**International Journal of Decision Support System Technology**
Volume 12 • Issue 1 • January-March 2020

# Big Data Warehouse: Building Columnar NoSQL OLAP Cubes

Khaled Dehdouh, Computer Engineering Department of Cherchell Military Academy, TIPAZA Algeria
Omar Boussaid, ERIC Laboratory/ University of Lyon 2, Bron, France
Fadila Bentayeb, ERIC Laboratory/ University of Lyon, Lyon 2, Bron, France

## ABSTRACT

In the Big Data warehouse context, a column-oriented NoSQL database system is considered as the storage model which is highly adapted to data warehouses and online analysis. Indeed, the use of NoSQL models allows data scalability easily and the columnar store is suitable for storing and managing massive data, especially for decisional queries. However, the column-oriented NoSQL DBMS do not offer online analysis operators (OLAP). To build OLAP cubes corresponding to the analysis contexts, the most common way is to integrate other software such as HIVE or Kylin which has a CUBE operator to build data cubes. By using that, the cube is built according to the row-oriented approach and does not allow to fully obtain the benefits of a column-oriented approach. In this article, the focus is to define a cube operator called MC-CUBE (MapReduce Columnar CUBE), which allows building columnar NoSQL cubes according to the columnar approach by taking into account the non-relational and distributed aspects when data warehouses are stored.

## KEYWORDS

Big Data, Columnar, Data Warehouses, NoSQL Model

## INTRODUCTION

The data warehouse is a database for online analytical processing (OLAP) to aid decision-making. It is designed according to a dimensional modelling which has for objective to observe facts through measures, also called indicators, according to the dimensions that represent the analysis axes (Inmon, 1992). It is often implemented in the relational database management system (RDBMS) (Chaudhuri & Dayal, 1997) Thanks to the OLAP (On-Line Analytical Processing), the users can create multidimensional representations related to the particular analysis contexts in compliance with the specific needs, according to the criteria which they define, called hypercubes or OLAP cubes (Chaudhuri & Dayal, 1997). Cube computation produces aggregations that are beyond the limits of the Group by (Gray et al., 1997). For example, in the case of calculation of the sum, it computes in a multidimensional way and returns sub-totals and totals for all possible combinations. This involves performance of all aggregations according to all levels of hierarchies of all dimensions. For a cube with three dimensions A, B and C, the performed aggregations relate to the following combinations:

DOI: 10.4018/IJDSST.2020010101

(A, B, C), (A, B, ALL), (A, ALL, C), (ALL, B, C), (A, ALL, ALL), (ALL, B, ALL), (ALL, ALL, C), (ALL, ALL, ALL). The (A, B, C) combination corresponds as the lowest (least) aggregate level of the cube, and the rest are considered as the high aggregate levels. The advent of the big data has created new opportunities for researchers to achieve high relevance and impact amid changes and transformations in how we study several science phenomena. Companies like Google and Microsoft are analyzing large volumes of data for business analysis and decisions, which impact the existing and the future technologies (Gandomi & Haider, 2015).

However, unusual volumes of data become an issue when faced with the limited capacities of traditional systems, especially when data storage is in a distributed environment which requires the use of parallel treatment as MapReduce paradigm (Dear & Ghemawat, 2004). To solve a part of this issue, other models have appeared such as the column-oriented NoSQL (Not Only SQL) which gives a data structure more adequate to the massive data warehouses (Bhogal & Choski, 2015). In the big data warehouses context, a column-oriented NoSQL database system is considered as the storage model which is highly adapted to data warehouses and online analysis (Rabuzin & Modruan, 2014). Indeed, the storage of data column by column allows values belonging to the same column to be shared in the same disk space which improves the column access time enormously when the aggregate operations are performed. Furthermore, the non-relational aspect that characterizes the NoSQL model when data are stored allows to deploy data easily in a distributed environment (Jerzy, 2012).

To build OLAP cubes corresponding to the analysis contexts, the most common way is to integrate other softwares such as HIVE which has a CUBE operator to build data cubes. By using that, the cube is built according to the row-oriented approach and does not allow to fully obtain the benefits of a column-oriented approach. To solve this problem, we propose an aggregation operator, called MC-CUBE (MapReduce Columnar CUBE) which allows OLAP cubes to be computed according to the columnar approach from big data warehouses implemented by using column-oriented NoSQL model. MC-CUBE implements the invisible join, used by the columnar RDBMS (Abadi et al., 2008), in order to compute aggregation from several tables and extend it to take into account all possible aggregations at different levels of granularity of the cube. To deal with very large data, MC-CUBE uses the MapReduce paradigm when handling data stored in a distributed environment.

We have evaluated the performance of MC-CUBE operator on star schema benchmark (SSB) (O'Neil et al., 2007), implemented within the column-oriented NoSQL DBMS HBase1[1] using Hadoop2[2]. The HBase DBMS and the Hadoop platform were chosen because of their distributed context which was necessary for storing and analyzing big data.

The rest of this paper is organized as follows. Section 2 gives a related work. Section 3 introduces basic concepts about columnar NoSQL Data warehouse. Section 4 explains the columnar approach that we propose for building a data cube. Section 5 introduces the MC-CUBE operator and shows the execution phases through an example. Section 6 shows performance results and exemplifies of MC-CUBE operator when OLAP cubes are performed. Finally, Section 7 concludes the paper and suggests some possible directions for future research work.

## RELATED WORK

Big data have led data warehouses towards to distributed environments to store and to analyze the large amount of data. Since the column storage has outperformed the row storage, several research projects based on the columnar relational model have been commercialized such as InfoBright (Iezak & eastwood, 2009), Brighthouse (Iezak et al., 2008), Vectorwise (Zukowski et al., 2012), MonetDB (Idreos et al., 2012), SAP HANA (Farber et al., 2012), Blink (Barber et al., 2012), and Vertica (Lamb et al., 2012). These systems have led legacy RDBMS vendors to add columnar storage options to their existing engines (Larson et al., 2012). However, the relational model that is often used for storing data warehouses has shown its limits. Indeed, the use of distributed solutions based on the relational model is as costly as the implementation of the referential integrity constraints

(RIC) that ensures the validity of relation between tables in the RDBMS, because the system must continuously ensure that the associated data are stored in the same node. Furthermore, it is very difficult to respect the ACID (Atomicity, Consistency, Isolation and Durability) properties that characterize the transactional systems with correct consistency when data are stored in distributed environment (Cattell, 2011). Moreover, the variety aspect of data cannot be supported by the relational model. Thus, for analytical purposes, Google developed a massively scalable infrastructure that includes a distributed system, a column-oriented storage system, a distributed coordination system and parallel execution algorithm based on the paradigm of MapReduce, and opened them to the community in 2004 and 2006, respectively (Chang et al., 2008).

Regarding the implementation of the big data warehouses within the column-oriented NoSQL model, two approaches are defined (Dehdouh et al., 2015). The first one (normalized approach) uses different tables for storing fact and dimension at physical level which requires to achieve the join between tables when aggregation is performed. The second approach (denormalized approach) stores the fact and dimensions into one table, which allows to avoid performing join between tables.

