Data Modeling for Cassandra

ABSTRACT: *Mesmo que um dos campos (resumo ou abstract) seja maior do que o outro, o espaçamento simples será mantido, não sendo alterado o "kernel" entre as letras. Utiliza-se sempre a justificação, mas evitam-se diferentes espaçamentos entre as palavras, ficando possível quebrar palavras eventualmente e* ***APENAS*** *quando o espaçamento se tornar como o da linha acima: excessivamente grande. O alinhamento fica vinculado aos títulos do campo e ao maior dos textos (abstract ou resumo). Neste caso, este ficou maior, portanto percebe-se que sempre o RESUMO se alinha com o ABSTRACT, como sempre se deve alinhar PALAVRAS-CHAVE com KEYWORDS. Este exemplo tem 130 palavras.*

KEYWORDS: *Template. Engineering. Keywords. Alinhar os títulos de Palavras-chave e Keywords*

1. Intruduction

NOSQL came as an alternative to supply a demand for scalability, availability and fault tolerance. It emerged as an alternative to Relational database management systems. Although there are already many different NOSQL solutions already available, to the best of our knowledge, there are barely no guidelines for data modeling for such solutions. Moreover, modeling guidelines for Relational DBMS are not adequate for NOSQL approaches. While Relational modeling focuses on creating distinct relations, and references between them, avoiding data redundancy, NOSQL approaches stimulate data aggregation and duplication [1].

NOSQL database systems are classified according to the way they store and access data. For each of these approaches, there may be a different way to model the data. There are already some initiatives to fill the gap of data modeling for these different NOSQL approaches [2][3][4]. In this last work, the authors propose data modeling for the HBase system, which is classified as a Column-oriented system. Cassandra is also classified as a column oriented system, but after running some initial tests, we have seen that Cassandra and HBase systems work differently, and that the data modeling process for them can not be the same.

Therefore, in this paper, we propose some guidelines to help the user on data modeling for the Cassandra NOSQL system. We assume that there is an initial schema that consists of a single table, which results from a denormalized relational schema. Additionally, we also consider a set of pre-defined queries on this schema. The main contribution of the proposed guidelines consists of a heuristic. It is formalized as an algorithm that ranks possibilities of primary keys and materialized views based on query demands, aiming at the reduction of the number of views to be created and maintained.

In order to evaluate the proposed algorithm we used a well-known benchmark for OLAP applications. The idea was to evaluate performance gains among different data modellings for the same relational schema, including the one generated by the proposed algorithm.

This paper is organized as follows. The second section presents some concepts and technological details that were used to develop the proposed method. Section 3 describes some related work, highlighting the differential of our approach, which is presented in section 4. Sections 5 and 6 present the method evaluation scenario, the results and their discussion. Finally, section 7 concludes the work pointing to future directions.

2. NOSQL databases

One of the main characteristics of NOSQL DBMS is the facility to deal with data scalability, while maintaining a good performance [1]. Some of these systems use resources such as parallel architectures, data sharding and replication in order to gain performance. On the other hand, in these systems, maintaining consistency may be an issue, and thus, many of them do not provide the ACID properties. They usually provide what is called eventual consistency, which allows that the replicas and/or shards are not fully consistent all the time, but at some future point in time. This disadvantage is acceptable in order to get benefits such as availability and performance.

Column-oriented databases has as its main characteristic, to store tables in columns instead of rows. In a relational database, each tuple (with all its attribute values) is stored together. Thus, to retrieve the values of part of these attributes, it is necessary to retrieve the entire tuple, affecting directly the query execution time [5]. Differently, in a columnar database, the attribute values of a tuple may be stored separately in columns. For instance, a column may store all values of a single attribute and their corresponding identifying keys. It also may be organized in families of columns, where each family may store a subset of attributes that compose the tuple stored in the database. With this approach, retrieving some attributes does not bring the whole tuple, resulting in a better performance if compared to relational databases.

Therefore, column-oriented databases tend to perform better than relational databases, especially when executing aggregation queries over some attributes. Cassandra and HBase are examples of column-oriented databases. In this work we chose to work with Cassandra for its popularity among the column-oriented databases1.

In Cassandra, data is distributed over all nodes of a cluster, according to the partition keys defined on each table. When a node is added or removed, all its data is automatically distributed over the other available nodes. If a node fails it will be replaced instantly. Because of this, it is no longer necessary to calculate and assign data to each node. Cassandra's architecture

[1]https://db-engines.com/en/ranking

is known to be peer-to-peer (it partitions tasks or workloads among peers equally) and overcomes master-slave limitations by providing high availability and massive scalability. Data is replicated over multiple nodes in the cluster. Failed nodes are detected by gossip protocols (peer-to-peer communication protocol in which nodes periodically exchange state information about themselves and about other nodes they know about) and those nodes can be replaced instantly [6].

In Cassandra, data is indexed by the primary key, that is composed by a partition key and clustering keys. The primary key leads to the row where the data is stored, and in each row the data is divided into columns and column families. Each column in Cassandra has a name, a value and a timestamp. Both the value and the timestamp are provided by the client application when data is inserted.

Recently, in Cassandra 3.0, the concept of Column Family is also called Table. Unlike columns, the Tables are not dynamic and must be previously declared in a configuration file. They are the unit of abstraction containing keyed rows which group together columns of highly structured data. Tables have no defined schema of column names and types supported. Lastly, tables are grouped into Keyspaces. These Keyspaces can be compared to Schemas in a relational database.

In Cassandra's peer-to-peer model, each node exchanges information across the cluster every second. A sequentially written commit log on each node captures write activity to ensure data durability[6]. Data is then indexed and written to an in-memory structure called memtable, which resembles a write-back cache. Once the memory structure is full, the data is written to disk in an SSTable (sorted string table) data file (a file of key/value string pairs, sorted by keys). All writes are automatically partitioned and replicated throughout the cluster. When a read or write request is made, any node in the cluster is able to handle it. Through the key, the node that answered the request can know which node possesses data information.

Cassandra also enables the creation of materialized views. The concept is the same as in relational databases. The idea is to store the data according to some pre-defined query, aiming to improve performance. Each table may have one or more materialized views.

Typically, the disadvantages on the usage of materialized views are: an extra storage cost and the time cost for the maintenance of consistent views, as updates occur in the base table. In Cassandra, when the user updates the base table, the views will be updated automatically, generating a lower maintenance cost at the user level.

3. Related work

This section summarizes and compares the papers presented as shown in Table I. In order to fill the data modeling gap for NOSQL databases, some works chose to focus on a specific performance demanding application: the OLAP application [7]. Typically, it is based on the multidimensional model, which includes the fact and the dimension concepts. These concepts are represented in the relational model as a star schema, where the fact corresponds to a table, as well as each dimension. Each fact tuple refers to tuples in each dimension.

The transformation of the multidimension conceptual model directly to the NOSQL logical model is proposed by [8]: each star schema is mapped into a single table. The fact is transformed into a column family, where each measure is a column. Each dimension is transformed into a column family, where each attribute is a column. In addition, all aggregation possibility for that schema is also similarly mapped into a separate table, as materialized views. However, in this work there is no intention in selecting a subset of those views, which implies high costs with respect to storage and view maintenance.

A complementary study over NOSQL Multidimensional Modeling [9], presents three different ways of logical modeling in a NOSQL columnar database. The first one, named normalized logical approach (NLA), adopts a vertical fragmentation of a denormalized star schema, and stores the fact and each dimension into different tables. The denormalized logical approach (DLA) maps the star schema into a single table, which stores the fact and dimensions all together. The third one, called denormalized logical approach using column families (DLA-CF), is similar to the DLA approach, but the dimensions and the fact are mapped, each one, to a different column family.

