relation, directly internal tables are created corresponding the attributes, thus saving one unique identifier.

Each internal relation will consist of two columns <record_id, attribute> wherein record_id column will be common with other attributes of the same relation. This column will act as a unique identifier of the tuple as a whole. For creating a column-store, table users will be given an option of colstore. A new keyword colstore will be included in the create query. So, a new query would look like

Create colstore table table-name (attr1 datatype, attr2 datatype ...);

The rest of all PostgreSql queries remain syntactically intact. Along with internal tables we propose to create a sequence and a view corresponding to main relation. A sequence is a database object which can generate unique integers sequentially. The reason for creating an internal sequence is that record_id has to be incremented internally whenever any data is inserted into the relation. So simply sequence value can be incremented internally and passed along with the values sent by the user.

View is basically a stored query. It does not take much space as only view definition is stored inside database and not the data. Whenever any changes are made in the tables which are present in the view, those changes are automatically reflected in the result since query stored for

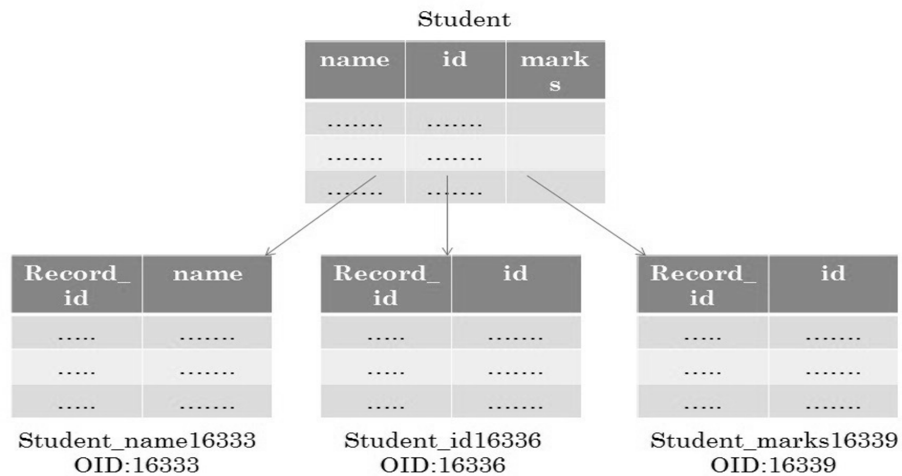

Figure 3: Creating Colstore Table

view is reflected every time view is called. This feature of view will help us whenever we want to take join of all internal tables(For example, select * from table name ...). So a logical view is formed which takes natural join of all internal tables on the basis of record_id. But when join of all internal tables is not required, we propose to take join of only those internal tables whose corresponding attributes are mentioned in the query as predicate or non-predicate rather than using existing view. Name of internal tables, sequence and view will be made unique by concatenating their respective unique identifiers (Oids) to their names so ambiguity in the names will be easily avoided. Also it is made sure that user would not be able to make any Row-Store or Column-Store relation with the name of any existing relation.

3.2. Metadata for the Column-store Relation:

Now that internal tables, sequence have all been created, there is a need to store metadata of the relation i.e. to identify which internal tables correspond to which column-store relations. For storing this mapping between the relation and internal tables, a new data structure is created named pg_map. One more data structure is created for storing view and sequence id generated for column-store relation named pg_attrnum. Every entry in these two system tables is uniquely identified by a key. For pg_map <relation-name, attribute number> forms a key whereas for pg_attrnum, <relation-name> forms the key. These two system tables are showed in Figure 4.1 and 4.2.

Relation Name	Attribute Number	Internal Table Name	Internal Table Oid
Student	1	Student_name16333	16333
Student	2	Student_id16336	16336
Student	3	Student_marks16339	16339

Figure 4.1.System Table pg_map

Relation Name	Number of Attributes	View Oid	Sequence Oid
Student	3	16442	16445

Figure 4.2.System Table pg_attrnum

When a column-store table is created, respective values are entered into these tables. Therefore, whenever user wants to fire any type of query on column-store relations. These system tables will be accessed. System caches are built for both the relations so that searching in these tables will be faster. These tables will be searched on the basis of a unique key as explained earlier.

