A Data Partitioning Optimization Approach for Distributed Data Warehouses on Column family NoSQL Systems

Mohamed. Boussahoua[1], **Fadila. Bentayeb**[1], **Omar. Boussaid**[1], **and Nadia. Kabachi**[2]
[1]University of Lyon, Lyon 2, ERIC EA 3083
[2]University of Lyon, University Claude Bernard Lyon 1, ERIC EA 3083
5 avenue Pierre Mendès France - F69676 Bron Cedex - France

Abstract— *Column family NoSQL databases offer storage techniques that are well adapted to data warehouses. Several scenarios are possible to develop data warehouses on these databases. In this paper, we propose a new method to build a distributed data warehouse using a column family NoSQL database. Our method is based on an attribute-grouping strategy to define the column families that constitute the logical warehouse schema, which allows to have the most appropriate physical data model. To do that, we adopt two regrouping algorithms. First, the K-medoids algorithm that gathers the similar queries in classes. Then, the Particle Swarm Optimization algorithm groups attributes to create the column families according to each similar class of queries. To evaluate our method, we use the TPC-DS benchmark. We then carry out several tests to show the effectiveness of these algorithms to build a data warehouse in NoSQL HBase database on a Hadoop platform.*

Keywords: NoSQL databases, Data warehouses, Data partition

1. Introduction

The storage and management of large, complex, structured or unstructured data is a challenging task in the current computing environments. To address the scalability requirements of today's data analytics, new robust and efficient data storage and processing systems have been established by the scientific and industrial institutions such as Google's MapReduce [1], C-Store [2], Apache Hadoop [1], Yahoo's PNUTS [3], NoSQL systems [2]. Recently, NoSQL databases have been used not only in web applications deployments (e.g., Facebook, Twitter and google, etc.,) but also in distributed storage and processing of large data warehouses on clusters of commodity hardware. However, the problematic thing with NoSQL systems is that the data implementation process is not trivial because each NoSQL database model has specific data structures and concepts (e.g. Key-Value stores, Column families databases, Document databases, and other models NoSQL databases). Consequently, these approaches require revisiting the principles of traditional data warehouse modeling process, especially at the logical and physical design level. We saw that, in most cases, the objective of this adaptive data modeling is to identify the structure of data representations ahead of time and to create a robust NoSQL data schema before the data is loaded from data sources. Indeed, in practice, the concepts of the schema-less and data schema flexibility in NoSQL databases offer limited freedom for building data warehouses. Therefore, to get the most beneficial features of NoSQL databases, the predefined data schema becomes necessary to store data warehouses in NoSQL systems.

In this paper, we address the storage and implementation process of data warehouses with column family NoSQL databases. So, to take advantage of these types of databases, the main question that arises when trying to accommodate the data structures is: how to organize data in column families to serve effectively OLAP queries? To answer this question, we studied the benefits of grouping techniques on column families' creation within the context of data warehousing. Based on query workload where queries are identified (but not limited), we propose the application of two clustering techniques. The first one is the *K-medoids* algorithm that gathers similar queries in a class. The second one is based on the meta-heuristics *Particle Swarm Optimization (PSO)*, to determine which attributes frequently used by the same class of similar queries should be grouped together. This helps to produce a better column family schema, that leads to a more distributed data warehouse schema (that exactly matches with the user's needs) in column family NoSQL databases.

Several tests were made to evaluate the effectiveness of our proposed method. We adopted the TPC-DS data benchmark. To design the columnar NoSQL data warehouse (*CN-DW*) for TPC-DS benchmarking database, we used 3 different methods, our method and 2 other methods that have been already tested and implemented successfully in [4] and [5]. Then, we execute a query workload on different schemas upon *CN-DW* built over TPC-DS. We show that the application of clustering techniques for designing a data warehouses model *CN-DW* effectively improves the query execution time.

[1]http://hadoop.apache.org/
[2]http://nosql-database.org/

The remainder of the paper is organized as follows. Section 2 presents related works. Section 3 describes the problem we address in this paper. Section 4 presents our proposed approach. Section 5 evaluates our approach. Conclusion and future works are given in section 6.

