VOL.38 Nº4 2021

Data Modeling for Cassandra

Thiago B Leão*, Maria Claudia R Cavalcanti, Raquel C G Pinto
Instituto Militar de Engenharia (IME),
Praça General Tibúrcio, 80, 22290-270,
Praia Vermelha, Rio de Janeiro, RJ, Brasil.
*** thiagobleao@gmail.com**

RESUMO: Devido à crescente demanda por escalabilidade e distribuição de dados, uma variedade de sistemas gerenciadores de banco de dados NOSQL surgiram e vêm sendo usados com diferentes objetivos. Entre as abordagens mais populares podemos citar a orientada a documentos, a orientada a colunas e a chave-valor. Embora já existam vários sistemas de banco de dados que adotam essas abordagens, até onde foi possível investigar, quase não há diretrizes de modelagem de dados para eles. O algoritmo proposto neste artigo analisa um conjunto de consultas pré-definido e, baseado nas cláusulas de filtro dessas consultas, ele define as chaves primárias e de clustering para um conjunto de visões materializadas. Adicionalmente, ele define um conjunto pares <consulta, visão materializada> indicando quais consultas cada visão materializada atende. Para avaliar o algoritmo, foi realizado um experimento que compara o desempenho entre usar diversas tabelas para cada consulta e usar as visões materializadas sugeridas pelo algoritmo. Os resultados mostraram-se promissores e apontam para novas direções com relação à modelagem de dados para sistemas NOSQL.

PALAVRAS-CHAVE: modelagem de dados, NOSQL, visões materializadas.

ABSTRACT: Due to the increasing demand for scalability and distribution of data, a variety of NOSQL database management systems have emerged and are being used for different purposes. Some of the most popular systems are: Document-oriented, Column-oriented and Key-value. Although there are many of these systems, to the best of our knowledge, there are barely any guidelines for data modeling for them. The proposed algorithm analyzes a predefined set of queries and, based on their filter clauses, it defines the composition of the primary and clustering keys for a set of materialized views. In addition, it defines a set of <query, materialized view> pairs, indicating which queries each materialized view addresses. To evaluate the algorithm, we report on an experiment that compares the performance of having different tables for each query and of having the materialized views suggested by the algorithm. It shows promising results and points to new directions on data modeling for NOSQL systems.

KEYWORDS: data modeling, NOSQL, materialized views.

1. Introduction

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 many different NOSQL solutions already available, to the best of our knowledge, there are barely any guidelines for data modeling for such solutions. Moreover, modeling guidelines for Relational DBMS are not appropriate 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 the data modeling process for them cannot be the same.

Therefore, in this paper, we propose some guidelines to help the user with 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 materialized views to be created and maintained.

In order to evaluate the proposed algorithm, we took a well-known benchmark for OLAP applications. The idea was to evaluate performance gains among different data modelings 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 contribution 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 by indicating future directions.

2. NOSQL databases

One of the main characteristics of NOSQL DBMS is the ease of dealing 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 the replicas and/or shards not to be fully consistent all the time, but at some future point in time. This disadvantage is acceptable to obtain benefits such as availability and performance.

The main characteristic of column-oriented databases is that they 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, directly affecting 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 because of its popularity among the column-oriented databases (according to the DB-Engines site [1]).

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 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, which is composed of a partition key and clustering keys. The partition key is used by Cassandra to define how data will be partitioned over the nodes. The clustering keys define how data will be ordered on the partition. 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

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

are not dynamic and must be previously declared in a configuration file. They are the unit of abstraction containing keyed rows that 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 can 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 predefined query, aiming at improving performance. Each table may have one or more materialized views.

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

3. Related work

This section summarizes and compares papers presented as shown in Table 1. 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] which has a heavy use of queries to retrieve large volumes of data. 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 multi-dimensional conceptual model directly to the NOSQL logical model is proposed by Chevalier *et al* [8]: each star schema is mapped into a single table. The fact is transformed into a column family, in which every measure is a column. Each dimension is transformed into a column family, in which every attribute is a column. In addition, all aggregation possibilities for that schema are also similarly mapped into a separate table, as materialized views. However, in this work, there is no intention in selecting a subset of those materialized views, which implies high costs concerning storage and materialized 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.