Similarly, when any column-store relation is dropped, all internal tables, corresponding view and sequence are dropped. Also, corresponding system table entries are deleted.

3.3. Inserting Data into Column-store Relation:

All modifications for executing these kinds of data manipulation queries are done at the query tree formation stage. Values which are passed by user for insertion are taken as a list in PostgreSql [7]. This list is broken into separate column values & values are passed to the corresponding tables along with the next unique sequence value generated. If a select clause is present within insert then it will be processed and expression list generated will be broken and sent similarly. This way even if user fires only one insert statement, multiple insert statements will be generated & processed internally.

3.4. Altering the Column-store Relation:

Adding or dropping any column from Column-store means creating or deleting internal table respectively. So, if user wants to add a column, an internal table will be created corresponding to the relation mentioned & system tables will be updated accordingly. Similar is the case with dropping a column. But one thing has to be kept in mind that if this is done then old view would not work hence, it will be dropped & a new view will be created.

3.5. Selecting data from Column-store Relation:

Select query is the one of which Column-Stores are expected to improve the performance. In Row-Stores, even when only some of the attributes are required to be accessed, all irrelevant attributes are accessed which increase execution time of the query. Using Column-Stores only attributes which are present the select query as a predicate or non-predicate, are accessed which reduces execution time as compared to that in Row-Stores [8]. This concept is implemented for Column-Store implementation in PostgreSql. Basically as we have created internal tables for every attribute of any Column-Store relation, we have to take join of required internal tables to produce the result. Because the result of any query is always entity-oriented, this tuple construction is required by taking join. Here join is actually natural join taken on the basis of the common key defined earlier i.e. record id. For taking join of only required internal tables, we first identify attributes present in the select query as either predicates or non-predicates. Then we find internal table names for all those attributes which are present in the query and take their natural join based on the common key Record_id and form a join node. We add this join node to list of from-clause. This process is repeated for all relations present in the from-clause entered by the user. Point to be noted is that join internal tables corresponding to only one relation is taken. In such a way, we would not have to access irrelevant attributes for any relation. Internal
tables corresponding to irrelevant attributes will not be included in the join formation. Only when all attributes are needed to be accessed (For example, select * from table-name...), view created will be accessed directly rather than adding internal tables one-by-one to the from clause. View is nothing but natural join of all internal tables of the mentioned colstore relation.

Figure 5 explains how select operation is performed on a column-store table. This will make the scenario more clear. Let us see what difference does this approach make in the query plan of a SELECT query which is as follows:

```
Select
        sum(l extendedprice) / 7.0 as avg_yearly
from
        lineitem,part
where
        p_partkey = l_partkey
        and p_brand = 'Brand#13'
group by
        avg_yearly
order by
        avg_yearly;
```

