International Journal of Research

Available at https://edupediapublications.org/journals

p-ISSN: 2348-6848
e-ISSN: 2348-795X
Volume 03 Issue 04
February 2016

The Implementation of Column-Oriented Database in Postgresql for Improving Performance of Queries

Sukhdeep Kaur
Guide Name: - Dr. Dinesh Kumar
University College of Computer Applications
EMAIL ID: - Sukhmahal48@gmail.com

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. Row-Store database stores data row-by-row which means it stores all information of an entity together. Hence, when there is a need to access the data at the granularity of an entity, Row-Store performs well. But, in decision making applications, data is accessed in bulk at the granularity of an attribute. The need for Column-oriented database arose from the need of business intelligence needed 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. This work discusses the best method for implementing column-store on top of Row store in PostgreSql along with successful design and implementation of the same. Performance results of our Column-Store are presented, and compared with that of traditional Row-store results with the help of TPC-H benchmark. We also discuss about the areas in which this new feature could be used so that performance will be very high as compared to Row-Stores.

Index Terms — Data Mining and Data Warehousing; Knowledge Database; Postgre SQL Database System and Database Relations

1. INTRODUCTION

Whenever we say relational data, most obvious interpretation is a table which has attribute as one dimension and entity as another. We imagine a table stored on some storage media in such a 2-dimensional form. But this is just a concept for better understanding of any relation stored some storage media. At physical level, it is not possible to store data like the way we imagine. Therefore, Data are physically stored consecutively one after another in 1-dimensional way. While storing in 1-dimensional manner we have 2 choices. We can either store the data entity by-entity or attribute-by-attribute. This leads to two kinds of databases Row-Store and Column-Store respectively. They are as follows:

1.1 ROW - STORE

Traditional Row-Store DBMS stores data tuple by tuple i.e. all attribute values of an entity are stored together rather sequentially one after the other. Hence, Row-store is used where information is required from DBMS on a granularity of an entity. In Row-Store, write-queries like insert, delete, update can be easily performed [3] since they apply for an entity/tuple. But, read-queries like select have predicates which are conditions to be applied on attributes/columns and non- predicates which are columns to be projected as result of the query. Therefore, these queries apply for attributes/columns rather than entity/tuple. Hence, Row- Store is said to be Write-Optimized since it favors write operations. The Figure 1.1 shows the manner in which data is stored in Row-Stores on physical media.

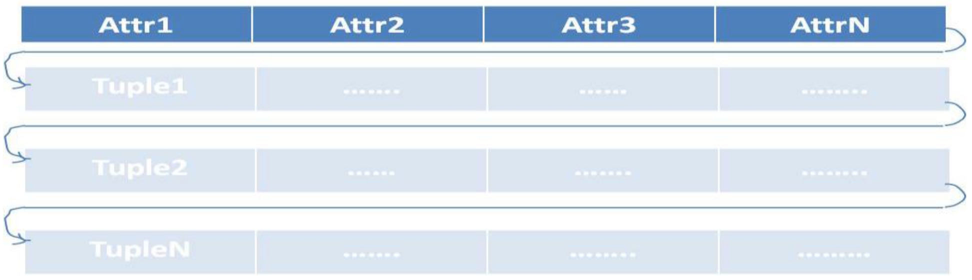

Figure 1.1: Relation stored Row-by-Row

1.2 COLUMN – STORE

Column-Store DBMS stores data column by column i.e. all values of an attribute are stored collectively. So that, ith value of every column of a relation will form a tuple together. Hence, Column-store is used where information is required from DBMS on a granularity of an attribute. In Column-Store, read-queries like select can be easily performed [3] since they apply for attribute/column. But, write- queries like insert, update, delete which are applied for entity or tuple are not easily processed. Therefore, Column-Store is said to be Read-Optimized since it favors read operations. The Figure 1.2 shows the manner in which data is stored in Column-Stores on physical media.