2. Related Work

Using column family NoSQL databases for data warehouse solutions has been debated within the scientific community. Several naïve approaches such as [4], [5] and [6] have been proposed to treat the problem of modeling and implementing data warehouses according to these models. These works can be classified into two main categories:

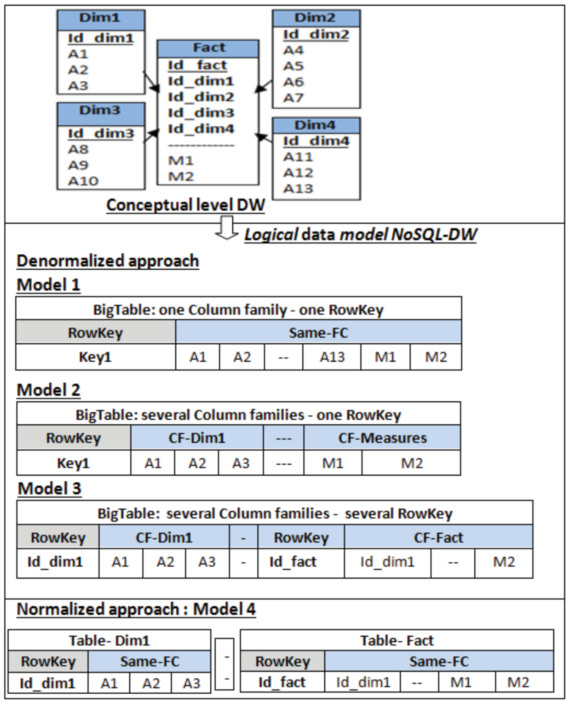


Fig. 1: Data warehouse-Columnar logical models

A) Denormalized approach: the aim of these works is to propose a storage schema that combines the fact and dimension tables into one table. 3 solutions are commonly applied for building the column family schema:

1) The first logical data model *(model 1)*: all attributes of fact and dimension tables are combined in one column family;
2) The second logical data model *(model 2)*: one column family for each dimension table and one column family dedicated for the fact table. *All column families are referenced by one RowKey*;
3) The third logical data model *(model 3)*: one column family for each dimension table and one column family dedicated for the fact table. But, *each column family is referenced by specific Rowkey.*

b) Normalized approach: it uses different tables *(Separate Tables)* for storing fact and dimension tables at a physical level. The fact table is stored into one table with one column family, each dimension table is stored into one table with one column family *(model 4)*.

It is worth noting the column family NoSQL databases provide variable-width tables that can be partitioned vertically and horizontally across multiple nodes of a cluster. Moreover, the RowKey ensures a data horizontal partitioning mechanism and the column family ensures a data vertical partitioning mechanism. In the following, we address the disadvantages of naïve solutions:

Table 1: Naïve solutions shortcomings

Solution	Disadvantages
Model 1	- Loss of the benefits of the vertical partitioning
Model 2	- Imbalance between column families - No control the number of column families can be generated
Model 3	- Imbalance between different column families - No control the number of column families can be generated - Loss of the benefits of the horizontal partitioning - Create a special join *(java codes)* between column families
Model 4	- Create a special join *(java codes)* between column families - Loss of the benefits of horizontal and vertical partitioning

Other research efforts tried to enhance the performance of the data schema by optimizing column family schema. In [7], to speed searching and have direct access to data blocks in column families structure, the authors propose a promising approach, in which they use composite indexes on the HBase table. In [8], to solve the problem of distributing attributes between column families, they implemented, by intuition, the data warehouse in an HBase table with two column families. The first one groups the attributes of fact and dimension tables more frequently interrogated. The second column family contains the attributes of the other dimensions. But the authors do not take into consideration clustering techniques to create the column families that contain the required data for processing a query or multiple queries. In [9], the authors propose an automatic approach based on a Genetic Algorithm to optimize the column family schema in HBase. The authors did not focus on the data warehouse implementation process on column family NoSQL databases. They evaluate their approach by using simple data sets and basic queries. In addition, the authors interested only in the impact of the number of column families in data physical schema performance. [10] proposes a cost-based approach, called NoSQL Schema Evaluator (NoSE), that can recommend a specific schema optimized for data storage in column family NoSQL databases. However, the proposed schema design provides more efficient access to data only for queries that involve a small number of attributes, this is not always the case for data warehouses.

3. Problem Statement

The implementation of a data warehouse that incorporates the best features of the column family NoSQL systems (scalability, aggregation capabilities and data partition options like the vertical and horizontal partitioning) is the goal of several research works. We have seen that implementation processes of the data warehouse based on these systems usually use denormalized approaches. In addition, these works are based essentially on only one input parameter: the conceptual model or the relational logic model. These are used as input parameters in the phases of the column family schema design process based on certain rules of transformation between the relational schemas to NoSQL schemas. In this case, the NoSQL modeling remains dependent on the relational modeling of the data warehouse, that can be suggested as the more generalized design process. It should be noted that the column family in HBase, Cassandra and BigTable can be considered a physical mechanism used to vertically distribute the writing and query load across the cluster's nodes. Therefore, an appropriate column family schema design can help in tuning the data warehouse's performance.