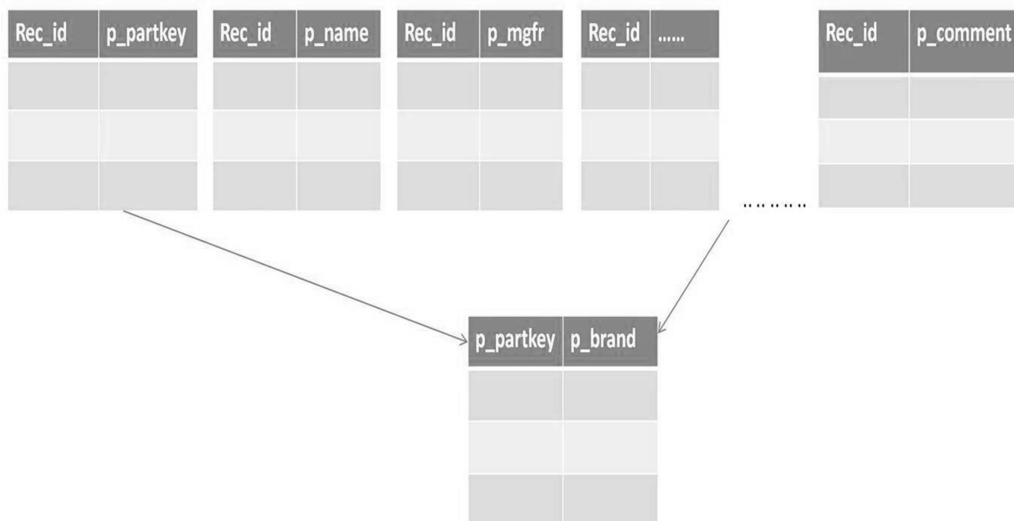

Figure 5: Taking Join of Internal Tables

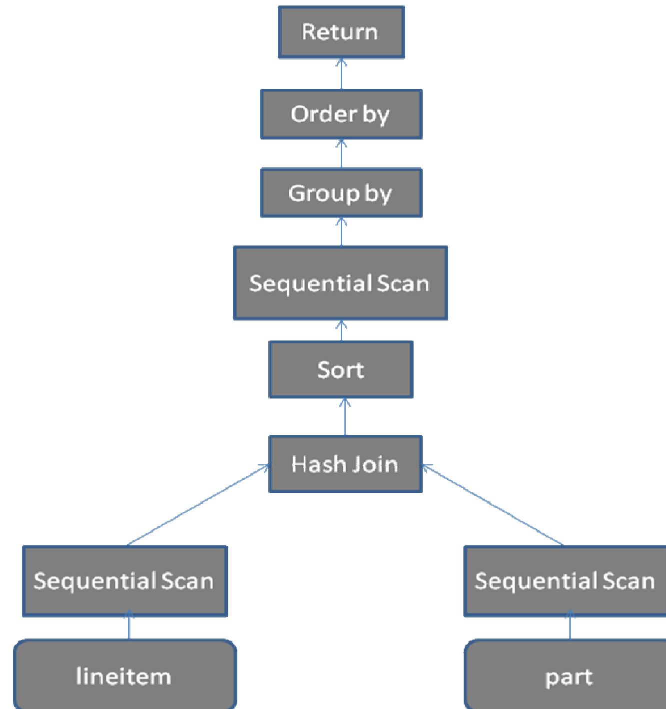

Figure 6: Query Plan of select query on Row-Store

In this SELECT query, p_partkey, l_partkey and p_brand are predicates. Predicate is a attribute present in a query on which some condition is applied. Also, l_extendedprice is a non-predicate. Non-predicate is an attribute present in the query which is to be projected. Hence, these attributes are considered while taking natural join for corresponding tables. Here, 2 natural joins of internal tables will be taken. One will be of lineitem_l_extendedprice and lineitem_l_partkey and another will be of part_p_partkey and part_p_brand.

It can be seen in figure 7 that lineitem table has been replaced with the internal table names whose corresponding attributes are present in the query. l_extendedprice is present as predicate, l_partkey is present as non-predicate in the select query so before forming a query plan from query tree, we modify query tree by replacing lineitem with natural join of lineitem_l_extendedprice and lineitem_l_ partkey so that internal tables are used for accessing data. Similarly, part table is replaced by natural join of part_p_partkey, part_p_brand. Now, let us consider the consequences of doing this. By looking at the plan tree one will obviously say that unnecessary overhead of join is introduced for lineitem as well as part tables. But, as dataset of lineitem, part grows the performance of Row-store degrades since irrelevant columns are also accessed in Row-Store as can be seen in Figure 6.

Suppose part contains 9 columns and lineitem contains 16 columns. Assume number of rows are in thousands. Then in Column-Stores, we just have to consider 4 internal tables corresponding to 4 attributes. Form 2 join nodes by taking 2 natural joins. Afterwards, their hash join is taken. But for Row-Stores, we have to consider 2 tables having 25 attributes and take hash join of 2 tables one having 16 attributes and the other having 9 attributes. This overhead of Row-Store goes on increasing with the increase in number of columns and data inserted into tables. Thus, our approach works very well on big data having large number of attributes.