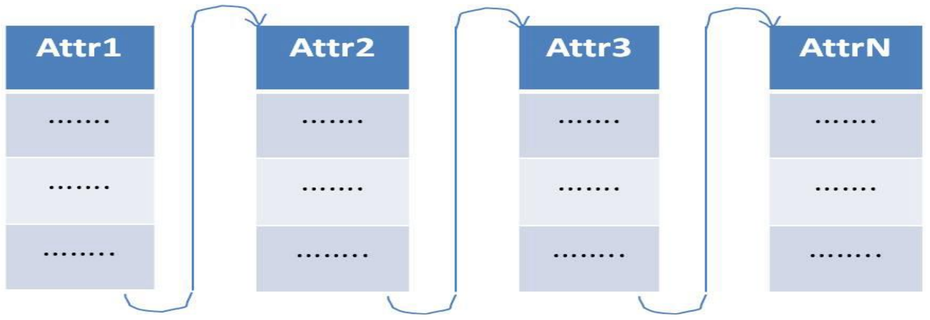

Figure 1.2: Relation stored Column-by-Column

1.3 COMPARISON OF COLUMN – STORE VS. ROW STORE

The question of which type of database system is better depends on the kind of query workloads [3]. If after data insertion, updation, deletions are going to be more and if accessing entire tuples is a need then Row-Stores are the best. They are the most common ones for business transactional data processing. For example, a bank uses databases to store information of its customers. Some customer A might want to transfer money to the account of customer B. Here, Customer A and B are entities. Here a simple updation has to be done in accounts of A and B both which is deduct amount x from account of A and credit amount x to account of B. As it can be seen information will be required by the bank from DBMS on the granularity of an entity here, Row-Store which stores data entity-by-entity will be most obvious choice out of the two database systems we studied. Therefore, when it comes to analytical applications or decision making applications, column- stores prove to be the best [3]. 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 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]. Again there are some optimizations possible with Column-Stores and are not possible with Row-Stores which can improve performance of Column-Stores compared Row-Stores significantly [1, 3]. The first and the most important is Compression [4]. As data are stored column-by-column, compression can be easily applied on a column. This is possible because a column has a data type in which similar data is stored. Like mobile number in India will always contain 10 digits. If one could store data is compressed format, performing column extraction will become very easy. Next is block processing, where multiple tuples from a column are extracted and are passed as a block from one operator to another. There is one more optimization called as Late Materialization where tuple construction i.e. joining of columns is performed as late as possible. These optimizations are specific to Column-Stores because Row-Stores do not have required properties to apply these optimizations.

1.4 POSTGRESQL DATABASE MANAGEMENT SYSTEM

PostgreSql is world's most advanced object-relational database management system [7]. It is free and open-source software. It is developed by PostgreSql Global Development Group consisting of handful of volunteers employed and supervised by companies such as Red Hat and Enterprise DB. PostgreSql is available for almost all operating systems like: Linux (all recent distributions), Windows, UNIX, Mac OS X, FreeBSD, OpenBSD, Solaris, and all other Unix-like systems. It works on all majority of architectures like: x86, x86-64, IA64, PowerPC, Sparc, Alpha, ARM, MIPS, PA-RISC, VAX, M32R [7]. MySQL and PostgreSql both compete strongly in field of relational databases since they both have advanced functionalities and also comparable performance and speed and most importantly they are open-source. PostgreSql which uses a client/server model can be broken-up into three large subsystems [7]:

1. Client Server: This subsystem consists of Client Application and Client Interface Library. Client Application wants to perform some operation on the data. This Client application can be anyone of text-oriented tool, a graphical application, or some specialized tool. Therefore, it is the responsibility of the Client interface library to convert each client application to proper SQL queries that the server can understand and parse. Hence, server need not parse different languages and waste its time, but only interpret SQL queries, which make the whole system faster.
2. Server Process: Postgres server executes daemon thread constantly which is the master server process. When it receives a call from a client process, it forks a

new postgres server process. Once the process is created, it links the client and postgres process so that they can communicate without the postmaster. An SQL query is passed to Postgres server and it is incrementally transformed into result data. The master server process always waits for calls from clients. But, slave master processes and clients come and go.