In this work, we tried to find new data models to improve the response times to the complex queries. We looked at the impact of other column family specific parameters on the performance of a data warehouse. To do this, we propose a clustering-based approach for column family schema construction to improve query gain performances. Our goal is to minimize the total amount of data scanned while performing OLAP query by optimizing: 1) The number of column families with a vertical pre-partitioning of data warehouse schema before its implementation; 2) The number of columns in column families. In this respect, to reduce the impact of workload change on data distribution strategy adopted, we tried to divide the initial set of queries into groups of queries so as to create a physical data warehouse schema consisting of separate storage of tables based on query groups, where a set of data that exactly matches a specific query group is stored in each table. This partitioning makes it possible to rebuild part of the initial data schema with a minimum amount of intervention from the DBA when the query workload is changed.

4. The proposed Approach

Our strategy, to create the Columnar NoSQL Data Warehouse (*CN-DW*), is subdivided into two phases that are detailed in the following sub-sections.

4.1 Query groups construction

The first optimization phase consists of applying a vertical partitioning to create data schema which consists of many separate tables according to the business requirements. We use query grouping strategy to summarize relevant data in several small tables which would be more manageable. That is, we specify a subset of attributes to create a table which contains rows with fewer columns, and is keeping enough data to run a group of the most similar queries, instead of processing all records storing in a single large table with high number of attributes. The process of queries clustering has three main steps, which are as follows:

1) Building the Attributes-Queries Matrix (AQM)
This step consists in processing the set of attributes relating to the initial query workload. To build AQM, we have taken into account all the attributes present in each query *(those that appear in the Select and Where clauses, except for the attributes of the join predicates)*. Let the workload consists of a set of most frequent queries $Q = \{q_1, q_2, .., q_m\}$, that access the set of attributes $R = \{a_1, a_2, ..., a_n\}$. The matrix AQM represents couples $((q_i)_{i=1,..m}, (a_j)_{j=1,..n})$, where general term AQ_{ij} equal to 1 if a_j appears in query q_i and to 0 otherwise.

2) Building the Queries-Distance Matrix (DQM)
In the second step, from the obtained (AQM) matrix, we build a Queries-Distance matrix DQM which is a (n x n) symmetric matrix, n is the number of queries in the initial workload, and its elements are the distances between two queries. This distance defines the queries proximity, that is if the distance between two queries is higher, then both the queries have less number of attributes in common. In our case, to compute the distance between two queries q_i and q_j, we used the Jaccard distance formula [3], a simple statistical method is usually used to calculate the similarity between two boolean data objects. It is defined as follows:

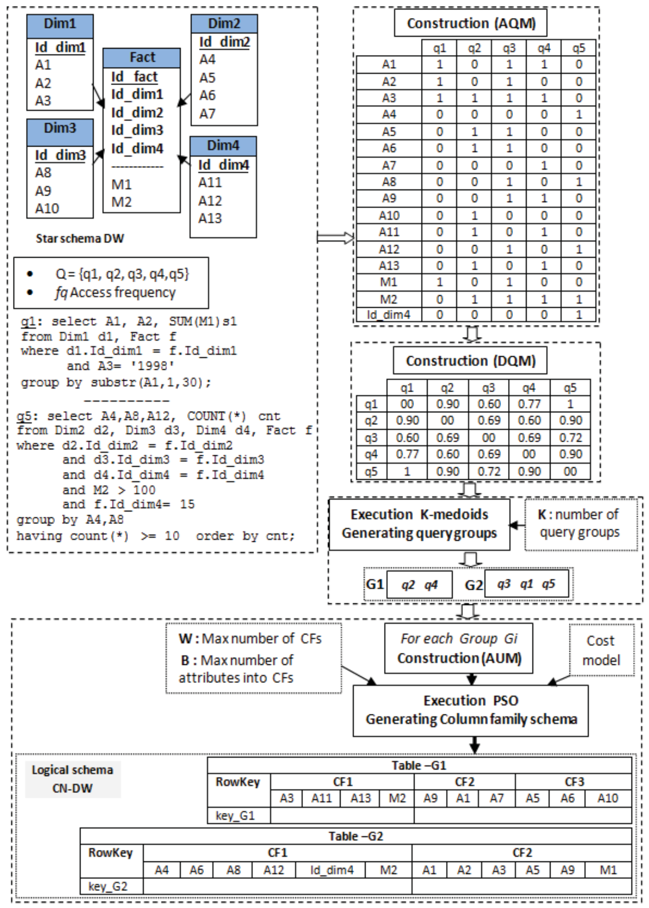
$$Distance(q_i, q_j) = 1 - Similarit(q_i, q_j) \tag{1}$$