[10] proposes a Cassandra data modeling based on the queries. It also defines modeling principles, mapping rules e mapping patterns. This methodology prioritizes the applications workflow and its access patterns. The normalization is removed, implying on data redundancy and views usage over joins. Because of those differences, it is necessary a paradigm change on modeling based purely on entities and relationships, to a modeling based on queries.

On [4] three modelings are used over HBase to evaluate the performance of OLAP Queries. In addition to the DLA e DLA-CF modelings presented on [9], which are denominated SameCF e CNSSB, respectively, the authors propose the FactDate modeling as the third alternative. It follows the same idea of the DLA-CF alternative, but it gathers in the same column family, the fact and the date dimension. Experiments with those three modeling alternatives showed that the FactDate alternative has better performance on queries that use few dimensions, i.e., the Date dimension and one other. On the other hand, the SameCF alternative has better performance on queries that use a higher number of dimensions.

These related works reported on modeling performance of column oriented databases, but they do not approach how to deal with the data distribution nor how to select materialized views to get the best performance of the database. This is a crucial factor to execute queries properly over Cassandra. In this work, we present a set of guidelines and a heuristic to help the modeler on selecting the best distribution keys (partition and clustering keys) and a set of materialized views for Cassandra database system.

TABLE 1: RELATED WORK COMPARISON.

Work	Modeling	Views	DBMS

[8]	Conceptual/Logical	-	MongoDB and HBase
[9]	Conceptual/Logical	-	HBase
[10]	Relational/Logical	-	Cassandra
[4]	Logical	Views with an external application	HBase

4. INITIAL EXPERIMENTS

As previously mentioned [Chevalier et al. 2015], in order to perform data modeling, it is a good practice to start with a conceptual schema of the data and then proceed to the data modeling, where the conceptual schema elements are mapped into a logical/physical schema of a specific DBMS.

Once a logical schema is designed, it is important to know the typical/critical queries that should be supported by the application. From these, it is possible to define logical/physical schema alternatives(candidate schemas) to the database. In the case of a column-oriented DBMS, the choice of such schema is not trivial. A careful analysis is necessary to identify the most appropriate logical/physical schema according to the application demand.

Thus, in short, data modeling consists of two steps. The first step is concerned with conceiving a first version of a logical schema. Then, the second step focuses on performance issues and on attending application demands, such as to address a set of critical queries. In this work, we address just the second step for the Cassandra DBMS. We assume that an initial logical schema is already available, and then, we apply a set of heuristics in the form of an algorithm.

In order to develop such algorithm, we performed some initial experiments for a typical OLAP application, which were detailed in Section 4. To guarantee that we would start with the best initial logical schema, we explored the CNSSB datasets [11] and queries, taking into account three logical schemas, as proposed in [9] and [4]: SameCF, DLA-CF and FactDate. Based on the results of such experiments, we found out that the best initial logical schema was the SameCF schema.

Then, assuming the SameCF schema as the initial schema, we observed the performance gains while using partition and clustering keys alternatives, as well as, while querying on materialized views. This discussion is presented in Section 4.2. Then, in Section 5, we identify some heuristics for choosing those keys and materialized views, to address most of the queries and reduce the set of materialized views. Finally, these heuristics were formalized in the algorithm presented in Section 5.

4.1 Initial logical schema definition

All three models (SameCF, DLA-CR and FactDate) were populated with data generated from the DBGen application of the CNSSB [11]. After populating the models, the thirteen CNSSB queries, organized in 4 typical sets, Q1, Q2, Q3 and Q4, were executed five times on each model to measure their performance average.

As it can be seen in the graph of Fig.1, the queries had a similar behaviour in the DLA-CF and FactDate models, with the exception of the Q1 set of queries that benefits from the fact that it does not require joins to perform the queries. The SameCF model is able to obtain better performance in sets Q2, Q3 and Q4, showing that the use of joins in Cassandra implies in a worse performance of the query. Analyzing the query behavior, we can note the influence of the Partition Key and Clustering Keys. Since Cassandra partitions the data and distributes them among the nodes according to the Partition Key, queries that use its attributes as filters usually perform better. However, these queries performance may be reduced if they also use other attributes

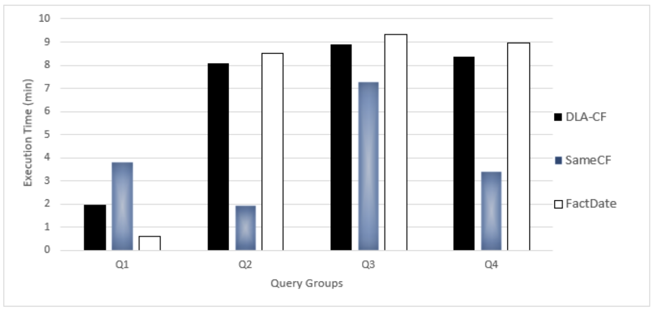


Fig. 1 – Query performance on each model

as filters. Therefore, query performance is highly dependent on the attributes used and on the fact that they are part of the Partition or Clustering Keys.

Due to the fact that the SameCF model had an average performance superior to the DLA-CF andFactDate, this model was chosen as the initial logical model. From this model, an evaluation of theuse of materialized views is presented in the following section. Then, a heuristic is defined based onthe use of materialized views according to the SameCF model.

4.2 Experiments results

Analyzing the SameCF model, we can see that the way the attributes are arranged in the PrimaryKey directly influences the queries performance. A query that filters on an attribute that belongs to the Partition Key will perform well as opposed to a query that filters on an attribute that is positioned at the end of the Clustering Key. The Clustering Key sorts the records of a partition according to each attribute defined in it, that is, it is an ordered list of attributes that determines the order of the records in the disk. From the graph of Fig.2, it can be seen that in the set of queries QG1, there was a significant variation on the performance of the queries. Queries 1.1 and 1.3 performed well because their filter attributes were those used for the formation of the Partition Key (year) and the Clustering Key (discount, quantity), in this case the first attributes. On the other hand, query 1.2 had the worst performance for two reasons: first, due to the fact that it does not include the Partition Key attribute as a filter, and second, because the attribute yearmonthnum is an attribute unfavorably positioned in the Clustering Key, that is, it is not an attribute that is positioned right at the beginning of the Clustering Key, impairing filter performance.

Regarding the QG2 set, queries showed the best average performance, with times very close to each other. On the other hand, the QG3 was the worst set. Interestingly, query 3.1 had one of the best performances in relation to all queries of all sets. This is explained by the fact that the attributes of the filters in this query are either an attribute of the Partition Key (year) or belong to the first positions of the Clustering Key (supregion, region). Queries 3.2 and 3.3, although using a filter based on the Partition Key attribute (year), the other attributes used as filters belong to unfavorable positions in the Clustering Key. Query 3.4 does not use the Partition Key (year) attribute as a filter, but as an ordering/grouping attribute, and in addition, also filters by unfavorable positioned Clustering Key attributes.