3. Database Control: Stored data is accessed through the Storage subsystem.

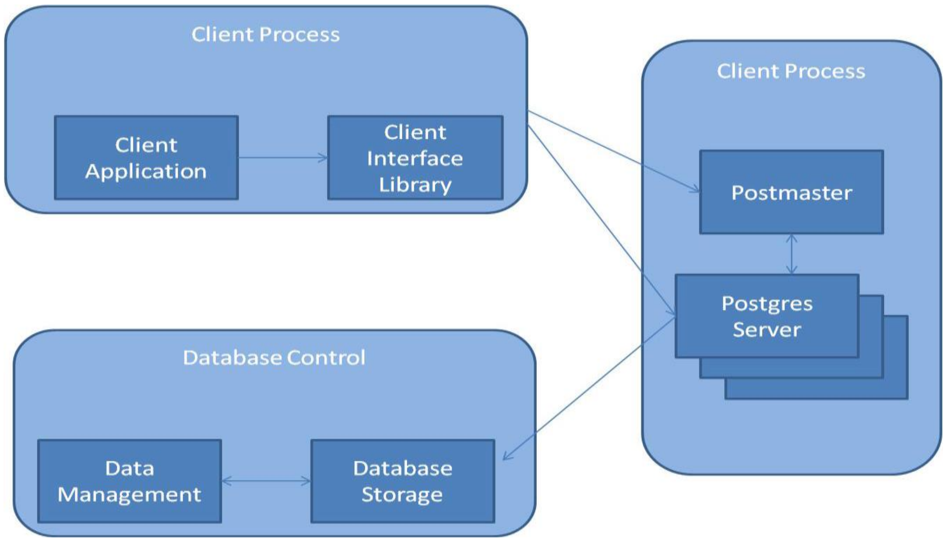

Figure 1.3: Architecture of PostgreSql Database Management System

PostgreSql conforms to SQL standard. Its features [7] are as follows:

1. Complex Queries: Queries can nested, consisting of many operators or the criteria may be complicated.
2. Triggers: It is a piece of code which gets executed implicitly when a certain event occurs on a specific relation in the database.
3. Views: View is a query which is stored inside database. Whenever a view is accessed, its corresponding query gets executed internally and this is transparent to the user.
4. Foreign keys: It is a key in a relational table that matches candidate key of another table.

1.5 THESIS OBJECTIVE AND SCOPE

Our aim is to implement Column-store on top of Row-store in PostgreSql to improve performance of Read-queries. PostgreSql is basically an open-source row-oriented database which does not have any feature of storing data column-by-column. We are enhancing PostgreSql to have this feature so that for decision making applications performance of Read queries will be higher than that compared with row-oriented database system. Modifications will be done in such a way that all queries processing for Column-Store table will be transparent to the user.

1.6 THESIS OUTLINE

In section II, we explore structure of PostgreSql, Column-Store optimizations and different approaches [1] for implementing column-store as part of our literature survey. In Section III, we explain 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, we will evaluate performance of our implementation by using TPC-H benchmark and compare it with Row-Store PostgreSql. Section V concludes the report and explains the future scope of the work.

LITERATURE SURVEY

Keeping our aim in mind, our literature survey will consist of 3 main areas Post-greSql, Column Store Databases and Approaches for Column-store implementation.

2.1 POSTGRE SQL

As we have implemented Column-Stores in PostgreSql, there is a need to understand what important aspects of PostgreSql are. In chapter 1 we have seen client/server model of PostgreSql. In this section we will see the system catalogs [7] and data types defined in PostgreSql and query processing stages [7].

2.1.1 SYSTEM CATALOGS AND DATA TYPES

System Catalog is the metadata for the system and Metadata is data about data i.e. which gives descriptive information about stored data itself. For modifying PostgreSql, these data structures should be thoroughly understood. Also, we might be required to create new ones. The data structures are as follows:

1. System Catalog for describing a relation
    - pg class: For each row-oriented table, one row is added to this system table containing name of the relation, its owner, permissions.
    - pg attribute: For each row-oriented table, one row is added to this system table for each attribute present in the table having attribute name, data type etc.
    - pg index: For every index created on any column of a relation, a row is added to this system table containing index name, type of index etc. Also index is a relation so its entry is also made into pg class.
    - pg proc: For every defined function, an entry is made into this system table about its name, arguments types, result types etc.
    - pg language: For defined functions, there is always some implementation language like C, SQL, PL/SQL. This entry is made into pg language.
2. System Catalog for Aggregate Functions
    - pg aggregate: For each defined aggregate function, an entry is made into this table about their working-state data type, update function, result function.

3. System Catalog for Operators
   - pg operator: Various types of operators are used while handling expression. These operators are included in this system table.
4. System Catalog for Data Types
   - pg type: An entry is made into this system table for every data type defined. It can built-in or user-defined.

2.2 COLUMN - STORE

There are some advantages of Column-Store over Row-Store. These advantages are due to the way in which data is stored in Column-Store.

1. Compression:

   Storing values of a column contiguously makes the adjacent values on disk similar to each other or rather of the same data type [4]. This leads us to one of the most important optimization called compression. We can compress the data using several existing compression techniques. This optimization is not possible in row-stores because in a tuple all the attributes are different. Therefore, compression is a new optimization strategy for column-stores.