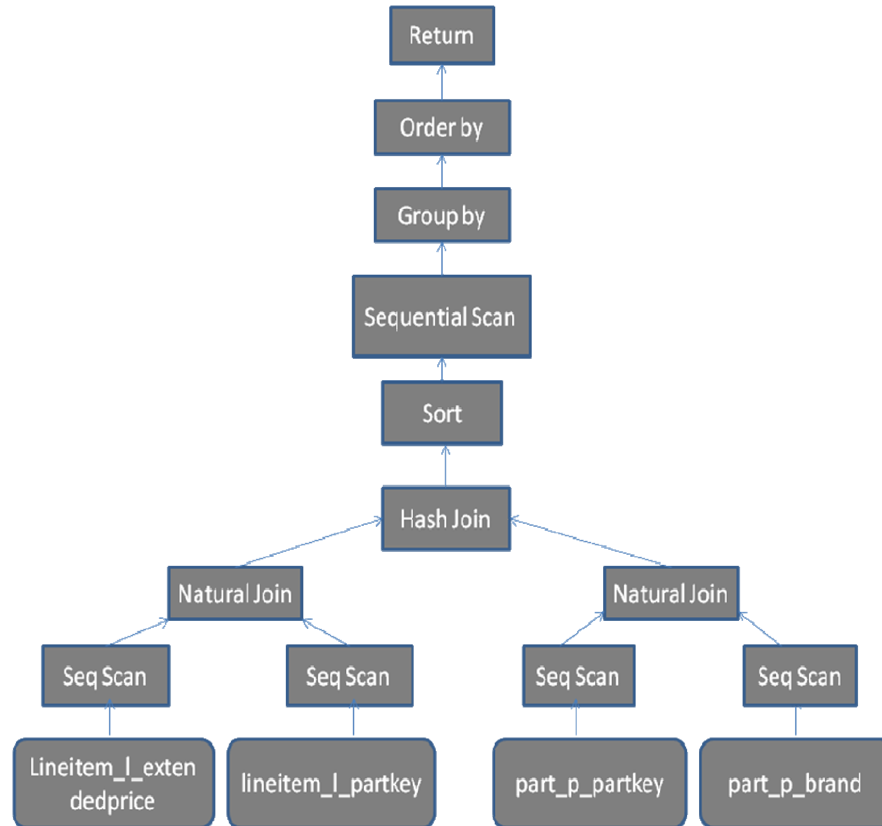

Figure 7: Query Plan of select query on Row-Store

3.6. Dropping the Column-store table:

When any Column-Store relation is dropped, all internal tables are dropped. Also, view, sequence created at the time of table creation are also dropped. Most importantly, system tables are freed from entries corresponding to relation being dropped.

4. POSTGRESQL COLUMN-STORE FROM END-USER POINT OF VIEW

In this section, we will check all the benefits of implementing Column-store on top of row-store this way & also will analyse our design implementation from end-user point of view.

When large numbers of relations are stored & a query is fired, row-store has to access all attributes of mentioned relations whereas column-store has an advantage of considering only those internal tables whose corresponding attributes are mentioned in the query while forming join.

Concept of column-store is implemented such that PostgreSql structure [7] remains intact. Also, the aim is to use PostgreSql structure to maximum extent so consistency will be maintained in the source code.

The column-oriented queries will be executed transparent to the user so that user does not have to bother about the internal processing & the query syntax would not be modified at all. All the modifications will be done at the query tree formation stage i.e. just before the query gets planned.

The performance of Read-oriented queries in Column-oriented databases is expected to increase by a factor of two as compared to their performance in Row-oriented databases [1]. The project is under testing phase. The performance will be evaluated on the basis of TPC-H benchmark.

As PostgreSql is open-source, it is expected that this will be a contribution to the open-source world. Apart from create, drop, alter, insert, update, delete and select queries we are also looking forward to create physical indices on internal table columns. Also, constraint checking is important because it gives control over the data being inserted into the column-store. Therefore, we plan to include them in our column-store.