For building OLAP cubes, we would name four relevant works (Abello et al., 2011; Dehdouh et al., 2014; Chevalier et al., 2015; Chavan & Phursule, 2014). Firstly, outlines the possibility of having data in a cloud by using columnar NoSQL DBMS to store data and MapReduce as an agile mechanism to build cubes. However, the cube is built at one level of granularity (least level) and from data warehouse implemented according the denormalized approach (Abello et al., 2011). Thus, the authors do not give any indications regarding the computing of the aggregates when the other levels of granularities that compose the cube are performed. A similar proposal to build cubes using MapReduce can be found in (Dehdouh et al., 2014), but in this case the authors just perform the cube at different level of granularity and do not provide a solution for performing join between tables (fact and dimensions) when data warehouses are implemented according the normalized approach. In (Chevalier et al., 2015), the different level of granularity that compose the OLAP cube can be computed using the naive method, using a combination of group-by queries and gathering the outputs via the UNION operator. This solution is not suitable for big data warehouses. Indeed, for D dimensions, the execution of $2^D$ sub-queries to perform the various aggregations, which considerably increase the number of times when the data warehouse is accessed. Consequently, the naive method reduces DBMS performance, particularly when scaling-up. Finally, (Chavan & Phursule, 2014) gives a solution to perform join between tables and performing aggregates from data warehouses implemented according the normalized approach. It consists to integrate software and tools such as Hive and Kylin in the ecosystem used for implementing the data warehouse, and use their cube building operators. However, the use of these tools is limited; because they are row-oriented, and once integrated into column-oriented NoSQL DBMS, they do not allow to take a full effective of the columnar NoSQL DBMS when data are handled.

## COLUMNAR NOSQL DATA WAREHOUSE

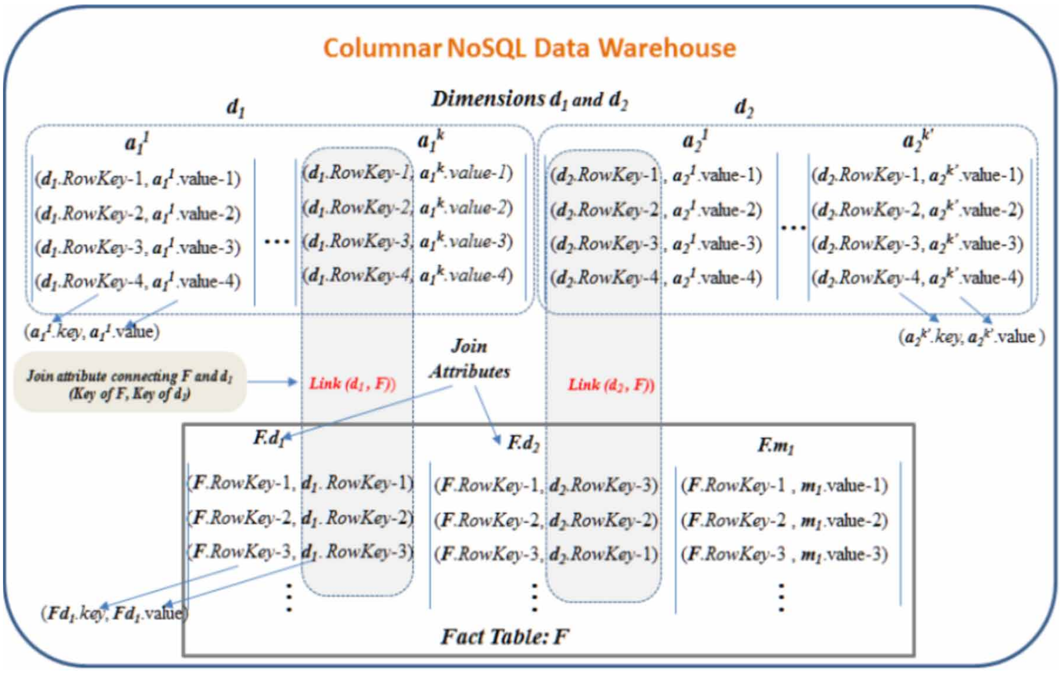
In the case of data warehouse implementation by using the columnar NoSQL model, both the dimensions and the fact are stored by column-wise. Each data is stored in the form of a “key/value” pair. When the data belongs to the dimension, the key part is represented by the key of this dimension. By cons, if the data belongs to the fact table, the part key is represented by the key of the fact table. As the dimensions and fact are stored in different tables, the link between them is ensured by the join attributes, where the key contains the key of the fact table and the value part contains the key of the dimension.

**Formalization:** Given the columnar NoSQL star data warehouse DW which is composed by the fact table F and dimensions tables $D = \{d_1, d_2, ..., d_n\}$. Each dimension table $d_j$, with $j$ € $[1, n]$ is composed by $a^i_j$ attributes with $i \in [1, k]$, such as $d_j = \{a^1_j, a^2_j, ..., a^k_j\}$, where k varies from dimension to another. Each dimension attribute value is stored in the form of a "key/value" pair. The key part contains the dimension identifier $d_j$.RowKey and the value part contains $a^i_j$.*value*. The fact table, on the other hand, groups together a number of attributes which represent the measures to aggregate $M = \{m_1, m_2, ..., m_t\}$. Each measure $m_q$, with $q \in [1, t]$, is identified by the fact table key (F.RowKey, $m_q$.value). Furthermore, F contains the join attributes F.$d_j$, where each one is composed by the fact table identifier and the identifier of dimension table that represent (F.RowKey, $d_j$.RowKey). Columnar NoSQL star data warehouse composed of one fact table F and two dimension tables $d_1$, $d_2$ is depicted in Figure 1.

## BUILDING COLUMNAR OLAP CUBES

To generate an OLAP cube compound aggregates of several levels of granularity, it is necessary to firstly identify data which satisfy all the predicates (filters) from the data warehouse. To carry out this phase, query predicates are applied separately to the respective dimensions to obtain the primary keys of dimensions that will be involved in the join with the fact table. Since the keys are foreign keys at the level of the fact table, they are used to identify the related fact table key. The result of this phase allows then to calculate the different levels of granularity that compose the cube. When a measure function applied is holistic, the calculation of aggregates of the different granularity levels will be from data that satisfy predicates. However, when the measure function is distributive or algebraic (which interests us), it will be better to perform the calculation of aggregates of the lowest level and then use it to compute the other levels of aggregations that compose the data cube for the sake of optimizing the data cube calculating. This allows to avoid returning to data warehouse and performs more join between tables when higher level aggregates are performed. We present here in after the general way for computing aggregates of several levels of granularity of the OLAP cube obtained according to the row-oriented approach, versus, our proposition for building the cube according the column-oriented approach.

**Figure 1. Columnar NoSQL data warehouse**


Columnar NoSQL Data Warehouse
Dimensions $d_1$ and $d_2$
$d_1$
$d_2$
$a_1^1$
$a_1^k$
$a_2^1$
$a_2^{k'}$
($d_1$.RowKey-1, $a_1^1$.value-1)
($d_1$.RowKey-2, $a_1^1$.value-2)
($d_1$.RowKey-3, $a_1^1$.value-3)
($d_1$.RowKey-4, $a_1^1$.value-4)
($d_1$.RowKey-1, $a_1^k$.value-1)
($d_1$.RowKey-2, $a_1^k$.value-2)
($d_1$.RowKey-3, $a_1^k$.value-3)
($d_1$.RowKey-4, $a_1^k$.value-4)
($d_2$.RowKey-1, $a_2^1$.value-1)
($d_2$.RowKey-2, $a_2^1$.value-2)
($d_2$.RowKey-3, $a_2^1$.value-3)
($d_2$.RowKey-4, $a_2^1$.value-4)
($d_2$.RowKey-1, $a_2^{k'}$.value-1)
($d_2$.RowKey-2, $a_2^{k'}$.value-2)
($d_2$.RowKey-3, $a_2^{k'}$.value-3)
($d_2$.RowKey-4, $a_2^{k'}$.value-4)
($a_1^1$.key, $a_1^1$.value)
($a_2^{k'}$.key, $a_2^{k'}$.value)
Join attribute connecting F and $d_j$ (Key of F, Key of $d_j$)
Link ($d_1$, F)
Join Attributes
Link ($d_2$, F)
F.$d_1$
F.$d_2$
F.$m_1$
(F.RowKey-1, $d_1$.RowKey-1)
(F.RowKey-2, $d_1$.RowKey-2)
(F.RowKey-3, $d_1$.RowKey-3)
(F.RowKey-1, $d_2$.RowKey-3)
(F.RowKey-2, $d_2$.RowKey-2)
(F.RowKey-3, $d_2$.RowKey-1)
(F.RowKey-1, $m_1$.value-1)
(F.RowKey-2, $m_1$.value-2)
(F.RowKey-3, $m_1$.value-3)
($Fd_1$.key, $Fd_1$.value)
Fact Table: F