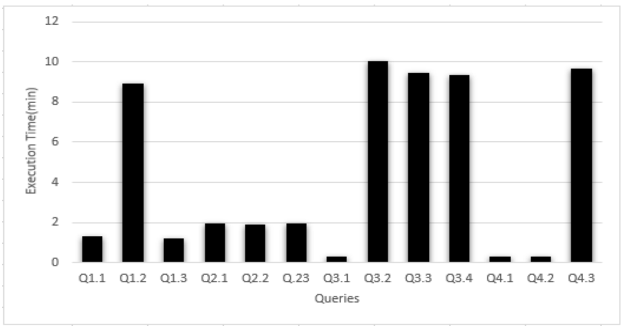


Fig. 2 – Queries performance in SameCF model

Finally, when analyzing the QG4 set, we note that queries 4.1 and 4.2 had a good performance. This is probably due to the fact that those queries filter using the first attribute of the Clustering Key. The use of the Partition Key (year) in query 4.2 may have led it to perform better. Query 4.3 had the worst performance because it used only unfavorable positioned attributes of the Clustering Key. However, interestingly, it uses the Partition Key (year) as filter, which shows that only using the Partition Key will not necessarily ensure the best performance for a query.

As a way to improve the performance of the queries, it is possible to create materialized views from a table in Cassandra. This allows the choice of attributes that will be part of each view and the reorganization of the Partition and Clustering Keys for that view. Using the criterion proposed by [12], we chose the worst performance set of queries (Q3) as the basis for the creation of the materialized view. The graph of Fig.3 shows the performance of these queries on the materialized view and compare them to the base table. When performing the four queries of group Q3 on the created materialized view, we can observe that query 3.1 loses performance. This is explained by the use of the modified Clustering Key. In addition, this is also the reason for the reduction of the execution time of the other queries. That is, query 3.1 can continue to be executed directly on the base table and the other queries would do better if executed on the materialized view.

We can conclude that doing joins through applications is not ideal, since there is loss of performance. That said, the ideal for modeling in Cassandra is to denomarlize the data and store them in a single table.

5. Guidelines for cassandra

Usually, at the logical modeling phase of a database design, the idea is to depart from a conceptual DMBS independent view of the application, and arrive at a DBMS dependent schema of it. In the context of a relational database, it consists on designing a set of tables and attributes, whereas in the context of a columnar database approach, this means to choose which column families should be created. Assuming an initial columnar logical schema is already chosen, the next phase of a database design is the physical schema design, which focuses on the performance of the database for a given set of applications, queries and activities. In the Cassandra case, this includes to define materialized views.

Based on the results of the initial experiments, reported on subsection 4.2, it was possible to devise some initial guidelines, concerning the choice of a logical/physical schema for Cassandra DBMS.

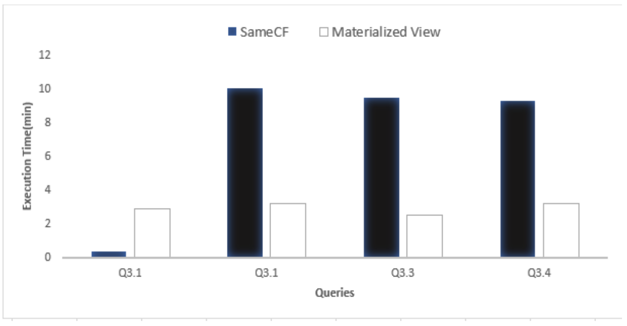


Fig. 3 – Materialized view performance

Guideline 1. Denormalize the logical schema.

Taking into account an initial conceptual-logical mapping as the logical initial schema, it is recommended to denormalize this schema, in such a way that it is able to answer a set of critical queries or demands.

Guideline 2. Define the primary key attributes.

Based on the set of critical queries, analyze the most frequently used attributes on the query constraints and then choose the ones to form the primary key.

Guideline 3. Define a set of materialized views.

The idea is to group queries, according to their common attributes, i.e., attributes that are present in the selection and in the filtering clauses. These groups are the basis for the choice of the materialized views. Once defined, these views may be created.

Guideline 4. Define a query redirection policy.

Given a set of materialized views, a query redirection mechanism could benefit from those views, by rewriting and redirecting the query to the materialized view that can probably provide the best performance for that query.

In order to help the designer on guidelines 2 and 3, in this section we present Algorithm 1. It is based on Cassandra query execution constraints, previously presented in Section 2. It takes as input an initial denormalized logical schema, as suggested by guideline 1, and a set of pre-defined critical/typical queries, and its output may be used as the input for guideline 4.

Cassandra supports query execution only directly into a partition, that is, it demands an equality constraint over the partition key. This is the first premise adopted to suggest a materialized view. Other premise is that constraints that are not equality based can only appear once in each query. In the case of constraints with the IN clause, it may appear along with another inequality constraint, and this must be the last one to be applied in the query expression, and only in the Clustering Key.

Therefore, based on these initial premises, Algorithm 1 finds possible combinations of attributes to form the primary key for each view. The main idea is to identify groups of queries, whose constraints use attributes in common, and for each group, suggest a reduced number of materialized views, with their respective primary keys. In addition, for each view, suggest the set of queries that are associated with it, i.e., indicate to which materialized view, each query should be redirected to (or rewritten to).

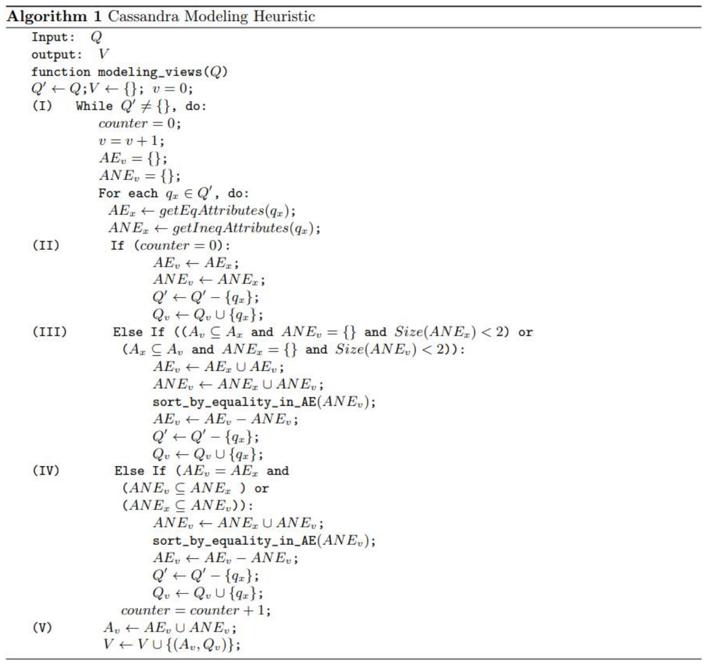
The following variables are used in Algorithm 1(Fig.4):

- Q: set of critical/typical queries to be executed on Cassandra;
- v: index of the view under construction, $1 \leq v \leq |Q|$;
- AEx: set of equality attributes, i.e., attributes that are involved in equality based constraints on a query qx expression, which will compose the set of attributes AEv of the primary key of the associated materialized view v;
- AEv: set of equality based attributes of view v;
- ANEx: set of non-equality attributes, i.e., attributes that are involved in inequality based constraints on a query qx expression, which will compose the set of attributes ANEv primary key of the associated materialized view v;
- ANEv: set of inequality based attributes of view v;
- Ax: set of attributes of query qx, where $Ax = AEx \cup ANEx$;
- Av: set of attributes that will compose the primary key of the associated materialized view, where $Av = AEv \cup ANEv$;
- Qv: set of queries, that will be addressed by the view v;
- V : set of materialized views to be generated, formed by a set of pairs (Av,Qv);
- Q´ : set of queries that are not supported by none of the existing materialized view in V ;

In step I, Q´ is initialized with the complete set of queries, then the algorithm iterates over the Q´ set, until it becomes empty. Each iteration on Q´ (step I) aims to build a new view v and its corresponding primary key. Every query qx from Q´ is analyzed by the algorithm with respect to its attributes involved in equality and inequality constraints, until the key for the view under construction is formed. Depending on the evolution of the key under construction for the view of the moment (v), query qx may be treated in steps II, III and IV. If it satisfies the constraints for one of these steps, it is included in the set of queries (Qv) that will be addressed by view v and removed from the Q´ set.