5. EXPERIMENTS AND RESULTS

The main aim of our work is improving performance of SELECT query. Write queries like insert, update, delete will give be very slow in Column-Stores as compared to Row-Stores. On small dataset, the results of SELECT queries in Column-Store are poor which was as expected. This is because of a number of join operations performed for each relation. But, on large datasets, Column-Store gives excellent results for select queries which have small number of attributes to be accessed. Our implementation is basically for large datasets. For evaluating the performance of our implementation, we use TPC-H benchmark [14, 15, 16, 20]. Dataset size we have taken for our analysis is 5000 tuples per table. The schema diagram of their dataset is as shown in Figure 9.

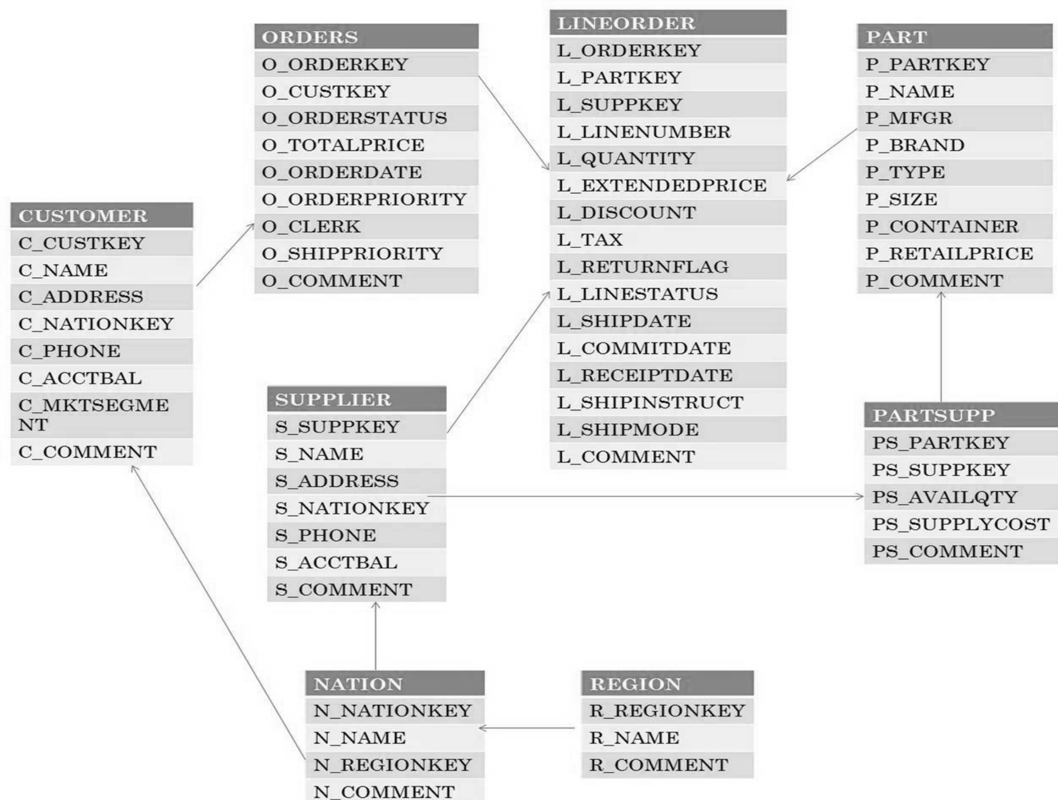

Figure 9: E-R Diagram of TPC-H Benchmark Dataset

For each attribute of each table shown in the Figure 9, an internal table will be created in our implementation of Column-Store Database. Firstly, we check how is the performance of select query on gradually increasing the number of columns accessed. The lineitem table consists of 16 attributes and orders table consists of 9 attributes. Hence, we start by selecting 2 attributes, one

from each table. Then, we gradually increase this number to 25 and observe the performance of SELECT query. This will give us exact idea of how SELECT query in Column-Store behaves.