## Row-Oriented Approach

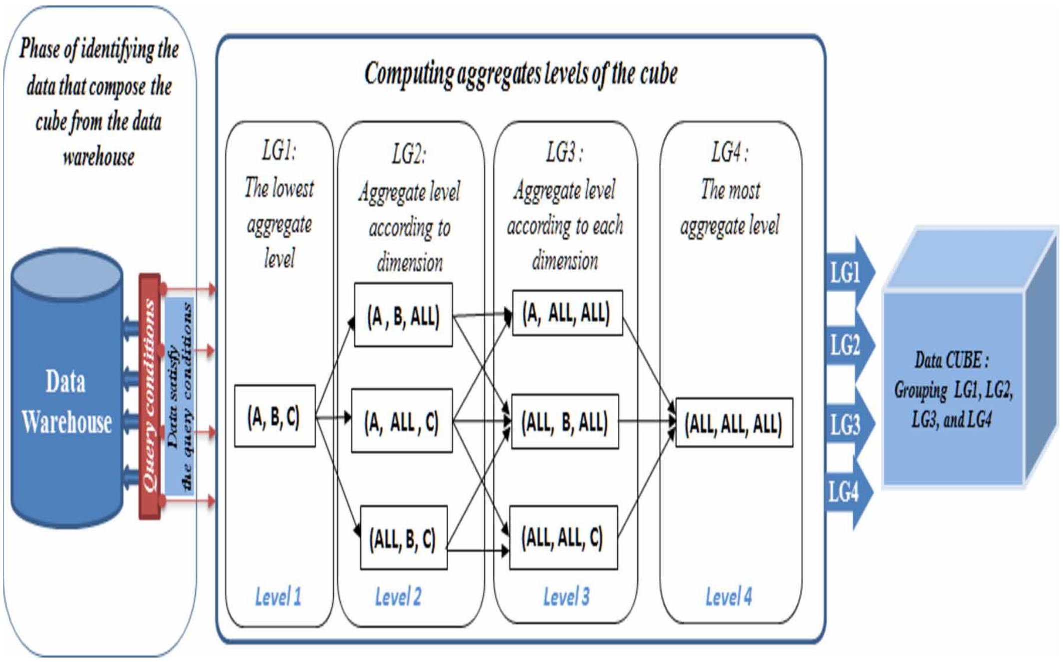
The storage of data is row-oriented; each table (dimension or facts) stores data by row (record). Each record stores the column (attribute) values which is a row of the table. Once the column value belonging to a record is accessed, the other values of columns that compose this record do the same. Therefore, to optimize the computation of aggregates when the cube is built, and after the identification of data that compose the cube, it would be more adequate to compute the most aggregate level from the least one, and to be used for computing the one after as it contains fewer attributes (Beyer & Ramakrishnan, 1999).

For instance, for a cube with three dimensions A, B, C, the computing of aggregates is defined according to four levels of granularity (LG1, LG2, LG3, LG4) where LG1 represents the least aggregate level (A, B, C) and LG2 represents aggregation level corresponding to dimensions' combinations and LG3 represents aggregation level corresponding to each dimension and LG4 represents the most aggregate level.

To optimize building OLAP cube, the LG1 which represents the least aggregate level (A, B, C) is performed first; then, it is used to compute aggregates of other levels of granularity (LG2, LG3, LG4). The transition from one level of granularity to another is carried out sequentially with a downward direction (LG2 → LG3 → LG4). Thus, the second level of granularity LG2: (A, B, ALL), (A, ALL, C), (ALL, B, C) allows to compute the third LG3: (A, ALL, ALL), (ALL, B, ALL), (ALL, ALL, C), that can be used to compute the next level LG4: (ALL, ALL, ALL) (Figure 2).

Thus, according to the row-oriented approach, the fourth level of granularity can be obtained only if the third level of granularity has already been performed. This approach (used by the traditional DBMS), based on the exploitation of least aggregate levels of granularity to perform the most aggregate levels, is effective when the storage environment is row-oriented.

Figure 2. Row-oriented approach for building the cube


Phase of identifying the data that compose the cube from the data warehouse
Data Warehouse
Query conditions
Data satisfy the query conditions
Computing aggregates levels of the cube
LG1: The lowest aggregate level
LG2: Aggregate level according to dimension
LG3: Aggregate level according to each dimension
LG4: The most aggregate level
(A, B, C)
(A, B, ALL)
(A, ALL, C)
(ALL, B, C)
(A, ALL, ALL)
(ALL, B, ALL)
(ALL, ALL, C)
(ALL, ALL, ALL)
Level 1
Level 2
Level 3
Level 4
LG1
LG2
LG3
LG4
Data CUBE: Grouping LG1, LG2, LG3, and LG4

### Column-Oriented Approach

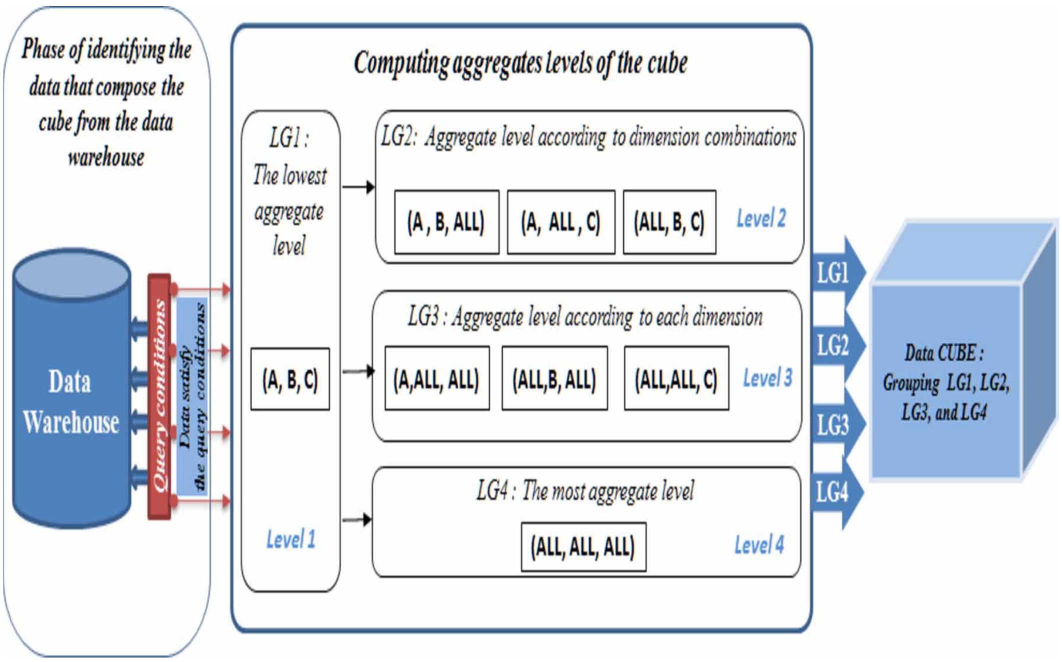
When data warehouse is stored according to column-oriented approach, the data is stored column by column. Therefore, the computing of aggregates requires access only to the values of columns (dimensions or measures) involved in the decisional query. For building the OLAP cube, and after identifying the data that compose the cube, it is not necessary to seek for reducing the number of attributes in order to compute a cube's aggregates at different levels of granularity. The aggregations of the second, the third, and the fourth levels can be performed from the first level. Also, the transition from levels 2 to 4 does not need to be sequential, and can be achieved at the same time in a parallel way. Thus, we propose this new approach for optimizing the computing of the cube where the aggregates granularity levels are all performed from one single level, which is the least aggregated one (Figure 3).