The first query of each iteration on Q´ is always treated within step II, where the sets of equality/inequality based attributes are initialized for the view of the moment. Once the first attributes for a view are defined, then the next Q´ queries are treated by the following steps, depending if they have attributes in common with the view under construction. Step III will add queries that bring to the view at most one attribute involved in an inequality constraint, and that have attributes in common with the set of attributes of the view (Av ⊆ Ax or Ax ⊆ Av). Step IV deals with queries that bring to the view more than one attribute involved in an inequality constraint. In this case, the query under analysis need to have the same equality constraint attributes. Queries that did not fit within any of the previous steps, remain in Q´ and will be analyzed again for a new iteration, i.e., for the creation of a new materialized view.

Finally, at the end of each while iteration, in step (V), the set of attributes that compose the view Av is created and the set of views V is incremented with the new pair (view (Av), query (Qv)).



Algorithm 1 Cassandra Modeling Heuristic

```
Input:  $Q$
output:  $V$
function modeling_views($Q$)
$Q' \leftarrow Q; V \leftarrow \{\};\ v = 0;$
(I)   While $Q' \neq \{\}$, do:
          $counter = 0;$
          $v = v + 1;$
          $AE_v = \{\};$
          $ANE_v = \{\};$
          For each $q_x \in Q'$, do:
            $AE_x \leftarrow getEqAttributes(q_x);$
            $ANE_x \leftarrow getIneqAttributes(q_x);$
(II)        If ($counter = 0$):
                $AE_v \leftarrow AE_x;$
                $ANE_v \leftarrow ANE_x;$
                $Q' \leftarrow Q' - \{q_x\};$
                $Q_v \leftarrow Q_v \cup \{q_x\};$
(III)       Else If (($A_v \subseteq A_x$ and $ANE_v = \{\}$ and $Size(ANE_x) < 2$) or
             ($A_x \subseteq A_v$ and $ANE_x = \{\}$ and $Size(ANE_v) < 2$)):
                $AE_v \leftarrow AE_x \cup AE_v;$
                $ANE_v \leftarrow ANE_x \cup ANE_v;$
                sort_by_equality_in_AE($ANE_v$);
                $AE_v \leftarrow AE_v - ANE_v;$
                $Q' \leftarrow Q' - \{q_x\};$
                $Q_v \leftarrow Q_v \cup \{q_x\};$
(IV)        Else If ($AE_v = AE_x$ and
             ($ANE_v \subseteq ANE_x$ ) or
             ($ANE_x \subseteq ANE_v$)):
                $ANE_v \leftarrow ANE_x \cup ANE_v;$
                sort_by_equality_in_AE($ANE_v$);
                $AE_v \leftarrow AE_v - ANE_v;$
                $Q' \leftarrow Q' - \{q_x\};$
                $Q_v \leftarrow Q_v \cup \{q_x\};$
            $counter = counter + 1;$
(V)       $A_v \leftarrow AE_v \cup ANE_v;$
          $V \leftarrow V \cup \{(A_v, Q_v)\};$
```

Fig. 4 – Algorithm 1

6. EXPERIMENTS WITH MATERIALIZED VIEWS IN CASSANDRA

In this section, we present the results of the experiments carried out to evaluate the heuristics proposed in the previous chapter. The purpose of these experiments is to compare the performance of three logical models with different input rates of read and write operations, one of which uses the proposed heuristics.

The experiments were carried out in a computational cluster with four nodes, each with 158 GB of RAM, 64 CPUs of 2.4 GHz. The operating system of each node is CentOS, with the DBMS Cassandra (version 3.0). The database used during the experiments was generated by the dbgen of the CNSSB [11] without any post-treatment, since it is already generated on a single denormalized table in a CSV file.

6.1 Heuristic application

Using the CNSSB schema with its thirteen queries, the proposed heuristic created a set of nine materialized views. Each materialized view has a different primary key that will allow execution of one or more queries. None of these queries could be performed by more than one materialized view. Therefore, a minimum number of materialized views was generated to meet all the specified queries.

Next, is exemplified the generation of a materialized view formed by the following two queries:

1. select year, nation, revenue, supplycost from cnssb.nlineorder where region = ’AMERICA’ and suppregion = ’AMERICA’ and mfgr in (’MFGR#1’,’MFGR#2’)
2. select year, nation, revenue, supplycost from cnssb.nlineorder where region = ’AMERICA’ and suppregion = ’AMERICA’ and year in (1997,1998) and mfgr in (’MFGR#1’,’MFGR#2’)

Both queries have three attributes in common in equality filters: region, suppregion and mfgr. These attributes will initially compose the primary key, one of them as the partition key and the other attributes will begin the clustering key. The query (2) also has an equality filter over the year attribute. This attribute must also be included in the clustering key, since it is not used by the query (1). The composition of the primary key of the materialized view that fits the two queries is:

- Partition Key: region
- Clustering Key: suppregion, mfgr, year

The next example is based on a group of queries that use equality and inequality filters:

1. select discount,quantity from cnssb.nlineorder where year = 1993 and quantity < 25 and discount between 1 and 3
2. select extendedprice,discount as revenue from cnssb.nlineorder where year = 1994 and yearmonth = 'Jan1994' and quantity between 26 and 35 and discount between 4 and 6
3. select extendedprice, discount as revenue from cnssb.nlineorder where year = 1994 and weeknuminyear = 6 and quantity between 26 and 35 and discount between 5 and 7

These queries, if executed as presented, will not run in Cassandra since this DBMS restricts comparisons of "greater than" and "less than" types only to the last field of the key to be filtered. Because of this, these three queries were adapted by changing the "quantity

1. select discount,quantity from cnssb.nlineorder where year = 1993 and yearmonth in ('Jan1993', 'Feb1993', 'Mar1993', 'Apr1993', 'May1993', 'Jun1993', 'Jul1993', 'Aug1993', 'Sep1993', 'Oct1993 ', 'Nov1993', 'Dec1993') and quantity in (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24) and discount in(1, 2, 3)
2. select extendedprice,discount as revenue from cnssb.nlineorder where year = 1994 and yearmonth = 'Jan1994' and quantity in (26, 27, 28, 29, 30, 31, 32, 33, 34, 35) and discount in(4, 5, 6)
3. select extendedprice, discount as revenue from cnssb.nlineorder where year = 1994 and weeknuminyear = 6 and quantity in (26, 27, 28, 29, 30, 31, 32, 33, 34, 35) and discount in(5,6,7)

Regarding the application of the heuristic on these three queries, another adaptation was made in query (1) so that it could be served by the same materialized vision that the query (2). Originally, query (1) does not have the filter on the attribute yearmonth, however, is a possible filter to be included in order to bring the same result without the filter, and adding few restrictions on it. The same cannot be done to the query (3) on the weeknuminyear filter. While the yearmonth filter represents the months of one year (maximum of twelve restrictions), the weeknuminyear filter represents the number of the week of a year (approximately fifty-two restrictions). The last filter has a very high amount of restrictions, escaping much from a scenario closer to a real-world application. In this way, the heuristic generated two materialized views, one to answer queries (1) and (2) and another to answer the query (3).