Table 1. Experimental results for simple select query

Number of Attributes Accessed	Execution Time in seconds for Row-Store	Execution Time in seconds for Column-Store
2	257.462 sec	128.731 sec
3	257.326 sec	128.899 sec
4	259.526 sec	153.923 sec
5	258.694 sec	168.932 sec
6	260.090 sec	208.639 sec
7	268.338 sec	226.282 sec
8	270.112 sec	264.680 sec
9	273.445 sec	288.565 sec
15	280.778 sec	8543.667 sec
25	290.199 sec	20899.542 sec

The graph of table 1 is plotted as shown in Figure 10.

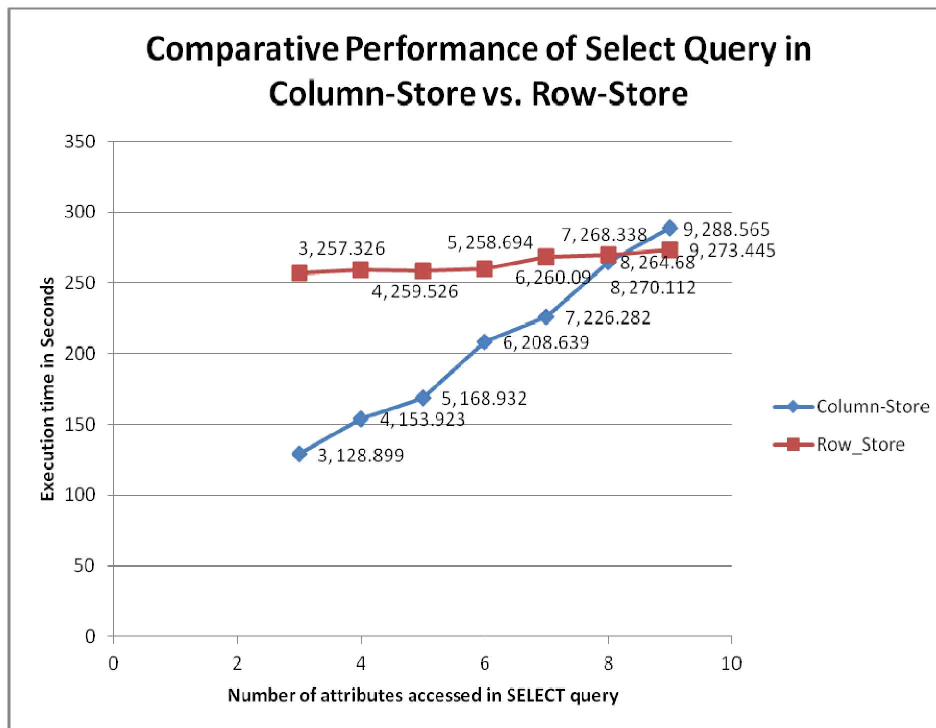

Figure 10: Comparison of Column-Store vs. Row-Store

This table 1 shows that as the number of attributes approach maximum possible value the execution time goes on increasing. Until the value of number of attributes is 8, the execution time required for Column-Store is less than that for Row-Store. In fact, Column-Store execution time is excellent until number of attributes accessed are 8. Again point to be considered is that "orders" has 9 attributes which is less than 15 of "lineitem". Therefore, the increase in execution time is not always in the same proportion. It can be seen that when number of attributes are increased

from 5 to 6 then there is a sudden increase in execution time. This is because, the effect of accessing one attribute from orders on execution time is more than the effect of accessing one attribute from lineitem.

From these results, it is concluded that if 1/3th of the attributes are accessed then performance of Column-Store is very good as compared to Row-Store. But, 2 conditions should be satisfied. First, number of attributes for tables must be high. Second, Dataset size should be in the range of thousands, lacks and more. The more the number of attributes and the larger the dataset, the lesser will be the execution time in Column-Store as compared to Row-Store.

Now let us see the performance comparison of Row-Store against Column-Store with the help of some TPC-H benchmark[16] queries. We have considered those queries which are suitable for Column-Oriented databases. i.e. queries which have less attributes to be accessed. For queries which access large number of attributes, performance will certainly be worse as compared to Row-Stores. We have considered following 10 queries for evaluating our performance.