We are positioning in this approach to provide MC-CUBE an aggregation operator to generate OLAP cubes from column-oriented data warehouses.

## MC-CUBE OPERATOR

Traditionally, building OLAP cubes tools have been focused on the row-by row data layout, since it performs better on the most common application for database systems (transactional data processing). However, in the context of the data warehouses storage, where the column by column data layout offers a better opportunity than row by row, these cube building tools have been integrated through connectors as Kylin and Hive tools. To perform aggregates of the cube, the data which are stored column by column are used according to the row-oriented approach because these tools build the cube according to row-oriented approach and lead consequently to the loss of the column-by-column data layout opportunity. As a solution to this problem, the MC-CUBE operator extends the use of the column-oriented approach beyond data storage in order to achieve data processing and building OLAP cubes. MC-CUBE allows the following:

**Figure 3. Column-oriented approach for building the cube**


Phase of identifying the data that compose the cube from the data warehouse
Data Warehouse
Query conditions
Data satisfy the query conditions
Computing aggregates levels of the cube
LG1 : The lowest aggregate level
(A, B, C)
Level 1
LG2: Aggregate level according to dimension combinations
(A , B, ALL)
(A, ALL , C)
(ALL, B, C)
Level 2
LG3 : Aggregate level according to each dimension
(A,ALL, ALL)
(ALL,B, ALL)
(ALL,ALL, C)
Level 3
LG4 : The most aggregate level
(ALL, ALL, ALL)
Level 4
LG1
LG2
LG3
LG4
Data CUBE : Grouping LG1, LG2, LG3, and LG4

- To avoid the use of additional algorithms which are used by the row-oriented approach in order to better organize the steps of the aggregates computing based on exploiting of the intermediate results to obtain a level of granularity from the smaller results of another level of granularity;
- To perform aggregates processing of the higher levels independently (simultaneously) since each of these levels of granularity is calculated from the same finest level;
- To optimize the time needed to perform aggregates corresponding to the different granularity levels (lowest to highest);
- To reduce the build time of the cube.

## Operating Principle

In this section, we present MC-CUBE operator for building OLAP cubes according to NoSQL columnar-wise approach. The MC-CUBE uses the MapReduce paradigm to optimize the processing of massive data. Recall that, MapReduce is a framework that hides distribution of data, parallelization, fault tolerance and load balancing from the programmer. It has been specially designed for scalability and processing in a cloud. It allows to deal with huge volumes of data.

MC-CUBE operator allows to perform a cube in five phases. In the first phase, the MC-CUBE identifies the data which satisfy all the predicates (filters) from the columnar data warehouse and allows the aggregation according to all columns representing dimensions to be produced. Since, the fact and dimensions are stored in different tables, the MC-CUBE operator implements in distribute environment the invisible join, which is used by the relational column-oriented to perform the join between tables (fact and dimensions) and achieve the aggregation computing. It is worth to mention, the invisible join improves substantially join performance in the case of column stores, especially on the types of schemas found in data warehouses. The result of the first phase corresponds to the first level of granularity of the cube which allows then to perform the different levels of granularity that compose the cube. In the second phase, the first level of granularity (output of the first phase) is used to perform the aggregations of the granularity level according to each dimension separately. The third phase uses the output of the first one to produce aggregations of the dimension combinations. The fourth phase uses the output of the first one to computes the highest levels of granularity. The last phase groups together the outputs of the previous four phases in order to produce the cube.

## Execution Phases of MC-CUBE Operator

To deal with massive data processing, MC-CUBE operator uses paradigm MapReduce model for processing large volumes of data and optimizes the calculation of OLAP cubes. Indeed, with MapReduce programming model, MapReduce job specifies a “Map” function that processes a “key/value” pair to generate a set of intermediate “key/value” pairs, as well as a “Reduce” function that merges all intermediate values associated with the same intermediate key. The five phases for building OLAP cube by MC-CUBE are achieved by executing MapReduce jobs.

## The First Phase

This phase involves using the data warehouse to identify data which satisfy all the predicates (filters) and allows to produce the lowest level of granularity of the cube. Firstly, the query predicates are applied on dimensions tables to identify the pairs of dimensions keys that will be involved in the join with the fact table. In the case of a snowflake schema where the dimension hierarchies are divided into separate tables when the predicate is applied on the higher hierarchy level dimension, the keys that satisfy the condition are used to perform a join with the lower level dimension. This is repeated until the dimension directly related to the fact table is reached. The outputs are used to perform the join with the fact table in order to give the values of the primary key of the fact table which satisfy all query predicates. As soon as the join results identify the data that compose the cube, the aggregations of the lowest level of granularity of the cube are performed. The MapReduce jobs that perform the first phase are achieved as follows:

**Job 1 (MRJ-1):** The “key/value” pairs of dimensions which will be used to perform the join with the fact table are identified when this job is performed (Algorithm 1 and 2).

At the Map function, the treatments are parallelized in order to apply the query predicates to the respective dimensions (algo1: line2). The result is output to the Reduce function which gathered the partial results and provides a set of pairs “key/value” representing the dimension values involved in the join with the fact table. The key part of “key/value” pair contains the value of the dimension (algo2: line1) and the other part contains the dimension identifier corresponding (algo2: line3) (Figure 4).

**Job 2 (MRJ-2):** This job (algorithm 3 and 4), uses the result of the first, and identifies at the map function level, the values of the primary key of the fact table corresponding to the dimensions’ primary keys which satisfied the query predicates. This allows building a list of “key/value”, where the key part contains the primary key of the fact table, and the value part contains a boolean value (0 or 1). This latter shows if the query predicate is satisfied (1) (algo3: line4) or not (0) (algo3: line7). For optimization reasons, only the first join attribute at the fact table is scanned, the others are scanned by using the keys that are obtained from the first scan. Indeed, the first scan allows to have the values of the primary key of the fact table that satisfied the predicate’s dimension (boolean value corresponding to 1). Thus, from this set of keys that the others join attributes will be checked. This avoids scanning the values of the unnecessary primary keys of the fact table and reduces the treatment of the number of “key/value” pairs. The result of the Map function is used by the Reduce function which apply the logical AND on the boolean value belonging to the same key (algo4: line4). The set of keys that keeping their boolean value to 1, represents the values of the primary key of the fact table which satisfy all query predicates (Figure 5).

**Job 3 (MRJ-3):** The granularity level according to all columns representing dimensions that corresponds to the lowest (least) aggregation of the cube, that we call it R1, is achieved when this job is performed (algorithm 5 and 6). Indeed, the Map function uses the result of the job 2 to identify the keys of dimensions (algorithm 5: line3) and the values of measure corresponding

**Figure 4. Job 1 pseudo-code**

```
Algorithm 1: MC-CUBE - Job 1 of phase1 - Fonction Map()
  Input  : ($d_j.RowKey$, $a_j^i.value$): (key,value) pair of dimension.
           $C$ : query condition.
  Output: ($key_{tmp}$, $val_{tmp}$)
  /* Identify (key, value) of dimensions that verify condition query.   */
1 foreach $a_j^i \in d_j$ do
2     if $a_j^i.value$ verify $C$ then
3         $key_{tmp} \leftarrow a_j^i.value$
4         $val_{tmp} \leftarrow d_j.RowKey$
5     end
6     Emit ($key_{tmp}$, $val_{tmp}$)
7 end
```

```
Algorithm 2: MC-CUBE - Job 1 of phase1 - Fonction Reduce()
  Input  : ($key_{tmp}$, $val_{tmp}$): output of Map function.
  Output: ($key_{out}$, $val_{out}$): (key, value) pairs that verify condition query
  /* Set dimensions identifiers to join with the fact table   */
1 $key_{out} \leftarrow key_{tmp}$
2 foreach $v \in val_{tmp}$ do
3     $val_{out} \leftarrow concat(v)$ // Concatenate dimension identifiers verifying the query condition $C$
4 end
5 Emit ($key_{out-Job1}$, $val_{out-Job1}$)
```