$$Similarit(q_i, q_j) = \frac{a}{(n-d)} \tag{2}$$

In the equation (2), a represents the total number of attributes that have a value 1 in both q_i and q_j, d represents the total number of attributes which have a value 0 in both q_i and q_j, n represent the total number of attributes used by the set of queries Q, the Jaccard distance ranges from 0 to 1. To illustrate, let $T = \{Fact, Dim2, Dim2, Dim3, Dim4\}$ be the set of the warehouse tables. The workload consists of a set of queries $Q = \{q_1, q_2, q_3, q_4, q_5\}$ that access the set of attributes $R = \{A1, A2, .., A13, M1, M2\}$. Figure (2) shows the AQM and DQM corresponding to five queries, for example, to measure the distance between q_1 and q_3: (total attributes $n = 16$), ($a = 4$ of attributes 1), ($d = 6$ of attributes 0): ($Distance(q_1, q_3) = 1 - \frac{4}{(16-6)} = 0.6$), between q_1 and q_5 : ($n = 16$), ($a = 0$ of attributes 1): ($Distance(q_1, q_5) = 1$).

3) Query clustering based on *K-medoids method*
To do that, we chose to use the *K-medoids* algorithm [11] a variant of the *K-means* Algorithm. *K-medoids* can use

[3]https://en.wikipedia.org/wiki/Jaccard_index



```
q1: select A1, A2, SUM(M1)s1
from Dim1 d1, Fact f
where d1.Id_dim1 = f.Id_dim1
      and A3= '1998'
group by substr(A1,1,30);
          ----------
q5: select A4,A8,A12, COUNT(*) cnt
from Dim2 d2, Dim3 d3, Dim4 d4, Fact f
where d2.Id_dim2 = f.Id_dim2
      and d3.Id_dim3 = f.Id_dim3
      and d4.Id_dim4 = f.Id_dim4
      and M2 > 100
      and f.Id_dim4= 15
group by A4,A8
having count(*) >= 10  order by cnt;
```

	q1	q2	q3	q4	q5
A1	1	0	1	1	0
A2	1	0	1	0	0
A3	1	1	1	1	0
A4	0	0	0	0	1
A5	0	1	1	0	0
A6	0	1	1	0	0
A7	0	0	0	1	0
A8	0	0	1	0	1
A9	0	0	1	1	0
A10	0	1	0	0	0
A11	0	1	0	1	0
A12	0	0	1	0	1
A13	0	1	0	1	0
M1	1	0	1	0	0
M2	0	1	1	1	1
Id_dim4	0	0	0	0	1

	q1	q2	q3	q4	q5
q1	00	0.90	0.60	0.77	1
q2	0.90	00	0.69	0.60	0.90
q3	0.60	0.69	00	0.69	0.72
q4	0.77	0.60	0.69	00	0.90
q5	1	0.90	0.72	0.90	00

Fig. 2: Clustering-based method to design a columnar NoSQL data warehouse

the similarity measure (instead of the euclidean distance) to compute distances between points. These distances can then be used for breaking the data points up into K groups known a priori, which gives an advantage for *K-medoids* clustering method. In our case, this algorithm inputs vectors of the DQM matrix and the number of groups K. It attempts, first to find a initial subset of queries in workload Q designated as the centers (called *medoids*) of that cluster. Then it attempts to minimize the total distance between medoids and all the other queries identified to be in a cluster. The aim of this distance measurement is to optimize an objective function for measuring the quality of the partition. It is defined as follows:

$$Costdistance = \sum_{j=1}^{K}\sum_{i=1}^{m} Distance_j(q_i, M_j) \tag{3}$$

In the equation (3), $K < m$, $(M_j)_{j=1,..,K}$ is the j^{th} *Medoid*, $Distance_j$ is the distance between the query $(q_i)_{i=1,..,m}$ and the *nearest medoid* M_j.

Note that the most time-consuming part of the *K-medoids algorithm* is the calculation of the distances between queries. To eliminate this intricacy, we use the precomputed distance matrix. For example, to divide the queries $(q_1, q_2, q_3, q_4, q_5)$ into two clusters $(K = 2)$. It works as follows:

Input: matrix DQM, $K = 2$, Max number of iterations
Step 1: Initialize 2 medoids $(K = 2)$
1.1. Let us assume 2 cluster centers $M1 = (q1)$ and $M2 = (q2)$ are selected randomly as initial medoids.
1.2. Assigning each query to the group with the nearest medoid the queries $q1$ and $q3$ are closer to $M1$, so they form one group $G1 = \{q1, q3\}$, the queries $q2, q4$ and $q5$ form another group $G2 = \{q2, q4, q5\}$
1.3. Calculate the total cost of this clustering by using the formula (3): $Costdistance = 2.10$
Step 2: swap medoids
2.1 Select randomly new medoids queries. Let us assume the new medoids are $q4$ and $q5$
2.2 Update the current cluster according to the new nearest medoids, the new groups obtained: $G1 = \{q4, q1, q2, q3\}, G2 = \{q5\}$
2.3 Calculate the new $Costdistance = 2.06$
2.4 Check if the $Costdistance$ decreases
`if YES`: the new medoids becomes the medoids of the cluster
`Otherwise`: If iteration < maximum number of iterations: *go back to the Step 2* Else *stop the algorithm*
Output: A partition of the queries in 2 groups characterized by their medoids $M1$ and $M2$

Here, at the end of the algorithm, *K-medoids* outputs: $G_1 = \{q_2, q_4\}$, $G_2 = \{q_3, q_1, q_5\}$, This fits with: $Costdistance = 1.92$, $M_1 = (q_2)$, $M_2 = (q_3)$. At this point, we can define the first vertical fragmentation schema of the initial data warehouse. This schema that needs two tables for representing the data structures of the subset of columns corresponds to query groups G_1 and G_2. For example, we keep the subset of the columns corresponds to G_1 and G_2 in two separate tables T_{G1}, T_{G2}, respectively. So, T_{G1} and T_{G2} can contain the following attributes: $T_{G1} = (A1, A3, A5, A6, A7, A9, A10, A11, A13, M2)$ and $T_{G2} = (A1, A2, A3, A4, A5, A6, A8, A9, A12, M1, M2, Id\_dim4)$.

We see that there are some duplicate attributes $(A1, A3, A6, A9, M2)$ in both table, which causes an extra data storage cost. However, the use of these implementations is a better fit for data warehousing in NoSQL databases. In fact, this can be explained by the inability to join tables in the query languages of the column family NoSQL systems. Generally, this absence of the join is compensated by a denormalization process (by adding redundant data in tables or by grouping data from various tables into a single table) so that the data can quickly be read together. Therefore, one of the key challenges of this approach is how to minimize the storage space for redundant data. To satisfy this requirement, when creating tables, we assume some restriction, such as: (1)- A simple solution is to keep row keys, column family names and column names as small as possible. Because in column-family NoSQL databases, each record of a table can contain the row key value, column family names and column names, column value and timestamp, for example, *preferably one or two characters for column family names*; (2)- Maximum number of query groups ($K \leq Int(\frac{m}{2})$), where m the total number of query workloads, because too many query groups can increase the number of duplicate attributes; (3) The new requests select the existing query groups (instead of creating new ones) based on their similarity to the *medoids* of each target group, taking into account the closest distance.

4.2 Column family schemas construction

In this second phase, our goal is to generate the column family schema, that optimizes data access for queries that are in the same group. Our solution is to implement a process of grouping the attributes that are frequently queried together. This grouping will form set of column families that make up the logical schema of the *(CN-DW)*. We chose to use the meta-heuristic *Particle Swarm Optimization (PSO)* algorithm [12]. This algorithm is inspired by the swarms of insects or animals and their displacement in groups to find needs. *PSO* processes a population (called *swarm*) of ($N \geqslant 2$) *particles*. The process starts with a random initialization of the swarm in the search space. During the optimization process, at each iteration, each particle is moving according to a velocity. To do this, it linearly combines three fundamental information: - its current speed; - its best position (until to the i^{th} iteration); - the best position of its neighbors in the entire swarm. Our choice to use *PSO* is motivated by the fact: (1) It is a dynamic approach; (2) It allows us to define two input parameters: the maximum number of groups to be constructed and the maximum number of individuals in a group. This proves to be an advantage as long as we want to control, on the one hand, the number of column families that can be created, on the other hand, the number of columns in column families. This helps to build well-balanced column families. To make it simple, to improve the design of the column family schema for the table T_g corresponding to the query group G, the optimization problem can be defined as follows:

- $G = \{q_1, q_2, ..., q_M\}$: set of M query; $P = \{a_1, .., a_N\}$: set of N attributes used by G; fq_l: access frequency related to a query $(q_l)_{l=1,..,M}$; W: maximum number of column families ($2 \leq W$); B: maximum number of attributes in a column family $(Int(\frac{N}{W}) \leq B \leq Int(\frac{N}{W}) + 1)$
- $S = \{S_1, S_2, ..., S_z\}$: the set of all realizable solutions, where z is the iterations number of the *PSO* algorithm, $(S_i)_{i=1,..,z} = \{CFi_1, ..., CFi_W\}$, where $(CFi_j)_{j=1,..,W}$ are subsets of attributes, such as: 1)$\forall CFi_j \in S_i : CFi_j \subset P$; 2)$\forall a \in P : \exists CFi_j \in S_i : a \in CFi_j$; 3)$\forall CFi_j, CFi_h \in S_i : CFi_j \cap CFi_h = \varnothing$
- F : An objectif function that takes its values on S.