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

On Scabora *et al* [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 Dehdouh *et al* [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 the fact and the date dimension in the same column family. 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 were 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 the Cassandra database system.

4. Initial experiments

As previously mentioned [8], 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.

In short, data modeling consists of two steps. The first step is concerned with conceiving the first version of a logical schema. Then, the second one focuses on performance issues and on attending to application demands, such as addressing 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.

Tab. 1 – related work comparison.

Work	Modeling	Materialized Views	DBMS
Chevalier *et al* [8]	Conceptual/ Logical	-	MongoDB and HBase
Dehdouh *et al* [9]	Conceptual/ Logical	-	HBase
Chebotko *et al* [10]	Logical	-	Cassandra
Scabora *et al* [4]	Logical	Materialized Views with an external application	HBase

In order to develop such an algorithm, we performed some initial experiments for a typical OLAP application, which are detailed in this Section. To guarantee that we would start with the best initial logical schema, we explored the CNSSB datasets [11] and queries, considering three logical schemas, as proposed in [9] and [4]: SameCF, DLA-CF, and FactDate. Based on the results of such experiments detailed in Section 4.2, 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 and 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)[2] were populated with data generated from the DBGen application of the CNSSB[3] [11]. After populating the models, the thirteen CNSSB queries, organized in 4 typical sets, Q1, Q2, Q3, and Q4 [11], were executed five times on each model to measure their performance average. Experimentally, it was noticed that there was not a large variation within a few executions. Thus, five executions seemed to be sufficient to characterize the performance.

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

Since the SameCF model had an average superior performance to the DLA-CF andFactDate, this model was chosen as the initial logical model. From this model, an evaluation of the use of materialized views is presented in the following section. Then, a heuristic is defined based on the 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 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 [6]. 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 **figure 2**, in the set of queries QG1, there was a significant variation in 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, since 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 concerning 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). Although queries 3.2 and 3.3 used a filter based on the Partition Key attribute (year), the other attributes used as filters belong to unfavorable positions in the Clustering Key, which explains their bad performances. Query 3.4 does not use the Partition Key (year) attribute as a filter but as an ordering/grouping attribute and, in addition, it also filters data by unfavorably positioned Clustering Key attributes.

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

[2]https://db-engines.com/en/ranking
[3]https://github.com/thiagobleao/cnssb_dataset

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.