**Figure 5. Job 2 pseudo-code**

```
Algorithm 3: MC-CUBE - Job 2 of phase1 - Fonction Map()
    Input  : ($F.RowKey$, $d_j.RowKey$): (key,value) pair of fact table.
             ($key_{out-Job1}$, $val_{out-Job1}$): output of Job1
    Output: ($key_{tmp}$, $val_{tmp}$)
    /* Identify the keys of fact table that verify condition query.          */
 1  foreach $v \in val_{out-Job1}$ do
 2      if $d_j.RowKey = v$ then
 3          $key_{tmp} \leftarrow F.RowKey$
 4          $val_{tmp} \leftarrow 1$
 5      else
 6          $key_{tmp} \leftarrow F.RowKey$
 7          $val_{tmp} \leftarrow 0$
 8      end
 9      Emit ($key_{tmp}$, $val_{tmp}$)
10  end
```

```
Algorithm 4: MC-CUBE - Job 2 of phase1 - Fonction Reduce()
    Input  : ($key_{tmp}$, $val_{tmp}$): Output of Map function.
    Output: ($key_{out-Job2}$, $val_{out-Job2}$): keys of fact tables that verified queries conditions
    /* Set keys of fact table that verified queries conditions.          */
 1  $key_{out-Job2} \leftarrow key_{tmp}$
 2  $val_{out-Job2} \leftarrow 1$
 3  foreach $v \in val_3$ do
 4      $val_{out-Job2} \leftarrow val_{out-Job2}$ & $v$
 5      if $val_{out-Job2} = 1$ then
 6          Emit ($key_{out-Job2}$, $val_{out-Job2}$)
 7      end
 8  end
```

(algorithm 5: line4), and sends it to the Reduce function in the form of "key/value" pairs (algorithm 5: line 6). The key part contains dimensions and the value part of pair contains the measure corresponding. At the second stage, the values of measure are aggregated according dimensions when the Reduce function is performed (algorithm 6: line1&3) and the least aggregate level of the cube (R1) is produced (Figure 6).

## The Second Phase

This phase allows, from the result obtained by job 3 of the first phase, to perform the aggregations according to each dimension separately. It is achieved by executing one MapReduce job as follows:

**Job (MRJ):** To achieve the aggregation level corresponding to each dimension that we call it R2, the result of the job 3 (R1) is used. Recall that the part key of pairs composed the R1 contains the dimensions identifiers, and the part value contains the corresponding measure. Thus, at the Map function level, each identifier of dimension is extracted (algorithm 7: line2) with the corresponding measure (algorithm 7: line3) which produces a pair of "key/value". The key part contains an identifier of dimension, and the part value contains the corresponding measure. For each same dimension identifier (key), the measures (value) are aggregated (algorithm 8: line3) when the Reduce function is performed (Figure 7).

## The Third Phase

It involves using the result obtained by job 3 of the first phase, to perform the aggregations according to dimension combinations. It is achieved by executing one MapReduce job, and we call the result of this operation R3.

**Figure 6. Job 3 pseudo-code**

```
Algorithm 5: MC-CUBE - Job 3 of phase1 - Fonction Map()
Input  : (($key_{out-Job2}$, $val_{out-Job2}$) : output of job 2.
         ($F.RowKey$, $d_j.RowKey$): (key,value) pair of fact table (join attribute $F.d_j$ ).
         ($F.RowKey$, $m_q.value$): (key,value) of fact table corresponding to measure attribute $m_q$.
Output: ($key_{tmp}$, $val_{tmp}$)
/* Identify the keys of dimensions and the values of measure corresponding.   */
foreach $v \in key_{out-Job2}$ do
    if $F.RowKey = v$ then
        $key_{tmp} \leftarrow concat(d_j.RowKey)$ //Concatinate dimensions' keys having the same fact table key
        $val_{tmp} \leftarrow m_q.value$
    end
    Emit ($key_{tmp}$, $val_{tmp}$)
end
```

```
Algorithm 6: MC-CUBE - Job 3 of phase1 - Fonction Reduce()
Input  : ($key_{tmp}$, $val_{tmp}$): Output of Map function.
Output: ($key_{out-R1}$, $val_{out-R1}$): (key, value) pairs that compose the cube $R_1$
/* Produce the least aggregate level of the cube (R1).   */
$key_{out-R1} \leftarrow key_{tmp}$
foreach $v \in val_{tmp}$ do
    $val_{out-R1} = Aggregate(v)$ // Apply the aggregate function
    Emit ($key_{out-R1}$, $val_{out-R1}$)
end
```

**Figure 7. Job pseudo-code of second phase**

```
Algorithm 7: MC-CUBE - Job of phase2 - Fonction Map()
Input  : ($key_{out-R1}$, $val_{out-R1}$): output of job 3
Output: ($key_{tmp}$, $val_{tmp}$)
/* Getting each identiier of dimension with measure corresponding.   */
foreach $v \in key_{out-R1}$ do
    $key_{tmp} \leftarrow d_j.RowKey$ // Getting each dimension separately
    $val_{tmp} \leftarrow m_q$
    Emit ($key_{tmp}$, $val_{tmp}$)
end
```

```
Algorithm 8: MC-CUBE - Job of phase2 - Fonction Reduce()
Input  : ($key_{tmp}$, $val_{tmp}$): Output of Map function
Output: ($key_{out-Job}$, $val_{out-Job}$): (key, value) pairs that compose the cube $R_2$
/* Produce the aggregations according to each dimension separately ($R_2$).   */
$key_{out-R2} \leftarrow key_{tmp}$
foreach $v \in val_{tmp}$ do
    $val_{out-R2} = Aggregate(v)$ // Applying the aggregate function
end
Emit ($key_{out-R2}$, $val_{out-R2}$)
```

**Job (MRJ):** The result of the job 3 of the first phase (R1) is used when the aggregation corresponding to the combinations of the dimensions is achieved. Thus, at the Map function level, the identifiers of dimension with different combinations are extracted with the corresponding measure (algorithm 9: line2&3) which produces in each case a pair of "key/value". The key part contains a combination of dimension identifiers, and the part value contains the corresponding measure. For each same combination of dimensions identifiers (key), the measures (value) are aggregated (algorithm 10: line3) when the Reduce function is performed (Figure 8).

## The Fourth Phase

It involves using the result obtained by job 3 of the first phase, to perform the aggregations according to dimension combinations. It is achieved by executing one MapReduce job, and we call the result of this operation R3.

**Job (MRJ):** The granularity level corresponding to the total aggregation, that we call it R4, is achieved when this job is performed. Thus, the Map function uses the result of job 3 of the first phase (R1) in order to extract the measure values and aggregate it when Reduce function is performed. This allows the most aggregate level of the cube (R4) to be produced (Figure 9).

## The Fifth Phase

Since the aggregates are performed with dimensions identifiers, this phase replacing these identifiers with the dimensions attributes cited in the query, and gathered it to provide the cube. It is achieved by executing one MapReduce job.