The queries (1) and (2), after the mentioned adaptations, have equality filter over the following attributes: year and yearmonth. These attributes will compose the primary key of the materialized view, with one being the partition key and the other being the clustering key. These queries also have two non-equality filters: quantity and discount. Both attributes will be included in the clustering key, right after the equality attribute that was included in it. The composition of the primary key of the vision that will support these queries is:

- Partition Key: year
- Clustering Key: yearmonth, quantity, discount

The query (3) should have a materialized view that will only serve it. The key of this materialized view will be composed of the equality attributes: year and weeknuminyear, with one of them being partition key. And also by the non-equality attributes: quantity and discount, which will compose the clustering key, just after the equality attribute that was included in it. The composition of the primary key of the vision that will support this query is:

- Partition Key: year
- Clustering Key: weeknuminyear, quantity, discount

Similar to the examples cited above, nine views where created from the application of the proposed heuristic on all of the thirteen queries, as shown in table 2. After the implementation of the three models in Cassandra, the five workloads were executed in each modeling mode to evaluate the proposed heuristic and verify its performance comparing with the other models.

6.2 Evaluation scenarios

To evaluate the performance of the use of the heuristic in Cassandra, three different scenarios were used. For each scenario, a keyspace was created with a replication equal to 2, that is, each record was replicated twice in the cluster.

1. Scenario 1: Heuristic generated model, with a minimum of materialized views that can handle all the queries.
2. Scenario 2: Modeling with a materialized view for each query.
3. Scenario 3: Modeling with a table for each query.

Five workloads were developed to evaluate different aspects of each scenario. Each workload represents a combination of read and write operations. The objective is to analyze the performance of the evaluated models considering different possibilities of operations that are made over a database.

TABLE 2: MATERIALZED VIEWS CREATED FROM THE HEURISTIC

View	Partition Key	Clustering Key	Supported Queries
V1	suppregion	region,mfgr,year	4.1, 4.2
V2	suppcity	city,year,yearmonth	3.3, 3.4
V3	suppregion	brand1	2.2, 2.3

V4	year	yearmonth,quantity,discount	1.1, 1.2
V5	year	weeknuminyear,quantity,discount	1.3
V6	suppregion	category	2.1
V7	suppregion	region,year	3.1
V8	year	suppnation,region,category	4.3
V9	suppnation	nation,year	3.2

All workloads are based on the same dataset (CNSSB), to ensure that they are executed under the same conditions. In this way, it is expected to identify which scenario provides the best performance for each workload.

The following workloads were used:

- Only read operations.
- Most read operations: 75% read and 25% write operations.
- Read and write operations equal: 50% read and 50% write operations.
- Most write operations: 25% read and 75% write operations. —Only write operations.

These workloads were developed using the CNSSB dataset file as a basis for writing operations and CNSSB queries of reading operations. The execution of each workload was performed through a program written in the Python language that performs all operations in parallel.

6.3 Performance analysis

Initially, the performance of the three scenarios is verified in a workload with read operations only. The execution performance of the three scenarios is very close, and took around 10 minutes. Considering that each scenario is modeled to attend all the proposed queries and Cassandra itself ensures that a query will be executed only if it has a good performance, model variations do not affect the execution performance of the queries. However, read operations concurs with update and insert operations, and in this case there is a usual performance loss.

Considering the workloads that include write operations, there are two different situations: write operations that result from existing records (updates) and write operations that are inserts of new registries (inserts). Due to the fact that Cassandra does not read the existing values when running an update [6], when the number of updates is greater than the number of inserts, the table-based modeling tends to offer better performance, even running more operations. The use of materialized views makes Cassandra lose this feature of not reading already existing records [13], tending to perform worse. Due to these differences, two comparisons were made with write operations in workloads. One performing only updates and another only inserting new records.

Fig.5 shows the results of the execution of the workloads with inserts. This graph uses logarithmic scale, but the absolute values of the runtime in minutes are highlighted in each bar. From these results, it was observed that scenario 3 (table-based modeling) is clearly the most affected by the increase of writing operations. This scenario has a table for each query and these tables are completely independent of each other. Therefore, each writing to be done in the database, must be done thirteen times in order to update all tables. This fact explains the significant increase of the execution time in this scenario, as the number of write operations increase.

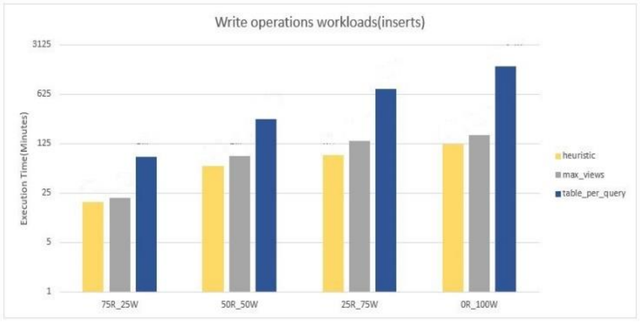


Fig. 5 – Graph of workloads with new registries insert operations

Still considering Fig.5, it was verified that scenario 1 (modeling generated by the heuristic) performs better than scenario 2 (modeling with a materialized view per query). This improvement in performance ranges from 13% to 35% and it is justified by the fact that in scenario 2 there is a greater number of materialized views. This does not imply the execution of a greater number of write operations, as occurs in the scenario 3. The replication of the updates is done automatically by Cassandra internally. Although this replication is much more efficient than to write separately in each table, it has a running cost. As scenario 2 has more views than scenario 1, this cost ends up directly affecting the performance in scenario 2.

Fig.6 shows the results of the executions of the workloads with updates. This graph also uses logarithmic scale and the values of the runtime in minutes are highlighted in each bar. In the case of existing records updates, there is a change of behavior considering the operations of new records insertions. It was observed that scenario 3 performed better compared to other scenarios. This is because Cassandra implements updates on materialized views as follows: performs reading the data already present in the materialized views, updates it, removes the old record and insert the new record. While without the use of materialized views, Cassandra simply inserts the record with a most recent timestamp and this is the record that will be returned to the queries.

Comparing scenarios 1 and 2, it can be observed in Fig.6 that scenario 1 presents better performance in all situations, ranging from 14% to 31%, performance similar to the use of inserts. In addition, there is an increase in performance gain as the number of updates increases.

Although scenario 3 performs better than scenarios 1 and 2, the use of the table-based modeling leads to a higher cost of maintaining the data, since change in one table should be reflected in all other tables created, losing Cassandra's feature of syncing materialized views.

We can observe that a factor that influences the number of views suggested by the heuristic is the variety of attributes in the queries filters. The smaller the variety of these attributes, the smaller the number of views generated by the heuristic, ensuring a more significant performance of scenario 1 compared to scenario 2, especially when there is a lot of write operations.