2. Materialization:

   In Column-stores, attribute values of a single tuple are stored at multiple locations on disk. But most of the times queries try to access more than one column from database. The output standard is always an entity-at-a-time not column-at-time. Therefore, the attributes from different columns of the same relation have to be combined together into rows to be displayed. This reconstruction of tuples is called as materialization [5]. This is nothing but taking join of various columns. There are two ways to reconstruct tuples from column-stores:

i. Early Materialization:

In this type, whichever columns are mentioned in the query are retrieved first join is taken so that we will have row-oriented tuples. And then the predicated are applied. Those who satisfy are answer to the query. But this way all advantage of column-stores is lost and process becomes less efficient. In early materialization [3, 5], as soon as a column is accessed, its values are added to an intermediate result tuple, eliminating the need for future re-accesses.

ii. Late Materialization:

In this approach, tuples are not formed until some part of the query plan has been processed. Here, predicates are applied on columns of relation first and those positions which satisfy the predicates are listed. Finally, a position-wise AND operation is performed on them. Now, you have a list of positions which satisfy the predicates so again access the columns required and extract the corresponding values and take a join. This seems to be a better strategy than early materialization [3, 5].The primary advantages of late materialization [5] are that it allows the executor to use high-performance operations on compressed, column- oriented data and to defer tuple construction to later in the query plan, possibly allowing it to construct fewer tuples.

2.3 APPROACHES FOR IMPLEMENTING COLUMN - STORE

Keeping our aim in mind, we did some literature survey to study various different approaches for Column-Store implementation. Now, we explore the approaches in detail.

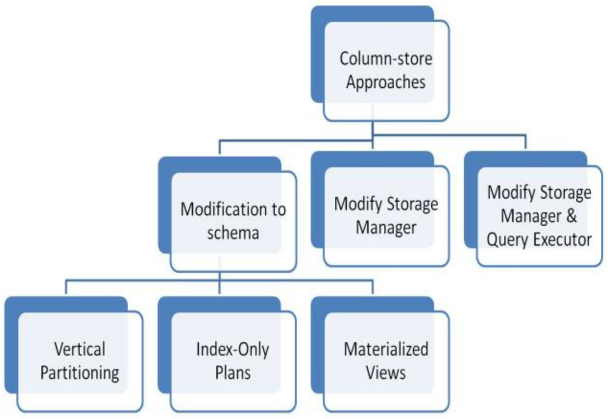

Figure 2.1: Types of approaches to design Column-store databases

Daniel J. Abadi, Samuel R. Madden and Nabel Hackem [1, 3] have done a lot of work on Column-Stores in recent times. Specifically Daniel J. Abadi has done immensely valuable work in the field of Column-oriented databases [1, 2, 3, 4, 5, 6, 17, 19]. They implemented Column Store from scratch called "C-Store" [2, 12]. They suggested three approaches for the implementation of Column-stores which are as follows:

2.4 SUMMARY

In the literature survey, we learned about PostgreSql query processing in brief. Thus, we decided to do modifications in the process between parse tree formation and query tree formation stages which simplifies our code and its understanding. Then we studied optimizations specific to Column-Store. For implementation of Column-Store, we considered various approaches from the past research. We observed that out of these, modifying the storage layer or execution layer or both would completely change the DBMS. Hence, changing logical schema is chosen as an approach for our implementation. Considering all advantages and disadvantages of all three approaches, vertical partitioning is considered for implementing column-store on top of row-store.

PROBLEM FORMULATION

Before developing research we keep following things in mind so that we can develop powerful and quality research.

3.1 PROBLEM FORMULATION

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. Row-Store database stores data row-by-row which means it stores all information of an entity together. Hence, when there is a need to access the data at the granularity of an entity, Row-Store performs well. But, in decision making applications, data is accessed in bulk at the granularity of an attribute. The need for Column-oriented database arose from the need of business intelligence needed 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. This work discusses the best method for implementing column-store on top of Row store in PostgreSql along with successful design and implementation of the same. Performance results of our Column-Store are presented, and compared with that of traditional Row-store results with the help of TPC-H benchmark. We also discuss about the areas in which this new feature could be used so that performance will be very high as compared to Row-Stores.