**Job (MRJ):** This job allows to adapt the results obtained in the previous phases to the query context. Indeed, at the Map function the identifiers of dimensions belonging to the R1, R2, and R3 are

**Figure 8. Job pseudo-code of third phase**

```
Algorithm 9: MC-CUBE - Job of phase3 - Fonction Map()
   Input  : ($key_{out-R1}$, $val_{out-R1}$): Output of job 3
   Output: ($key_{tmp}$, $val_{tmp}$)
   /* Getting combinations of the dimensions' identifiers with measure corresponding.  */
1  foreach $v \in key_{out-R1}$ do
2      $key_{tmp} \leftarrow$ combinat($d_j.RowKey$) // Getting combinations of the dimensions' identifiers
3      $val_{tmp} \leftarrow m_q$
4      Emit ($key_{tmp}$, $val_{tmp}$) // Emitting combinations of the dimensions' identifiers with measure
       corresponding
5  end
```

```
Algorithm 10: MC-CUBE - Job of phase 3 - Fonction Reduce()
   Input  : ($key_{tmp}$, $val_{tmp}$) : Output of Map function
   Output: ($key_{out-R3}$, $val_{out-R3}$) : (key, value) pairs that compose the cube $R_3$
   /* Produce the aggregations according to combinations of the dimensions ($R_3$).  */
1  $key_{out-R3} \leftarrow key_{tmp}$
2  foreach $v \in val_{tmp}$ do
3      $val_{out-R3} = Aggregate\ (v)$ // Applying the aggregate function
4  end
5  Emit ($key_{out-R3}$, $val_{out-R3}$)
```

Figure 9. Job pseudo-code of fourth phase

```
Algorithm 11: MC-CUBE - Job of phase 4 - Fonction Map()
  Input  : ($key_{out-R1}$, $val_{out-R1}$): Output of job 3
  Output: ($key_{tmp}$, $val_{tmp}$)
  /* Getting the measure corresponding to the total aggregations.      */
1 foreach $v \in key_{out-R1}$ do
2     $key_{tmp} \leftarrow ALL$ // Refers to the total aggregate
3     $val_{tmp} \leftarrow val_{out-R1}$
4 end
5 Emit ($key_{tmp}$, $val_{tmp}$)
```

```
Algorithm 12: MC-CUBE - Job of phase 4 - Fonction Reduce()
  Input  : ($key_{tmp}$, $val_{tmp}$) : Output of Map function.
  Output: ($key_{out-R4}$, $val_{out-R4}$) : (key, value) pairs that compose the cube $R_4$
  /* Produce the total aggregations.                                    */
1 $key_{out-R4} \leftarrow key_{tmp}$ foreach $v \in val_{tmp}$ do
2     $val_{out-R4} = Aggregate\ (v)$ // Applying the aggregate function  Emit ($key_{out-R4}$,
      $val_{out-R4}$)
3 end
```

changed by the values of dimensions cited in the query (algorithm 13: line 2). The results of this function are gathered when Reduce function is performed which produce the cube (Figure 10).

In order to set out the jobs that defined the MC-CUBE execution phases, we use in the next section an example.

## Deployment of Building a Cube by Using MC-CUBE

We use the Star Schema Benchmark as a data warehouse example in order to set out the different jobs required to perform the compute phases of the cube by using MC-CUBE operator from columnar NoSQL data warehouse. Recall that SSB is a data warehouse which manages line orders according to dimensions, PART, SUPPLIER, CUSTOMER and DATE. It consists of a single fact table called LINEORDER made up of seventeen columns to give the information about order, with a composite primary key consisting of the Orderkey and Linenumber attributes, and foreign keys that refer to the dimension tables.

In order to provide more detail on MC-CUBE execution phases, we illustrate our explanation with an example. This example computes the sales revenue from products delivered by FRENCH suppliers and for which orders were placed by ITALIAN customers in the years 1996 and 1997. According to the example, the four phases are performed as follows.

### The First Phase

**MJR-1:** The Map function applies query predicates Customer.nation = Italy and Supplier.nation = France and Date.year 1996 and Date.year 1997, to the appropriate dimension table respectively Customer, Supplier, and Date in order to identify the primary keys of dimensions that satisfied the query conditions. The Reduce function grouped dimension keys corresponding to the Italy,

Figure 10. Job pseudo-code of fifth phase

```
Algorithm 13: MC-CUBE - Job of phase 5 - Fonction Map()
Input  : ($d_j.RowKey$, $a_j^i.value$): (Key, value) of an attribute belonging to the dimension
         ($key_{out-R1}$, $val_{out-R1}$): Output of phase 1 ($R_1$)
         ($key_{out-R2}$, $val_{out-R2}$) : Output of phase 2 ($R_2$)
         ($key_{out-R3}$, $val_{out-R3}$) : Output of phase 3 ($R_3$)
         ($key_{out-R4}$, $val_{out-R4}$) : Output of phase 4 ($R_4$)
Output: ($key_{tmp}$, $val_{tmp}$)
/* Adapting results of previous phases to the context of query.   */
foreach $v \in key_{out-R1}$ ou $key_{out-R2}$ ou $key_{out-R3}$ ou $key_{out-R4}$ do
    $key_{tmp} \leftarrow a_j^i.value$ // changed identifier by the values of dimensions cited in the query.
    $val_{tmp} \leftarrow m_q$
end
Emit ($key_{tmp}$, $val_{tmp}$)
```

```
Algorithm 14: MC-CUBE - Job of phase 5 - Fonction Reduce()
Input  : ($key_{tmp}$, $val_{tmp}$): Output of Map function
Output: ($key_R$, $val_R$): (key, value) pairs corresponding to the cube $R$
/* Produce the cube.   */
foreach $v \in key_{tmp}$ do
    $key_R \leftarrow key_{tmp}$
    $val_R \leftarrow val_{tmp}$
end
Emit ($key_R$, $val_R$)
```

France, 1996, and 1997. The result is a set of the keys of FRENCH suppliers, ITALIEN customers, and 1996 and 1997 years that will be used in the join with the fact table Lineorder (Figure 11).

**MRJ-2:** Performs the join between dimensions and the fact table, and allows to identify the key of Lineorder (fact table) that satisfy all the query conditions at once. Thus, in Map function, for each dimension key found in fact table, the mapper associate to the Lineorder key a boolean value “1”, otherwise “0”. For example, for Lineorder key = 6, the Map function produces three pairs (6, 1), (6, 1), and (6, 0) corresponding respectively to the scan of the join attributes Custkey, Suppkey, and Orderdate. It means that when the Lineorder key = 6, it satisfies the predicates of CUSTOMER and SUPPLIER dimensions but does not satisfy the predicate of DATE dimension. The Reduce function uses the “logical AND” to associate the boolean values of to the same key which excludes Lineorder key = 6 from the list of Lineorder key that satisfy all query conditions (1 and 1 and 0 = 0). The result is a Lineorder key (1, 2, 3, 5) that keeps the boolean value equal “1” represents the one which satisfies all join predicates (Figure 12).

**MRJ-3:** In Map function, the value of measure (Revenue) corresponding to the Lineorder key (1, 2, 3, 5) are extracted and will be aggregated in Reduce function according Custkey, Suppkey, Orderdate. The result is the granularity level corresponding to ((Custkey, Suppkey, Orderdate), Revenue); ((2, 1, 01011996), 600), ((3, 2, 01011996), 200), and ((1, 1, 01121997), 300) (Figure 13).

## The Second Phase

This phase performs the aggregations according to each dimension separately which correspond in our case to (Custkey, ALL, ALL), (ALL, Suppkey, ALL) and (ALL, ALL, Orderdate).