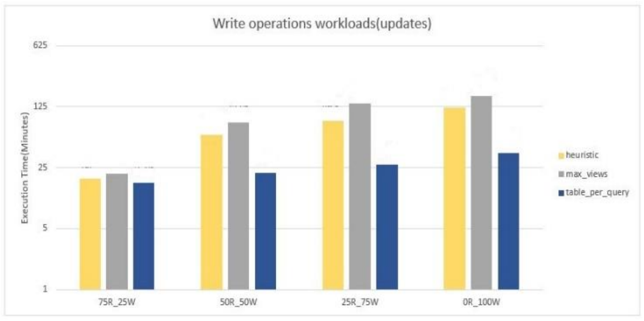


Fig. 6 – Graph of workloads with existing registries update operations

7. Conclusion

This work presents a set of guidelines to support the logical/physical design of database schemas for Cassandra DBMS. It includes a heuristic for data modeling based on specific queries to define a set of materialized views and their corresponding primary keys.

The CNSSB benchmark dataset and its queries were used to evaluate the proposed heuristic. The experiments used workloads varying the rate of read/write operations. The results showed that the more insert operations the better was the performance of the heuristic. On the other hand, when most of the operations are updates, the use of a table for each query performs better. However, it is worth to say that the reduced number of materialized views (heuristic scenario) is still a better choice if compared to the use of all possible materialized views. Therefore, the proposed guidelines bring a light to the data modeling for Cassandra DBMS. Moreover, in the case of analytical applications, where write operations are usually a large set of inserts, the heuristic is particularly useful.

For future work, we plan to apply the heuristic with different datasets and sets of queries. Also, we intend to investigate the application of the heuristic, with adjustments, over different NOSQL DBMS that behave similarly to Cassandra. Additionally, we intend to evaluate the impact of Cassandra replication factor and how its variation may affect query performance and memory usage.

REFERENCES

[1] Pramod J. Sadalage and Martin Fowler. 2012. NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence (1st ed.). Addison-Wesley Professional

[2] R. A. S. N. Soransso and Maria Cláudia Cavalcanti. 2018. Data modeling for analytical queries on document-oriented DBMS. In Proceedings of the 33rd Annual ACM Symposium on Applied Computing, SAC 2018, Pau, France, April 09-13, 2018. 541–548.

[3] Silas P. Lima Filho, Maria Cláudia Cavalcanti, and Cláudia Marcela Justel. 2018. Managing Graph Modeling Alternatives for Link Prediction. In Proceedings of the 20th International Conference on Enterprise Information Systems, ICEIS 2018, Funchal, Madeira, Portugal, March 21-24, 2018, Volume 2. 71–80.

[4] Lucas C. Scabora, Jaqueline Joice Brito, Ricardo Rodrigues Ciferri, and Cristina Dutra de Aguiar Ciferri. 2016. Physical Data Warehouse Design on NoSQL Databases - OLAP Query Processing over HBase. In Proc. of the 18th Int. Conf. on Enterprise Information Syst. (ICEIS) (1 ed.). 111–118.

[5] Gheorghe Matei. 2010. Column-Oriented Databases, an Alternative for Analytical Environment. Database Systems Journal 1 (2010), 3–16.

[6] DataStax. 2018. How Cassandra reads and writes data. https://docs.datastax.com/en//cassandra/3.0/cassandra/dml/dmlHowDataWritten.html 13 jul. de 2018.

[7] R. Kimball and M. Ross. The Data Warehouse Toolkit: the Definitive Guide to Dimensional Modeling. John Wiley & Sons, New York, USA, 3 edition, 2013.

[8] Max Chevalier, Mohammed El Malki, Arlind Kopliku, Olivier Teste, and Ronan Tournier. 2015. Implementing Multidimensional Data Warehouses into NoSQL. In ICEIS 2015 - Proceedings of the 17th International Conference on Enterprise Information Systems, Volume 1, Barcelona, Spain, 27-30 April, 2015 (1 ed.). 172–183.

[9] Khaled Dehdouh, Fadila Bentayeb, Omar Boussaid, and Nadia Kabachi. 2015. Using the column oriented NoSQL model for implementing big data warehouses. In Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA) (1 ed.). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 469.

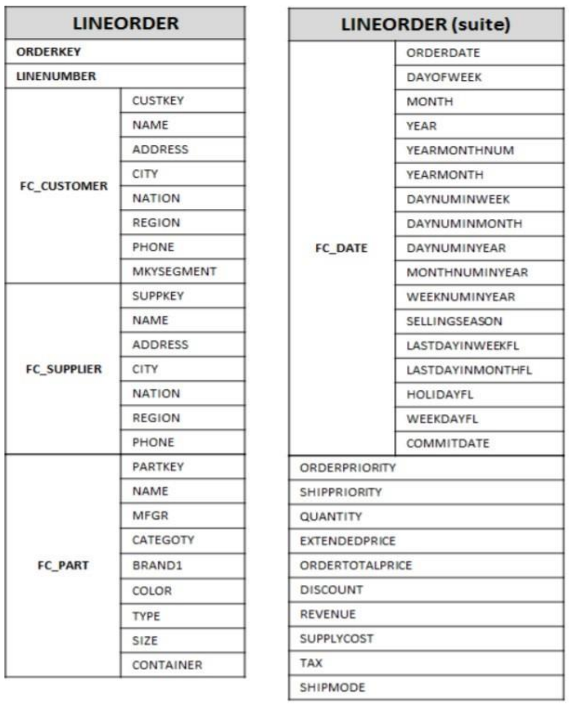
[10] A. Chebotko, A. Kashlev, and S. Lu. 2015. A Big Data Modeling Methodology for Apache Cassandra. In 2015 IEEE International Congress on Big Data (1 ed.). 238–245. https://doi.org/10.1109/BigDataCongress.2015.41

[11] Khaled Dehdouh, Fadila Bentayeb, and Omar Boussaid. 2014. Columnar NoSQL Star Schema Benchmark. In Model and Data Engineering (1 ed.). The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), Springer International Publishing, 281–288.

[12] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. 1997. Materialized Views Selection in a Multidimensional Database. In VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece (1 ed.). 156–165.

[13] Jonathan Ellis. 2018. Materialized View Performance in Cassandra 3.x.https://www.datastax.com/dev/blog/materialized-view-performance-in-cassandra-3-x13 jul. de 2018.

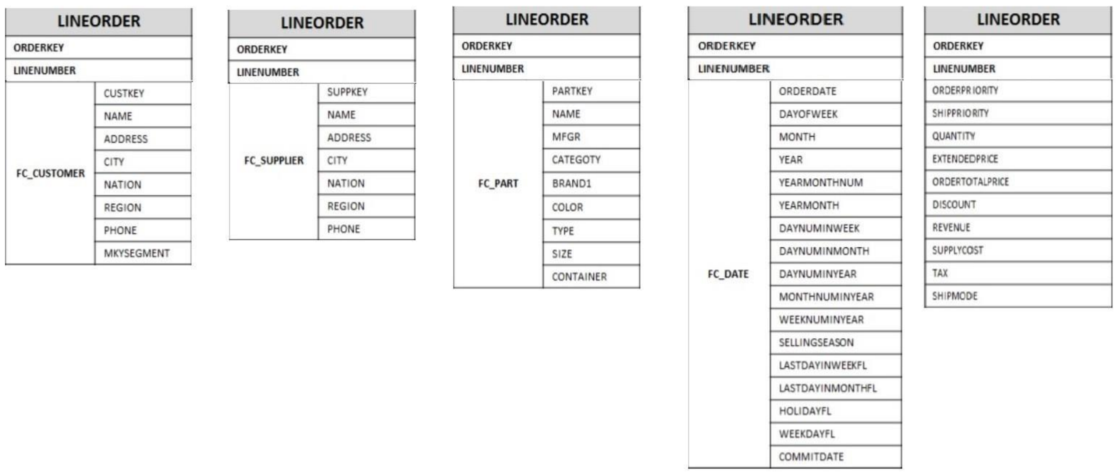
Appendix



LINEORDER	
ORDERKEY	
LINENUMBER	
FC_CUSTOMER	CUSTKEY
	NAME
	ADDRESS
	CITY
	NATION
	REGION
	PHONE
	MKYSEGMENT
FC_SUPPLIER	SUPPKEY
	NAME
	ADDRESS
	CITY
	NATION
	REGION
	PHONE
FC_PART	PARTKEY
	NAME
	MFGR
	CATEGOTY
	BRAND1
	COLOR
	TYPE
	SIZE
	CONTAINER