3.2 OBJECTIVE

Our aim is to implement Column-store on top of Row-store in PostgreSql to improve performance of Read-queries. PostgreSql is basically an open-source row-oriented database which does not have any feature of storing data column-by-column. We are enhancing PostgreSql to have this feature so that for decision making applications performance of Read queries will be higher than that compared with row-oriented database system. Modifications will be done in such a way that all queries processing for Column-Store table will be transparent to the user.

RESEARCH METHODOLOGY

POSTGRESQL COLUMN – STORE DESIGN

In this section we give a brief idea about how Vertical partitioning approach [1, 3] is implemented in PostgreSql for having Column-store feature. There are a lot of DDL and DML queries implemented and the required design modifications into PostgreSql are proposed as follows:

4.1 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 col store. A new keyword col store will be included in the create query. So, a new query would look like Create colstore table table-name (attr1 datatype, attr2 datatype ...);
View is basically a stored query. It does not take much space as only view

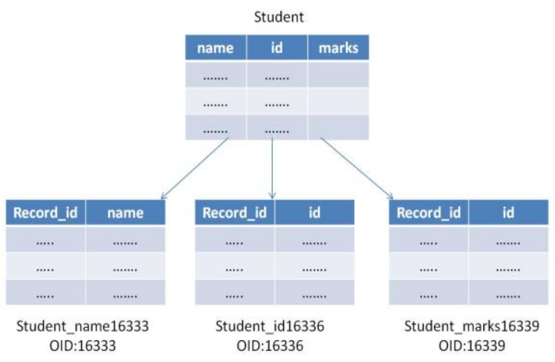

Figure 4.1: Creating Colstore Table

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 view is _red 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.

4.2 META DATA FOR THE COLUME – STORE RELATION

Now that internal tables, sequence has all been created, there is a need to store meta-data 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 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 figures 4.2 and 4.3. When a column-store table is created, respective values are entered into these tables. Therefore, whenever user wants to _re 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 [29, 31, 32]. These tables will be searched on the basis of a unique key as explained earlier. In chapter 2, we had seen various system catalogs for describing, Row-Store relation, functions, operators etc. Similarly, for Column-Store we define new system catalog for describing a Column-Store relation.

i. pg map: For every attribute in Column-Store relation, an internal Row-Store table is created. Therefore, for each attribute of Column-Store table, there will be a mapping stored between [Colstore table, attribute] and corresponding row-store table created. So, for each attribute relation (Row-Store relation) created, a row is created containing its object identifier, name, corresponding attribute number, name of Column-Store relation.

ii. Pg_attrnum: For each Column-Store table, there will be an entry in this system table which will store number of attributes, corresponding object identifier of sequence created, view created.

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 4.2: System Table pg map for storing mapping between relation and internal tables

4.3 INSERTING DATA INTO COLUMN – STORE TABLE

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].

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

Figure 4.3: System Table pg_attrnum for storing meta-data of Column-Store Table

In Row-Store this list is directly inserted into required table.But, for Column-Stores, this list is broken into separate column values and values are passed to the corresponding tables along with the next unique sequence value generated. Since each internal table has two attributes <record id, attribute>, value of attribute column is given as input by user but input for record id is generated internally by using unique sequence generated for each relation. Whenever insert is performed, sequence value is incremented each time. 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 and processed internally.

4.4 ALTERING THE COLUMN – STORE TABLE

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 and system tables will be updated accordingly. Similar is the case with dropping a column.

4.5 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.

4.6 SELECTING DATA FROM COLUMN – STORE TABLE

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. 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 4.4 explains various stages of a column-store table on which select query is applied. This will make the scenario clearer.