**MRJ:** In Map function, each dimension key is extracted with their measure corresponding. For example, (2, 600), (3, 2000), (1, 3000) that correspond to (Custkey, Revenue). The Reduce function aggregates the values of measure (revenue) when the dimension key is the same. For

Figure 11. Building of (key, value) pairs that satisfy the dimension conditions

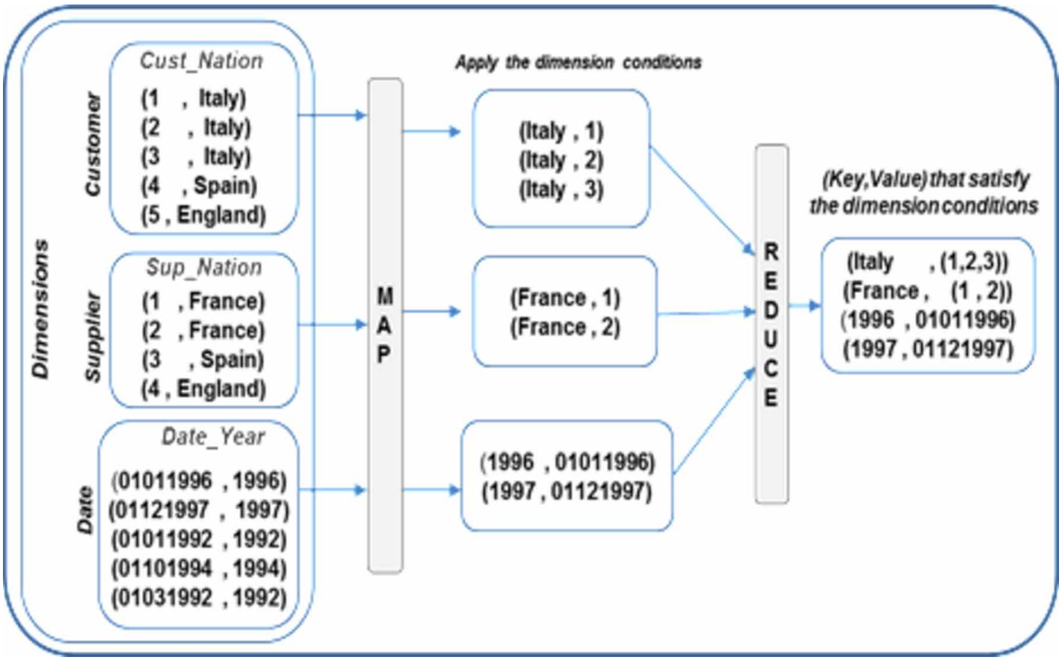


Figure 12. Building a list of fact table key that satisfy all query conditions

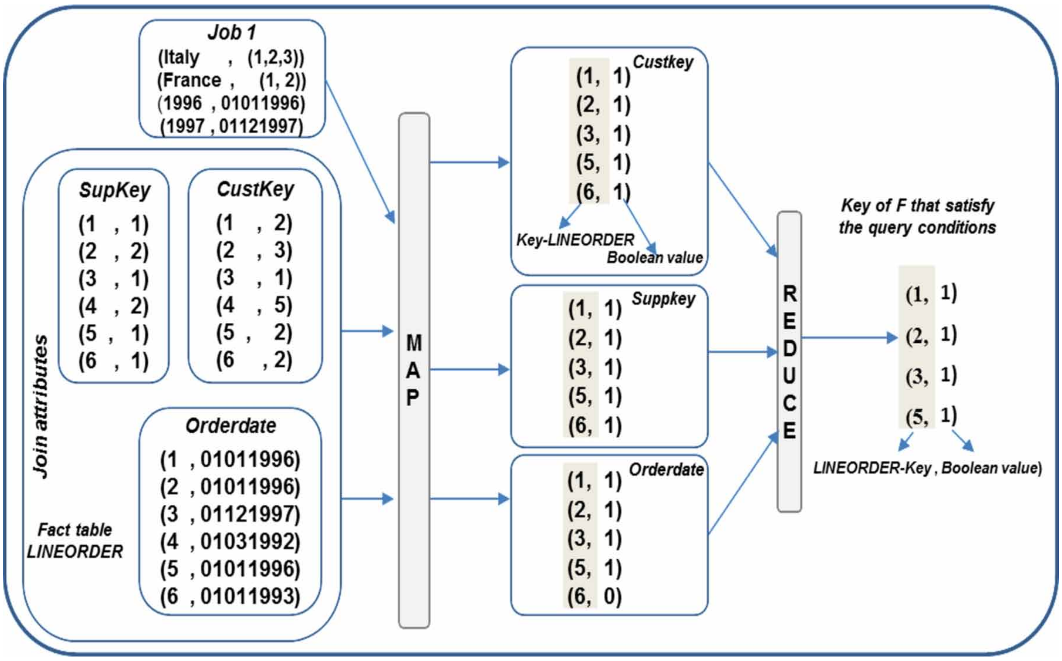
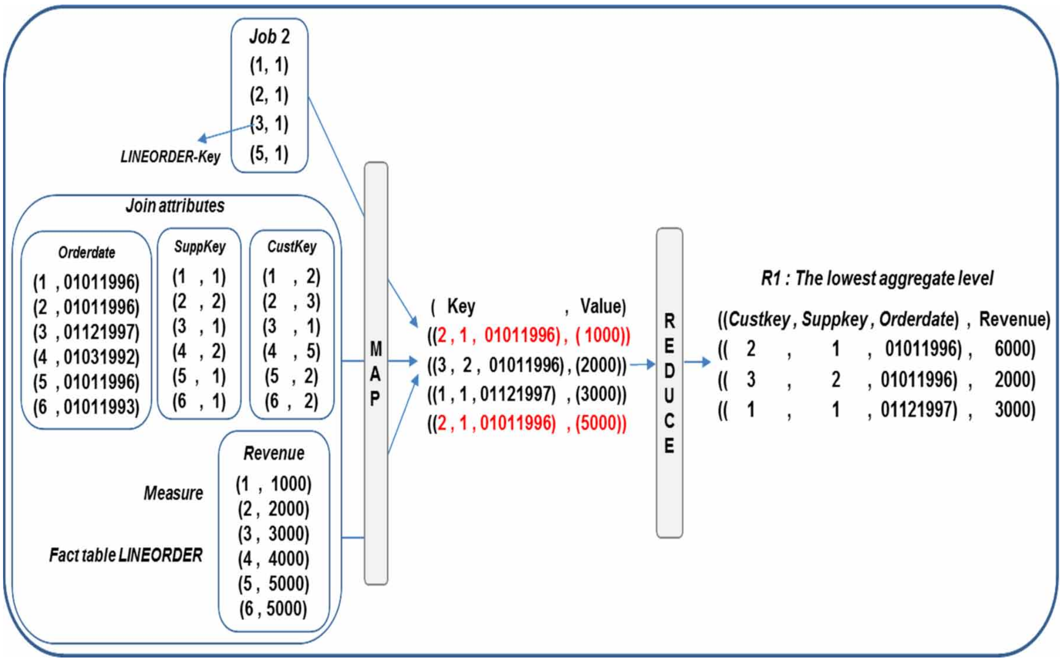


Figure 13. Building the lowest aggregate level of the cube



example, for (Suppkey, Revenue), the (1, 6000), (2, 2000), (1, 3000) will be aggregate to (1, (6000+3000)), (2, 2000) (Figure 14).

## The Third Phase

It performs the aggregations corresponding to (Custkey, Suppkey, ALL), (Custkey, ALL, Orderdate), and (ALL, Suppkey, Orderdate).

**MRJ:** Produces the aggregations according to the different dimensions combinations remaining to compose the cube. The Map function extracts for each dimension keys combinations with their measure corresponding. For example, the ((2, 1), 6000), ((3, 2), 2000), ((1, 1), 3000) that correspond to (Custkey, Suppkey, Revenue). The Reduce function aggregates the values of measure (revenue) when the dimension key combination is the same (Figure 15).

## The Fourth Phase

This phase performs the highest (total) aggregation of the cube corresponding to (ALL, Sum (Revenue)).

**MJR:** In Map function, the value of measure (Revenue) corresponding to the Lineorder key (1, 2, 3, 5) are extracted and aggregated (Revenue: 1000 + 2000 + 3000 + 5000) in Reduce function (Figure 16).