LINEORDER (suite)	
FC_DATE	ORDERDATE
	DAYOFWEEK
	MONTH
	YEAR
	YEARMONTHNUM
	YEARMONTH
	DAYNUMINWEEK
	DAYNUMINMONTH
	DAYNUMINYEAR
	MONTHNUMINYEAR
	WEEKNUMINYEAR
	SELLINGSEASON
	LASTDAYINWEEKFL
	LASTDAYINMONTHFL
	HOLIDAYFL
	WEEKDAYFL
	COMMITDATE
ORDERPRIORITY	
SHIPPRIORITY	
QUANTITY	
EXTENDEDPRICE	
ORDERTOTALPRICE	
DISCOUNT	
REVENUE	
SUPPLYCOST	
TAX	
SHIPMODE	

Fig. 7 – SameCF data model



LINEORDER	
ORDERKEY	
LINENUMBER	
FC_CUSTOMER	CUSTKEY
	NAME
	ADDRESS
	CITY
	NATION
	REGION
	PHONE
	MKYSEGMENT

LINEORDER	
ORDERKEY	
LINENUMBER	
FC_SUPPLIER	SUPPKEY
	NAME
	ADDRESS
	CITY
	NATION
	REGION
	PHONE

LINEORDER	
ORDERKEY	
LINENUMBER	
FC_PART	PARTKEY
	NAME
	MFGR
	CATEGOTY
	BRAND1
	COLOR
	TYPE
	SIZE
	CONTAINER

LINEORDER	
ORDERKEY	
LINENUMBER	
FC_DATE	ORDERDATE
	DAYOFWEEK
	MONTH
	YEAR
	YEARMONTHNUM
	YEARMONTH
	DAYNUMINWEEK
	DAYNUMINMONTH
	DAYNUMINYEAR
	MONTHNUMINYEAR
	WEEKNUMINYEAR
	SELLINGSEASON
	LASTDAYINWEEKFL
	LASTDAYINMONTHFL
	HOLIDAYFL
	WEEKDAYFL
	COMMITDATE

LINEORDER
ORDERKEY
LINENUMBER
ORDERPRIORITY
SHIPPRIORITY
QUANTITY
EXTENDEDPRICE
ORDERTOTALPRICE
DISCOUNT
REVENUE
SUPPLYCOST
TAX
SHIPMODE

Fig. 8. DLA-CF data model

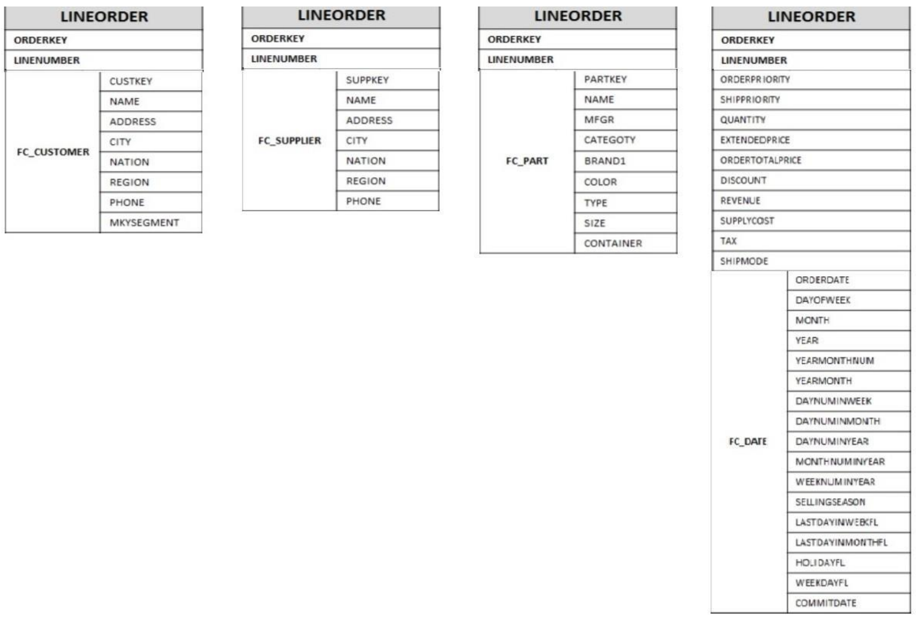


Fig. 9. FactDate data model

Columnar Star Schema Benchmark Queries

Query 1.1 from CNSSB

select sum(lo_extendedprice*lo_discount) as revenue from lineorder, date where lo_orderdate = d_datekey and d_year = 1993 and lo_discount between1 and 3 and lo_quantity < 25;

Query 1.2 from CNSSB

select sum(lo_extendedprice*lo_discount) as revenue from lineorder, date where lo_orderdate = d_datekey and d_yearmonthnum = 199401 and lo_discount between 4 and 6 and lo_quantity between 26 and 35;

Query 1.3 from CNSSB

select sum(lo_extendedprice*lo_discount) as revenue from lineorder, date where lo_orderdate = d_datekey and d_weeknuminyear = 6 and d_year = 1994 and lo_discount between 5 and 7 and lo_quantity between 26 and 35;

Query 2.1 from CNSSB

select sum(lo_revenue), d_year, p_brand1 from lineorder, date, part, supplier where lo_orderdate = d_datekey and lo_partkey = p_partkey and lo_suppkey = s_suppkey and p_category = ’MFGR#12’ and s_region = ’AMERICA’

group by d_year, p_brand1 order by d_year, p_brand1;

Query 2.2 from CNSSB

select sum(lo_revenue), d_year, p_brand1 from lineorder, date, part, supplier where lo_orderdate = d_datekey and lo_partkey = p_partkey and lo_suppkey = s_suppkey and p_brand1 between ’MFGR#2221’ and ’MFGR#2228’ and s_region = ’ASIA’

group by d_year, p_brand1 order by d_year, p_brand1;

Query 2.3 from CNSSB

select sum(lo_revenue), d_year, p_brand1 from lineorder, date, part, supplier

where lo_orderdate = d_datekey and lo_partkey = p_partkey and lo_suppkey = s_suppkey and p_brand1 = ’MFGR#2221’ and s_region = ’EUROPE’

group by d_year, p_brand1 order by d_year, p_brand1;

Query 3.1 from CNSSB

select c_nation, s_nation, d_year, sum(lo_revenue) as revenue from customer, lineorder, supplier, date where lo_custkey = c_custkey and lo_suppkey = s_suppkey and lo_orderdate = d_datekey and c_region = ’ASIA’ and s_region = ’ASIA’ and d_year >= 1992 and d_year <= 1997

group by c_nation, s_nation, d_year order by d_year asc, revenue desc;