1. Select
```
        l_returnflag,sum(l_quantity) as sum_qty,sum(l_extendedprice) as
        sum_base_price,sum(l_extendedprice * (1 – l_discount) * (1 + l_tax)) as
        sum_charge,avg(l_extendedprice) as avg_price, avg(l_discount) as avg_disc
    from
        lineitem
    group by
        l returnflag;
```

Row-Store: 12.015 sec
Column-Store: 7.0 sec

2. Select
```
        n_name,sum(l_extendedprice) as revenue
    from
        nation,lineitem,region
    where
        r name = 'AFRICA'
    group by
        n_name
    order by
        revenue;
```

Row-Store: 16.086 sec
Column-Store: 1.527 sec

3. Select
```
        c_name,sum(l_quantity)
    from
        customer,orders,lineitem
    where
        c_custkey = o_custkey
        and o_orderkey = l_orderkey
    group by
        c_name;
```

Row-Store: 13.087 sec
Column-Store: 9.14 sec

4. Select
```
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,part
where
    p_partkey = l_partkey
    and p_brand = 'Brand#13';
```

Row-Store: 12.978 sec
Column-Store: 8.284 sec

5. Select
```
    min(ps_supplycost)
from
    lineitem,supplier,nation,region,part,partsupp
where
    p_partkey = l_partkey
    and s_suppkey = l_suppkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'AMERICA';
```

Row-Store: 14.847 sec
Column-Store: 1.739 sec

6. Select
```
    sum(l_extendedprice * l_discount) as revenue
from
    lineitem
where
    l_quantity<25;
```

Row-Store: 12.936 sec
Column-Store: 2.638 sec

7. Select
```
    l_shipmode,
    sum(case when o_orderpriority = '1-URGENT' or o_orderpriority = '2-HIGH' then 1 else
    0 end) as high_line_count,
    sum(case when o_orderpriority <> '1-URGENT' and o_orderpriority <> '2-HIGH' then 1
    else 0 end) as low_line_count
from
    lineitem,orders
group by
    l_shipmode;
```

Row-Store: 326.421 sec
Column-Store: 162.005 sec

8. Select
```
    100.00 * sum(case when p_type like 'PROMO%' then
    l_extendedprice *(1 – l_discount) else 0 end) /
    sum(l_extendedprice * (1 – l_discount)) as promo_revenue
from
    lineitem, part;
```

Row-Store: 388.471 sec
Column-Store: 193.018 sec

9. Select
```
        l_suppkey
    from
        lineitem
    where
        l_shipdate >= date '1994-08-01'
        and l_shipdate < date '1994-08-01' + interval '3' month
    group by
        l_suppkey;
```

Row-Store: 12.959 sec
Column-Store: 6.507 sec

10. Select
```
        substring(c_phone from 1 for 2) as cntrycode
    from
        customer
    where
        substring(c_phone from 1 for 2) in ('40', '41', '33', '38', '21', '27', '39');
```

Row-Store: 0.021 sec
Column-Store: 0.018 sec

6. CONCLUSION AND FUTURE SCOPE

We Read queries when applied on huge datasets perform poorly due to their storage structure (tuple-by-tuple). But, Column-Stores give a very good performance for such queries. In PostgreSql, Column-Store could not be built from scratch due to its Row-oriented structure. Thus, we decided to implement Column-Store on top of Row-Store. The design of Column-Store on top of Row-Store is a great challenge because modifications should be done at proper stages of query processing to get optimal performance improvement over Row-Store. In our work, we investigated various approaches of implementation of Column-Store on top of Row-Store and found that Vertical Partitioning is most preferred of all due to less complexity and no limitations on the kind of possible read queries. We studied the architecture of PostgreSql. After understanding the intricacies of PostgreSql, query tree formation stage was found to be most suitable for modification. The thesis discussed the design and architecture of Column-Store Database System along with its implementation in PostgreSql.

The results show that performance of our Column-Store implementation is very high as compared to Row-Store in queries which access less attributes. Also, relation should consist of large number of attributes. We see that as number of columns accessed increases, the performance of Column-Store degrades which is as expected. This is because number of joins of internal tables increases in such a case which leads to increase in execution time. The same case would be very efficient in Row-Store. But, the idea behind Column-Stores is to use them for specific applications as described in section 3.