The problem is to find a solution $S^* \in S$ witch optimizes the value of the objectif function F such as: $\forall (S_i)_{i=1,..,z} \in S : F(S^*) \leq F(S_i)$.

The objectif function F allows to measure the quality of S_i solutions obtained after each iteration of *PSO*. Our cost function based on the works of [13]. Initially, this function is computed using the *Square Error (E^2)*, taking account of the access frequency of queries. The *(E^2)* of the attribute groups schema (S_i) is calculated as follow:

$$E^2_{S_i} = \sum_{j=1}^{W} \sum_{l=1}^{M} [(f_{q_l})^2 \times \alpha_j^{q_l} (1 - \frac{\alpha_j^{q_l}}{\beta_j})] \qquad (4)$$

($\alpha_j^{q_l}$) is the number of attributes in CFi_j appearing on a schema S_i accessed by the query q_l, (β_j) is the total number of attributes in CFi_j. Also, more the $E^2_{S_i}$ value approaches 0, more optimum is this grouping schema. In algorithm 1 we present a general version of the *PSO*, which exploits the objective function F to build the column families. *(we do not have space to present all part of the PSO).*

Algorithm 1 PSO-CF algorithm

```
Input: AUM Attribute Usage Matrix corresponding to G and P, AFM Access Frequency Matrix, W: maximum number of column families, B: maximum number of attributes in a column family, VPSO: Set of the variants of PSO
Notation: $S_{init}$: initial attribute groups; $S_i$: $i^{th}$solution; F : PSO objectif function; $S_{opt}$ : optimal attribute groups
begin
Randomly initialize: $S_0$ = Swarm($S_{init}$)
while Stop criterion is not satisfied do
    1- Determine the best position of all or part of swarm
    2- Particle displacement depending on the adopted strategy
    3- Structural adaptations by executing Split and Merge functions
    4- Update the velocity and position of the particles
    5- Evaluate the objectif function $F(E^2, S_i)$
    6- If Best Solution : $S_{opt}$ = $S_i$
end while
return ($S_{opt}$) /* Optimal column family schema */
End
```

5. Implementation, experiments and results

1. Dataset: To evaluate our approach, we used the TPC-DS benchmark [4]. The TPC-DS uses a constellation schema which consists of 17 dimension tables and

[4]Benchmark (TPC-DS) v2.0.0, http://www.tpc.org/tpcds/.

7 fact tables. In our case, we used the STORE_SALES fact table and its 9 dimension tables (CUSTOMER, CUSTOMER_DEMOGRAPHICS, CUSTOMER_ADDRESS, ITEM, TIME, DATE, HOUSEHOLD_DEMOGRAPHICS, PROMOTION, STORE). The DSDGEN data generator of TPC-DS allows to generate data files in a (*file.data*) format with different sizes according to a *Scale Factor (SF)*. We set *SF* to 100 which produces in STORE_SALES fact table (287.997.024 tuples).

2. Query workload: The TPC-DS benchmark offers 99 queries. We selected 19 separate queries (Table 2) that access 67 attributes, which exploit the entire schema of the STORE_SALES fact table and its dimension tables, using the operations *(selection, join, aggregate, projection)*. These queries compute the OLAP cubes with a gradually increasing number of dimensions. The degree of this dimensionality is divided into 3 levels: (*small: SD)*, (*medium: MD)* and (*large: LD)*, according to: (1) The number of tables used by a query; (2) The number of attributes and predicates for each query. It should be noted that our objective is to use TPC-DS benchmark to evaluate the performance of our technique when forming column families. Due to some requirements are not feasible with Apache Phoenix [5] (on query read capabilities) and HBase databases, these queries would require some modifications (syntax changes).