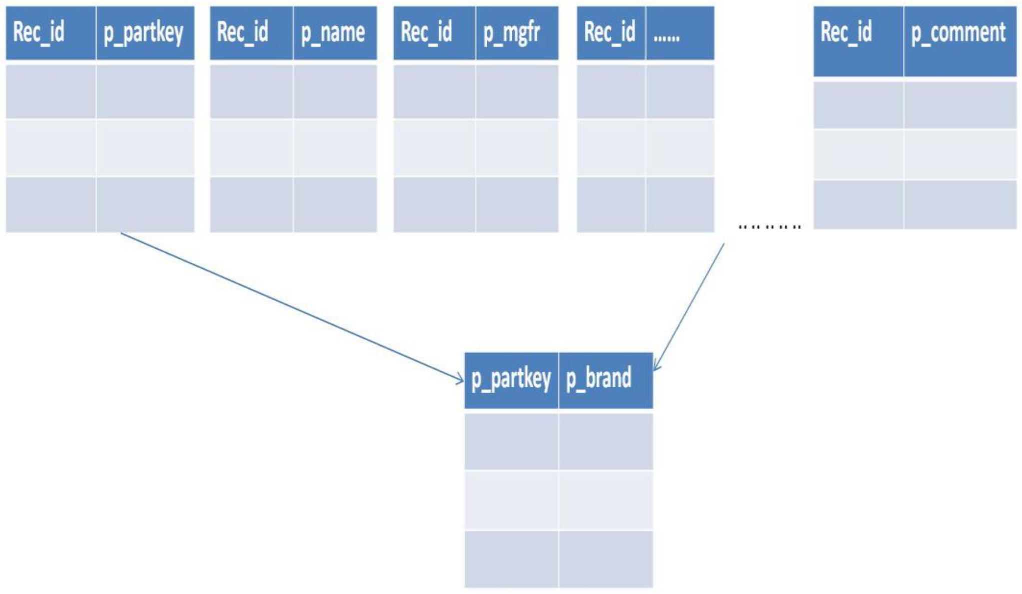

Figure 4.4: Taking join of internal tables

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;
```

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.

4.7 SUMMARY

In this section, design and architectural details of our Column-Store implementation in PostgreSql are described. Create table query is modified as required. Insertion is quite slow as compared to Row-Store but for large datasets, good performance of insert is not required. Data is loaded in warehouse and is not time critical query. Select query is modified for Column-Store in such a way that its performance is improved by a large factor for analytical queries.

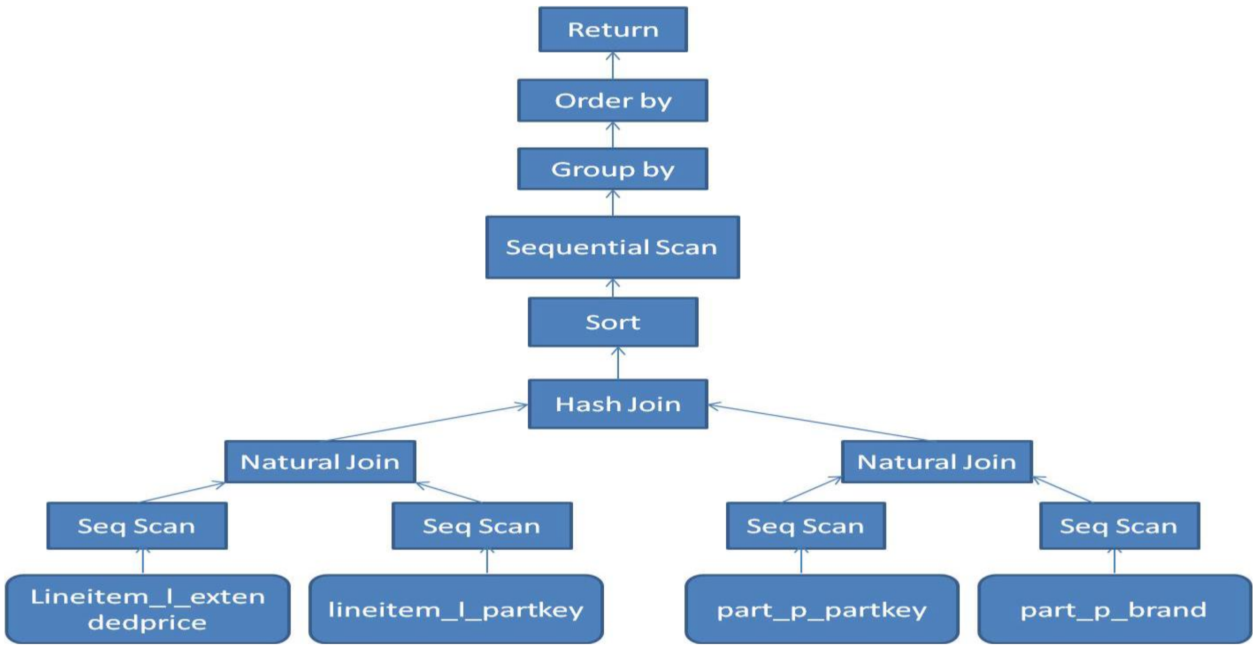

Figure 4.6: Plan Tree for SELECT query in Col-Store

EXPERIMENTAL RESULT

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. We have mentioned that point in chapter 3. For evaluating the performance of our implementation, we use TPC-H benchmark [14, 15, 16, 25]. Dataset size we have taken for our analysis is 5000 tuples per table. The schema diagram of their dataset is as given below:

For each attribute of each table shown in the figure 5.1, 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. 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.

This table 5.1 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 have 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. The graph is plotted as shown in figure 5.2. 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 [20] 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.

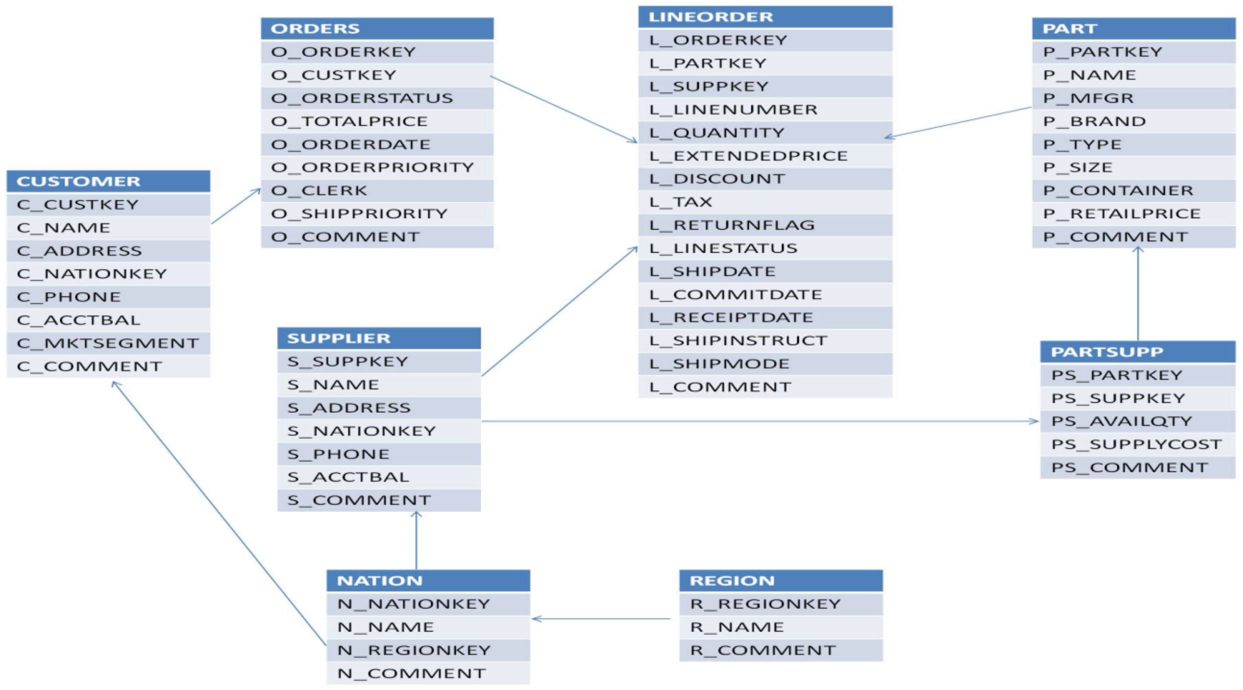

Figure 5.1: E-R Diagram of TPC-H Benchmark Dataset

Table 5.1: 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

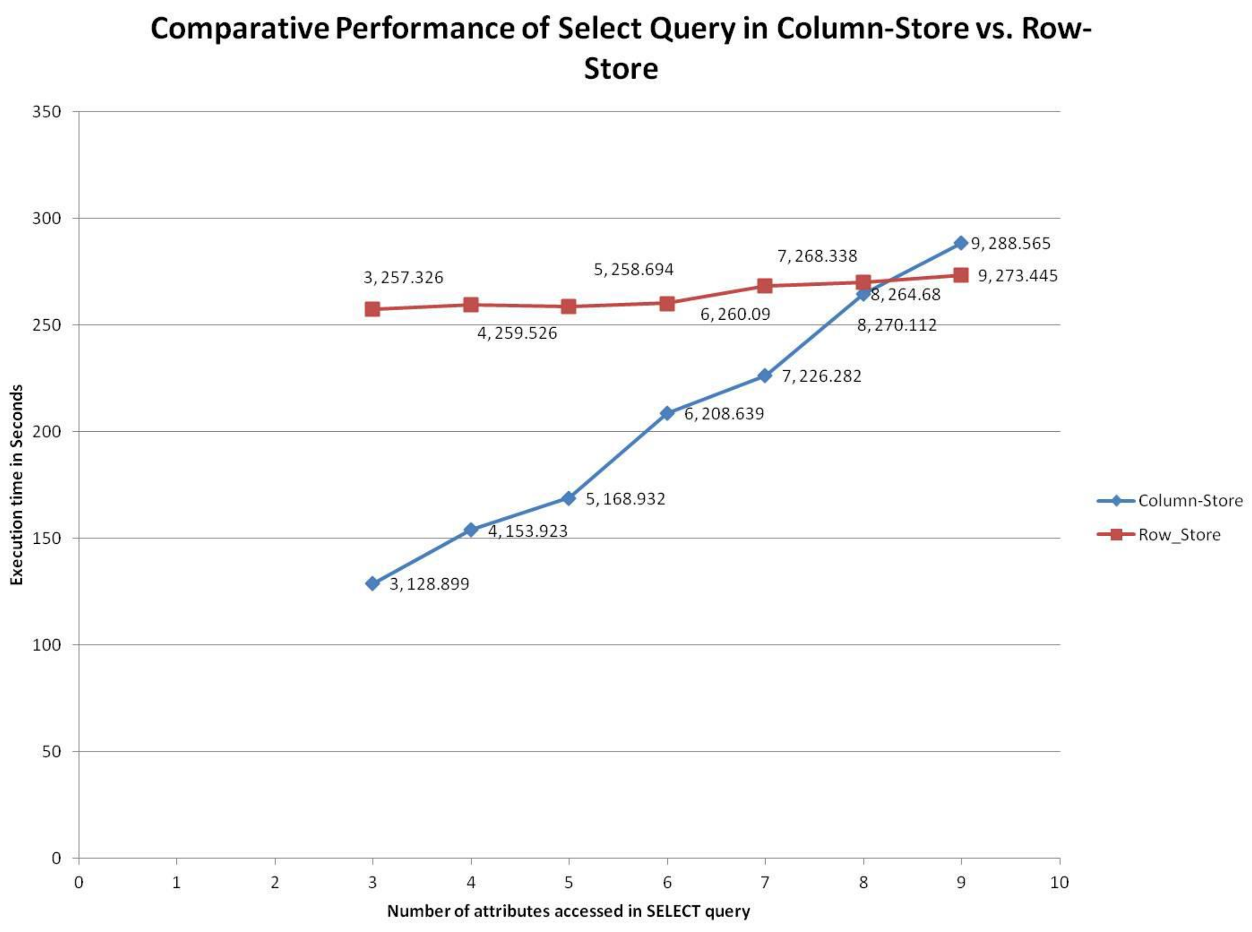

Figure 5.2: Comparison of Column-Store vs. Row-Store

We have considered following 10 queries for evaluating our performance.

1. Select
   l returnag,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 returnag;
```
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