Query 3.2 from CNSSB

select c_city, s_city, d_year, sum(lo_revenue) as revenue from customer, lineorder, supplier, date where lo_custkey = c_custkey and lo_suppkey = s_suppkey and lo_orderdate = d_datekey and c_nation = ’UNITED STATES’ and s_nation = ’UNITED STATES’ and d_year >= 1992 and d_year <= 1997 group by c_city, s_city, d_year order by d_year asc, revenue desc;

Query 3.3 from CNSSB

select c_city, s_city, d_year, sum(lo_revenue) as revenue from customer, lineorder, supplier, date where lo_custkey = c_custkey and lo_suppkey = s_suppkey and lo_orderdate = d_datekey and (c_city=’UNITED KI1’ or c_city=’UNITED KI5’) and (s_city=’UNITED KI1’ or s_city=’UNITED KI5’) and d_year >= 1992 and d_year <= 1997 group by c_city, s_city, d_year order by d_year asc, revenue desc;

Query 3.4 from CNSSB

select c_city, s_city, d_year, sum(lo_revenue) as revenue from customer, lineorder, supplier, date where lo_custkey = c_custkey and lo_suppkey = s_suppkey and lo_orderdate = d_datekey and (c_city=’UNITED KI1’ or c_city=’UNITED KI5’) and (s_city=’UNITED KI1’ or s_city=’UNITED KI5’) and d_yearmonth = ’Dec1997’ group by c_city, s_city, d_year order by d_year asc, revenue desc;

Query 4.1 from CNSSB

select d_year, c_nation, sum(lo_revenue - lo_supplycost) as profit from date, customer, supplier, part, lineorder where lo_custkey = c_custkey and lo_suppkey = s_suppkey and lo_partkey = p_partkey and lo_orderdate = d_datekey and c_region = ’AMERICA’ and s_region = ’AMERICA’ and (p_mfgr = ’MFGR#1’ or p_mfgr = ’MFGR#2’) group by d_year, c_nation order by d_year, c_nation

Query 4.2 from CNSSB

select d_year, s_nation, p_category, sum(lo_revenue - lo_supplycost) as profit from date, customer, supplier, part, lineorder where lo_custkey = c_custkey and lo_suppkey = s_suppkey

and lo_partkey = p_partkey and lo_orderdate = d_datekey and c_region = ’AMERICA’ and s_region = ’AMERICA’ and (d_year = 1997 or d_year = 1998) and (p_mfgr = ’MFGR#1’ or p_mfgr = ’MFGR#2’) group by d_year, s_nation, p_category order by d_year, s_nation, p_category

Query 4.3 from CNSSB

select d_year, s_city, p_brand1, sum(lo_revenue - lo_supplycost) as profit from date, customer, supplier, part, lineorder where lo_custkey = c_custkey and lo_suppkey = s_suppkey and lo_partkey = p_partkey and lo_orderdate = d_datekey and c_region = ’AMERICA’ and s_nation = ’UNITED STATES’ and (d_year = 1997 or d_year = 1998) and p_category = ’MFGR#14’ group by d_year, s_city, p_brand1 order by d_year, s_city, p_brand1

Columnar Star Schema Benchmark Queries Apdapted to Cassandra

Query 1.1 adapted from CNSSB

select discount,quantity from
cnssb.nlineorder_v3 where year = 1993
and yearmonth in ('Jan1993','Feb1993','Mar1993','Apr1993','May1993','Jun1993'
,'Jul1993' ,'Aug1993','Sep1993' ,'Oct1993','Nov1993','Dec1993') and quantity in
(0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25) and discount in(1,2,3);

Query 1.2 adapted from CNSSB

select extendedprice,discount as revenue from
cnssb.nlineorder_v3 where year = 1994 and yearmonth
= 'Jan1994' and quantity in (26,27,28,29,30,31,32,33,34,35) and
discount in(4,5,6);

Query 1.3 adapted from CNSSB

select extendedprice, discount as revenue from
cnssb.nlineorder_v4 where year = 1994 and weeknuminyear
= 6
and quantity in (26,27,28,29,30,31,32,33,34,35) and discount
in(5,6,7);

Query 2.1 adapted from CNSSB

select revenue, year, brand1 from
cnssb.nlineorder_v5
where suppregion = 'AMERICA' and category = 'MFGR#12';

Query 2.2 adapted from CNSSB

select revenue, year, brand1 from
cnssb.nlineorder_v1 where suppregion
= 'ASIA'
and brand1 <= 'MFGR#2228' and brand1 >= 'MFGR#2221';

Query 2.3 adapted from CNSSB

select revenue, year, brand1 from
cnssb.nlineorder_v1
where suppregion = 'EUROPE' and brand1 = 'MFGR#2221';

Query 3.1 adapted from CNSSB

select nation, suppnation, year, revenue as revenue from
cnssb.nlineorder_v6 where suppregion = 'ASIA' and region
= 'ASIA' and year >= 1992 and year <= 1997;

Query 3.2 adapted from CNSSB

select city, suppcity, year, revenue as revenue from
cnssb.nlineorder_v8
where nation = 'UNITED STATES' and suppnation = 'UNITED STATES' and year >= 1992 and
year <= 1997;

Query 3.3 adapted from CNSSB

select city, suppcity, year, revenue as revenue from cnssb.nlineorder_v2 where city in ('UNITED KI1', 'UNITED
KI5') and suppcity in ('UNITED KI1', 'UNITED KI5') and yearmonth in
('Jan1992','Feb1992','Mar1992','Apr1992','May1992','Jun1992'
,'Jul1992','Aug1992','Sep1992','Oct1992','Nov1992','Dec1992',
'Jan1993','Feb1993','Mar1993','Apr1993','May1993','Jun1993'
,'Jul1993','Aug1993','Sep1993','Oct1993','Nov1993','Dec1993',
'Jan1994','Feb1994','Mar1994','Apr1994','May1994','Jun1994'
,'Jul1994','Aug1994','Sep1994','Oct1994','Nov1994','Dec1994',
'Jan1995','Feb1995','Mar1995','Apr1995','May1995','Jun1995'
,'Jul1995','Aug1995','Sep1995','Oct1995','Nov1995','Dec1995',
'Jan1996','Feb1996','Mar1996','Apr1996','May1996','Jun1996'
,'Jul1996','Aug1996','Sep1996','Oct1996','Nov1996','Dec1996',

’Jan1997’,’Feb1997’,’Mar1997’,’Apr1997’,’May1997’,’Jun1997’ ,’Jul1997’,’Aug1997’,’Sep1997’,’Oct1997’,’Nov1997’,’Dec1997’) and year >= 1992 and year <= 1997;

Query 3.4 adapted from CNSSB

select city, suppcity, year, revenue as revenue from cnssb.nlineorder_v2
where city in (’UNITED KI1’, ’UNITED KI5’)
and suppcity in (’UNITED KI1’, ’UNITED KI5’) and yearmonth = ’Dec1997’;

Query 4.1 adapted from CNSSB

select year, nation, revenue, supplycost from cnssb.nlineorder
where region = ’AMERICA’ and suppregion = ’AMERICA’ and mfgr in (’MFGR#1’,’MFGR#2’);

Query 4.2 adapted from CNSSB

select year, nation, revenue, supplycost from cnssb.nlineorder
where region = ’AMERICA’ and suppregion = ’AMERICA’ and year in (1997,1998) and mfgr in (’MFGR#1’,’MFGR#2’);

Query 4.3 adapted from CNSSB

select year, nation, revenue, supplycost from cnssb.nlineorder_v7 where region = ’AMERICA’ and suppnation = ’UNITED STATES’ and year in (1997,1998) and category = ’MFGR#14’;