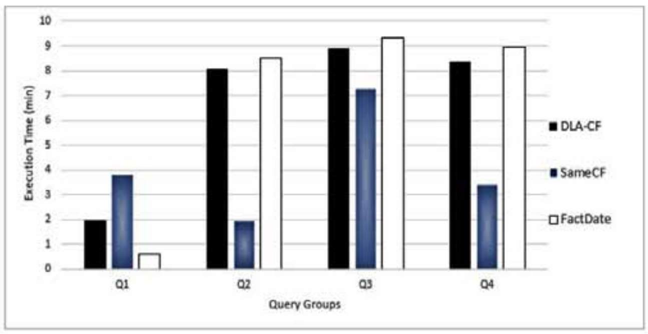

Fig. 1 – Query performance on each model

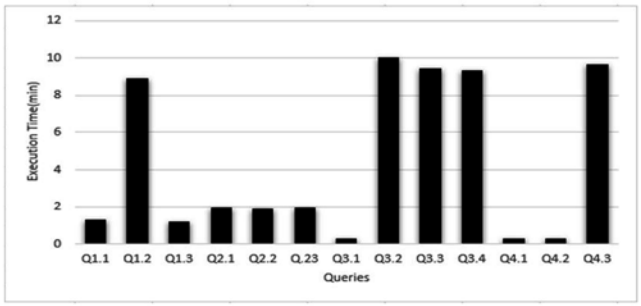

Fig. 2 – Queries performance in SameCF model

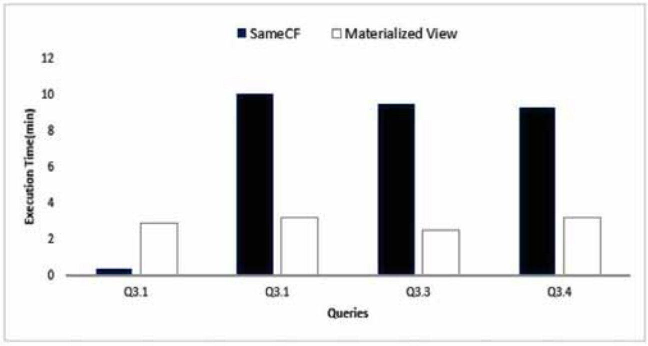

Fig. 3 – Materialized view performance

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 materialized view and the reorganization of the Partition and Clustering Keys for that materialized view. Using the criterion proposed by Baralis *et al* [12], we chose the worst performance set of queries (Q3) as the basis for the creation of the materialized view. The graph of **figure 3** shows the performance of these queries on the materialized view and compares 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 using 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 using joins through applications is not ideal since there is a loss of performance. That said, the ideal for modeling in Cassandra is to denormalize 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 DBMS independent view of the application and arrive at a DBMS dependent schema of it. In the context of a relational database, it consists of designing a set of tables and attributes, whereas, in the context of a columnar database approach, this means choosing 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 defining materialized views.

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

Guideline 1. Denormalize the logical schema.

Considering 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 selecting and filtering clauses. These groups are the basis for the choice of the materialized views. Once defined, these materialized 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 materialized 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 with 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. Another premise based on Cassandra's limitations is the use of inequality constraints, which 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 materialized view. The main idea is to identify groups of queries, of which constraints use attributes in common, and for each group, the algorithm suggests a reduced number of materialized views, with their respective primary keys. In addition, for each materialized view, it suggests the set of queries that are associated with it, i.e., it indicates to which materialized view each query should be redirected to (or rewritten to).

The following variables are used in Algorithm 1 (**figure 4**):

- Q: set of critical/typical queries to be executed on Cassandra;
- v: index of the materialized view under construction, $1 \leq v \leq |Q|$;
- AE_x: 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 AE_v of the primary key of the associated materialized view v;
- AE_v: set of equality-based attributes of materialized view v;
- ANE_x: 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 ANE_v primary key of the associated materialized view v;
- ANE_v: set of inequality-based attributes of materialized view v;
- A_x: set of attributes of query qx, where $A_x = AE_x \cup ANE_x$;
- A_v: set of attributes that will compose the primary key of the associated materialized view, where $A_v = AE_v \cup ANE_v$;
- Q_v: set of queries that will be addressed by the materialized view v;
- V : set of materialized views to be generated, formed by a set of pairs (A_v, Q_v);

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 at building a new materialized view v and its corresponding primary key. Every query q_x from Q´ is analyzed by the algorithm concerning its attributes involved in equality and inequality constraints until the key for the materialized view under construction is formed. Depending on the evolution of the key under construction for the materialized view of the moment (v), query q_x 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 (Q_v) that will be addressed by materialized 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 materialized view of the moment. Once the first attributes for a materialized view are defined, then the next Q´ queries are treated by the following steps, depending on if they have attributes in common with the materialized view under construction. Step III will add queries that bring to the materialized view one attribute involved in an inequality constraint at the most, and those queries have attributes in common with the set of attributes of the materialized view ($A_v \subseteq A_x$ or $A_x \subseteq A_v$). Step IV deals with queries that bring to the materialized view more than one attribute involved in an inequality constraint. In this case, the query under analysis needs to have the same equality constraint attributes. Queries that did not fit within any of the previous steps remain in Q´, and they will be analyzed again for a new iteration, i.e., for the creation of a new materialized view.

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

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

6. Experiments with materialized views in cassandra

In this section, we present experiments' results to evaluate the heuristics proposed in the previous section. The purpose of these experiments is to compare the performance of three logical models with different input rates of reading and writing 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 dataset 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[4], the proposed heuristic created a set of nine materialized views. Each materialized view has a different primary key that will allow the 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, it 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

[4] https://github.com/thiagobleao/rmct_appendix/blob/main/rmct_apendix.pdf

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

If executed as presented, these queries 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 <" to "quantity in". According to Cassandra's constraints, the filter on the discount attribute could be made using "greater than" or"less than" since it is the last attribute of the clustering key of the materialized view. But in order not to assume that this will necessarily be done by the heuristic, it was decided to adapt this filter also to use the"in" clause. After these adaptations, the queries became as follows:

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 the query (1) so that it could be served by the same materialized view of the query (2). Originally, query (1) does not have the filter on the attribute yearmonth. However, it could be included in query (1) to achieve the same result and to add a few restrictions to query (1). The same cannot be done to 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 number 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 aforementioned adaptations, have equality filters 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

Query (3) should correspond to a materialized view that will only attend to that query. The key of this materialized view will be composed of the attributes "year" and "weeknuminyear". One of them will be chosen for the partition key. The other one will be the first attribute in the clustering key of that view, and it will be followed by the attributes quantity and discount (involved in inequality conditions). The composition of the primary key of the view that will support this query is:

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

Like the examples cited above, nine materialized views were 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 compared 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 reading and writing operations. The objective is to analyze the performance of the evaluated models considering different possibilities of operations that are made over a database.

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 reading 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 the update and insert operations and, in this case, there is a usual performance loss.

Tab 2: materialized 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

Considering the workloads that include writing 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 when it runs more operations. However, the use of materialized views makes Cassandra lose this feature [13], leading to a worse performance. Due to these differences, two comparisons were made with write operations in workloads, one performing only updates and another only inserting new records.

Figure 5 shows the results of the execution of the workloads with inserts. This graph uses a 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 increases.

Still considering **figure 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 it occurs in scenario 3. The replication of the updates is done automatically by Cassandra internally. Although this replication is much more efficient than writing separately in each table, it has a running cost. As scenario 2 has more materialized views than scenario 1, this cost ends up directly affecting the performance in scenario 2.

Figure 6 shows the results of the executions of the workloads with updates. This graph also uses a 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: it performs reading the data already present in the materialized views, updates it, removes the old record and inserts the new one. 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.

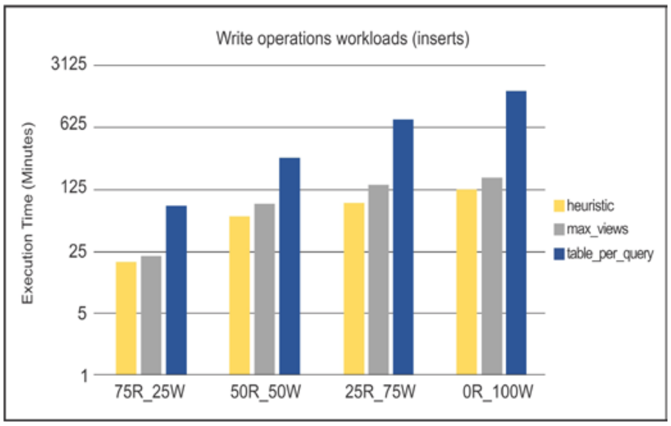

Fig. 5 – Graph of workloads with new registries insert operations

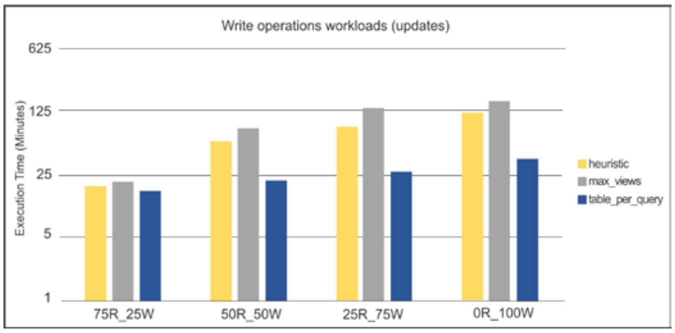

Fig. 6 – Graph of workloads with existing registries update operations

Comparing scenarios 1 and 2, it can be observed in **figure 6** that scenario 1 presents a 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 a 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 materialized views suggested by the heuristic is the variety of attributes in the query filters. The smaller the variety of these attributes, the smaller the number of materialized views generated by the heuristic, ensuring a more significant performance of scenario 1 compared to scenario 2, especially when there are a lot of write 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 updated, the use of a table for each query performs better. However, it is worth saying 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 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, sets of queries, and applications like OLTP. 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.

Acknowledgments

The authors would like to thank the Brazilian agency CAPES for the financial support and Cristina Dutra de Aguiar Ciferri for her careful review of this work. In addition, thanks to the Brazilian Army for the use of the computational infrastructure of the High-Performance Computing Laboratory for Cyber Defense, funded with resources from the Brazilian Army Strategic Project.

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.