One very useful extension to this work is to pack many tuples together to form page sized "Super Tuples" [11]. This way duplication of header information can be avoided and many tuples could be processed together in a block. The super tuple design uses a nested iteration model, which ultimately reduces CPU overhead and disk I/O. But, again accessing single tuple becomes difficult here. Since, our implementation is application specific, it can be assumed that we would not be required to access specific tuple. Compression techniques [4] could also be applied while

data storage not for saving disk space but for increasing performance by doing operations on compressed data. Compression optimization is unique to Column-Stores since similar data are stored on disk contiguously. This is because data of same attribute will be of same data type.

ACKNOWLEDGEMENTS

We would like to thank our project guide Mr. Shirish Gosavi, our institute – College of Engineering Pune, for providing essential guidance throughout the work also, Mr. Arvind Hulgeri for his valuable guidance. A major thanks goes to dearest and closest friends and of course parents who have been the backbone, advisors and constant source of motivation throughout the work.

REFERENCES

[1] Daniel J. Abadi, Samuel R. Madden, Nabil Hachem, (2009-12) "Column-Stores vs Row-Stores : How Different Are They Really?", Vancouver, BC, Canada, SIGMOD'08.

[2] Mike Stonebraker, D. J. Abadi, (2005) "C-Store : A Column-oriented DBMS", 31st VLDB Conference, Throndhiem, Norway.

[3] D. J. Abadi, (2008) "Query execution in column-oriented database systems", MIT PHD Dissertation, PhD Thesis

[4] D. J. Abadi, S. R. Madden, and M. Ferreira, (2006) "Integrating compression and execution in column-oriented database systems", In SIGMOD, pp 671-682.

[5] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden, (2007) "Materialization strategies in a column-oriented DBMS", In ICDE, pp 466-475.

[6] Stavros Harizopoulos (HP Labs), Daniel Abadi (Yale), Peter Boncz (CWI), (2009) "Column-Oriented Database Systems", VLDB Tutorial.

[7] http://www.postgresql.org/

[8] S. Harizopoulos, V. Liang, D. J. Abadi, and S. R. Madden. (2006) "Performance tradeoffs in read-optimized databases", VLDB, pp 487–498.

[9] G. Graefe, (2007) "Efficient columnar storage in b-trees". SIGMOD Rec., 36(1):pp 3–6.

[10] A. Weininger, (2002) "Efficient execution of joins in a star schema", SIGMOD, pp 542–545.

[11] Halverson, J. L. Beckmann, J. F. Naughton, and D. J. Dewitt, (2006) "A Comparison of C-Store and Row-Store in a Common Framework." Technical Report TR1570, University of Wisconsin-Madison.

[12] C-Store source code http://db.csail.mit.edu/projects/cstore/

[13] R. Ramamurthy, D. Dewitt, and Q. Su. (2002) "A case for fractured mirrors". In VLDB, pp 89-101.

[14] TPC-H, http://www.tpc.org/tpch/.

[15] TPC-H benchmark with postgresql, http://www.fuzzy.cz/en/articles/dss-tpc-h-benchmark-with-postgresql/.

[16] TPC-H Result Highlights Scale 1000GB. http://www.tpc.org/tpch/results/tpchresultdetail.asp?id=107102903.

[17] Daniel J. Abadi, (2007) "Column Stores For Wide and Sparse Data", Conference on Innovative Data Systems Research (CIDR) 710.

[18] S. Harizopoulos, V. Liang, D. J. Abadi, and S. Madden,(2006) "Performance tradeoffs in read-optimized databases", VLDB, pp 487-498.

[19] Vertica. http://www.vertica.com/.

[20] Patrick E. ONeil, Elizabeth J. ONeil, and Xuedong Chen, The Star Schema Benchmark (SSB), http://www.cs.umb.edu/poneil/StarSchemaB.PDF.

Authors

Aditi Deepak Andurkar

MTech in Computer Engg. & IT,

COEP, Shivajinagar, Pune.