CONCLUSION AND FUTURE WORK

6.1 Conclusion

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 chapter 3. The main focus of this thesis was to implement Column-Store in PostgreSql in a systematic manner so that performance of read-oriented queries becomes better than Row-Store. We have compared Row-Store and Column-Store performance on TPC-H benchmark. The results clearly show that Column-Stores are better than Row-Stores in the cases we expected.

6.2 Future Work

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.

REFERENCES

[1]. Daniel J. Abadi, Samuel R. Madden, Nabil Hachem, Column-Stores vs. Row-Stores: How Different Are They Really? Vancouver, BC, Canada, SIG-MOD08, June, 2008.

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

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

[4]. D. J. Abadi, S. R. Madden, and M. Ferreira, Integrating and execution in column-oriented database systems, SIGMOD, pages 671-682, 2006.

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

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

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

[8] S. Harizopoulos, V. Liang, D. J. Abadi, and S. R. Madden, Performance trade-offs in read-optimized databases, VLDB, pages 487498, 2006.

[9] G. Graefe, Efficient columnar storage in b-trees, SIGMOD Rec., 36(1), pages 36, 2007.

[10] A. Weininger, Efficient execution of joins in a star schema, SIGMOD, pages 542545, 2002.

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

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

[13] R. Ramamurthy, D. Dewitt, and Q. Su A case for fractured mirrors In VLDB, pages 89 101, 2002.

[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/.