Table 2: Query characteristics

		Tables	Attributes	Predicates
SD	q1	1	4	4
	q2		9	3
	q3	2	9	10
MD	q4	3	4	4
	q5		4	5
	q6		6	12
	q7		6	7
	q8		6	7
	q9		6	6
	q10		9	6
	q11	4	6	5
	q12		7	7
LD	q13	5	10	22
	q14		10	13
	q15		10	10
	q16		11	8
	q17		11	23
	q18		12	15
	q19	6	14	15

3. Experimental configuration: To achieve our evaluation goals, we set up 2 storage environments. The first one is relational non-distributed with intel-core machine TMi7-4790S CPU@3.20 GHZ, 8 GB of RAM and 500 GB disk. It runs under the 64-bit Ubuntu-14.04 LTS operating system, which is used as a PostgreSQL server dedicated to the storage of the relational data warehouse *(data source)*. The second is a distributed NoSQL storage environment. It is a cluster of computers consisting of 1 master server *(NameNode)* and 3 slave machines *(Data Nodes)*. The *NameNode* has an Intel-Core TMi5-3550 processor CPU@3.30 GHZx4 with 16 GB RAM, and a 1TB SATA drive. Each of the *DataNodes* has an Intel-Core TMi5-3550 processor CPU@3.30 GHZx4 with 16 GB RAM and 500 GB of disk space. These machines run on 64bit Ubuntu-14.04 LTS and Java JDK 8. We used Hadoop (v2.6.0), MapReduce for processing, HBase (v0.98.8), ZooKeeper for track the status of distributed data in the *Region-Servers (DataNodes)*, Phoenix (v4.6.0) and SQuirreL SQL Client to simplify data manipulation and increase the performance of the HBase.

[5] https://phoenix.apache.org/

4. Tests and results: We chose three different methods, 2 already existing approaches in addition to our method, for implementing the *TPC-DS* in HBase system: (1) in the first one, the STORE_SALES fact table and its 9 dimension tables are stored into one HBase table with only one column family for all attributes (called *Flat schema*); (2) in the second, the STORE_SALES fact table and its 9 dimension tables are stored into one HBase table with 9 column families, each of the tables would correspond to a column family (called *Naïve schema*); (3) the last one consists of *CN-DW*, built according to our method with in addition three different schemas named: *schema A*, *schema B* and *schema C*. For the two first, (*schema A* , *schema B*), all queries are grouped in a same group G_1 *(k=1)*, i.e all data are stored in one HBase table, we varied the number of the column families $(W = 5, W = 10)$ corresponding to (*schema A* with a square error value $E^2_{w=5} = 3.94$) and (*schema B* with $E^2_{w=10} = 1.61$); in (*schema C*), the set of queries is divided in a 3 groups *(k=3) i.e three HBase tables)*, the number of the column families are determined according to each group $G_{i(1 \leq i \leq 3)}$. We executed all queries presented in (Table 2), on the five configurations described above. Note that, in this experiment, we do not want to make a performance comparison between the traditional relational database management systems and NoSQL databases. Our primary focus is to seek the main elements that have an impact on query execution time in a column oriented NoSQL data warehouse.

5. Discussion: We discus our results in this sub-section.

a) Impact of the data size on query execution time:

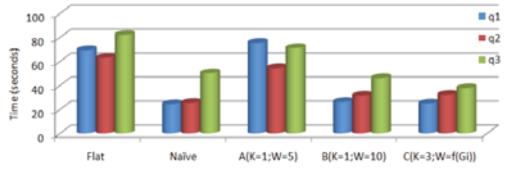


Fig. 3: SD Queries execution time

As shown in Figure 3, queries execution time increases significantly, when using the *Flat schema* or the *schema A*. These schemas records poor results compared with other schemas *(Naïve, schema B, schema C)*. In the *schema A*, these results are due to its poor quality *(caused by bad choices of the number of column families)*. But, in the *Flat schema*, these results are due to the pressure on the memory caused by the large amounts of data coming from the same column family (in *Flat* method: all facts and dimensions attributes are combined in one column family). Indeed, in HBase, the data, from a single column family, is stored in set of *HFiles*(*the number of HFiles depends on the data size in a column family)*. To read data in *Flat schema*, HBase will automatically solicit and load into a memory a large number of *HFiles*. This offers the possibility of performing multiple processing in memory, which results in increased execution time and decreased system performance. On the other hand, we observe a slight variation between the query execution times, run on the schemas *(Naïve, schema B, schema C)*. In the *Naïve schema* the queries relate 1 to 2 column families, in the *schema B* and *schema C* the queries use 2 to 3 column families. To respond to $q1$, $q2$ and $q3$, in these 3 schemas, HBase exploits column families having small data sizes. This allows it to considerably reduces the number of *HFiles* in memory *(that is a fewer HFiles of data are scanned during the query execution)*.

b) Impact of the number of column families on query execution time:

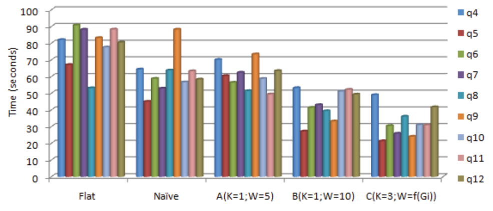


Fig. 4: MD Queries execution time

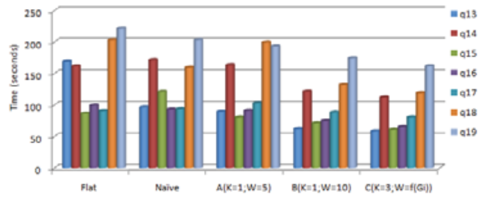


Fig. 5: LD Queries execution time

The objective of this experiment is to examine the scalability of our method when faced with variations in the number of dimensions. To do that, we executed 16 queries, $q4$ to $q19$, presented in (Table 2). These queries compute the OLAP cubes with a gradually increasing number of dimensions. In Figures 4 and 5, we observed that the proposed method gives better performance for all queries, whatever the number of dimensions used by query when using the *schema C*. When queries clustering process is applied $(k = 3)$ *groups*, the global execution time necessary for queries ($q4$ to $q19$) is 949.66 seconds. But, this time increases in the *schema B* $(k = 1)$, *Naïve schema* and *Flat schema*, we obtain 1115.16 seconds, 1489.75 seconds, 1742.97 seconds, respectively. From Figure 5, it can be seen that *Naïve schema* and *Flat schema* are equivalent in terms of response times for some complex queries (LD queries). This was foreseeable, indeed, the *Naïve* approach constructs the column families according to the principle where each dimension of the relational model must be transformed into a column family. So, to respond of these queries in the *Naïve schema*, HBase system solicits a very large number of column families, which generates a high cost of combinations and reconstruction of the intermediate results.

6. Conclusion

In this paper, we proposed a new method based on clustering techniques to create a column family schema adapted to our analysis needs to build a distributed data warehouse using a column family NoSQL Database. To evaluate our method, we carried out some experiments that using the TPC-DS benchmark and HBase database, several tests are made to evaluate the effectiveness of our method. The obtained results confirm the benefits of grouping techniques for the column families creation. In future work, we plan to extend the experiments to study the effects of other configuration parameters on data distribution in the context of parallel data warehousing such as data warehouse size, replication factor, and the number of nodes.

References

[1] J. Dean and S. Ghemawat: Mapreduce: simplified data processing on large clusters. USENIX Symposium on Operating Systems Design and Implementation, 2004.
[2] M. Stonebraker et al.: C-store: a column-oriented DBMS. In: Proceedings of the 31st Int. Conf. on VLDB Endowment: 553-564, 2005.
[3] B.F. Cooper et al.: Pnuts: Yahoo!'s hosted data serving platform. Proceedings of the VLDB Endowment: 1277-1288, 2008.
[4] K. Dehdouh et al.: Using the column oriented NoSQL model for implementing big data warehouses. In Int. Conf. on PDPTA: 469-475, 2015.
[5] M. Chevalier et al.: Implementation of multidimensional databases in column-oriented NoSQL systems. In Int. Conf. on ADBIS :79-91, 2015.
[6] R. Yangui et al.: Automatic Transformation of Data Warehouse Schema to NoSQL Data Base: Comparative Study. Procedia Computer Science, vol. 96, pp. 255-264, (2016).
[7] O. Romero et al.: Tuning small analytics on Big Data: Data partitioning and secondary indexes in the Hadoop ecosystem. In Information Systems: 336-356, 2015.
[8] L. C. Scabora et al.: Physical data warehouse design on NoSQL databases OLAP query processing over HBase. In Int. Conf. on Enterprise Information Systems, XVIII. INSTICC, p.111-118, (2016).
[9] F. Yang et al.: An Evolutionary Algorithm for column family schema optimization in HBase. Int. Conf. on IEEE BigDataService:439-445, 2015.
[10] Michael J. Mioret et al.: NoSE: Schema design for NoSQL applications. IEEE Transactions on Knowledge and Data Engineering: 2275-2289, 2017.
[11] L. Kaufman and P.J. Rousseeuw: Clustering by means of Medoids, in Statistical Data Analysis Based on the L1Norm and Related Methods, edited by Y. Dodge, North-Holland: 405-416, 1987.
[12] R. Eberhart and J. Kennedy: A new optimizer using particle swarm theory. In Proceedings of the siwth international symposium on micro machine and human science: 39-43, 1995.
[13] H. Derrar et al.: An Objective Function for Evaluation of Fragmentation Schema in Data Warehouse. In: Encyclopedia of Information Science and Technology, Third Edition, IGI Global: 1949-1957, 2015.