## The Fifth Phase

It gathers the outputs of previous phases and produces the OLAP cube which corresponds to (c.city, s.city, d.year), (c.city, s.city, ALL), (c.city, ALL, d.year), (ALL, s.city, d.year), (c.city, ALL, ALL), (ALL, s.city, ALL), (ALL, ALL, d.year), and (ALL, ALL, ALL).

Figure 14. Building the aggregate level according to each dimension

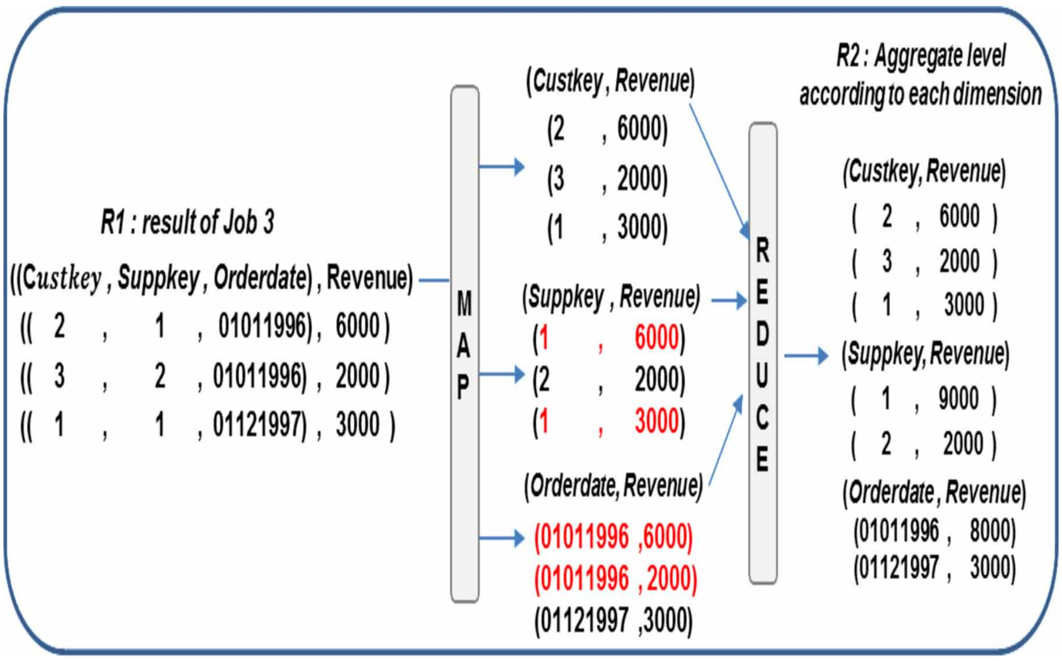


Figure 15. Building the aggregate level according to dimension combinations

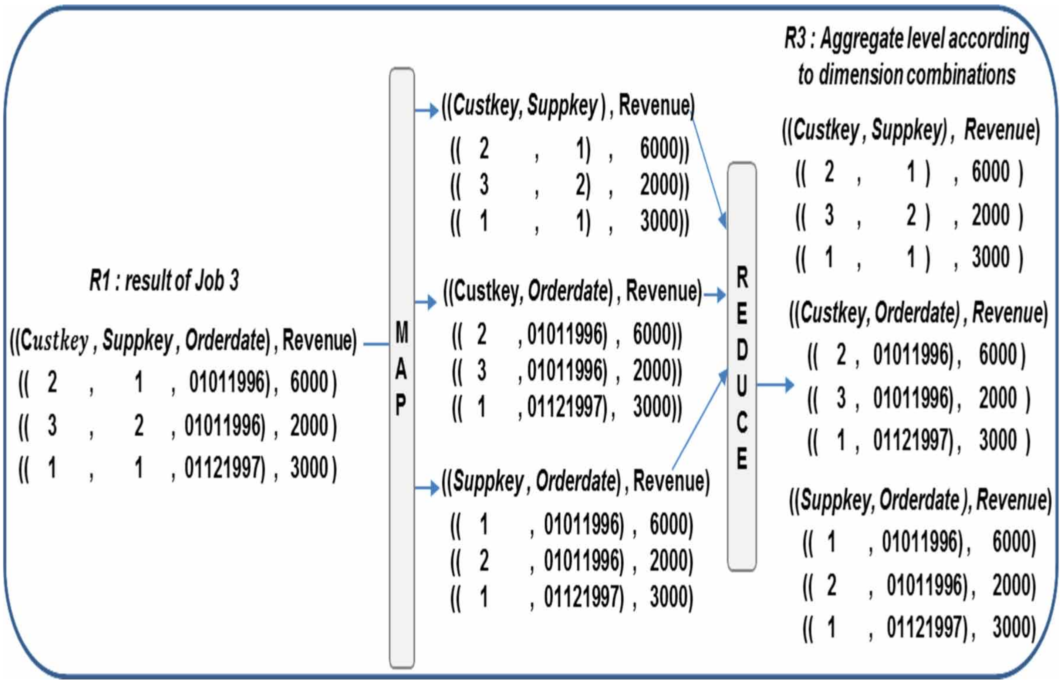
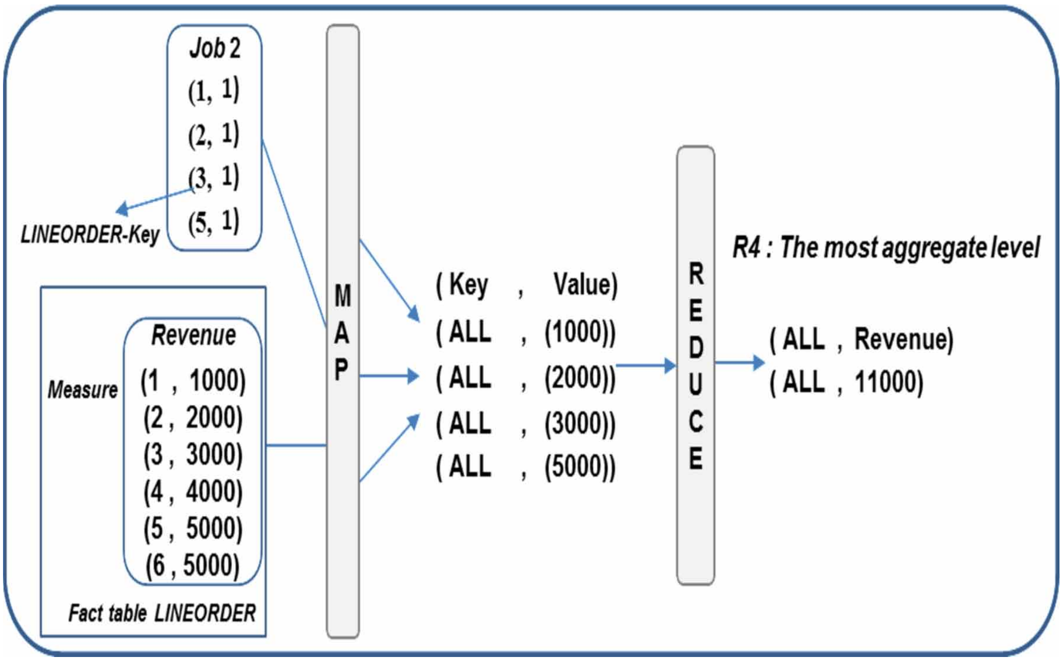


Figure 16. Building the most aggregate level of the cube



**MRJ:** The Map function changes the identifiers of dimensions that are used with the dimensions attributes cited in the query, and gathered it in Reduce function to provide the cube. According to our example, at the Map function, the keys (2, 3, 1) of the Custkey, the keys (1, 2) of the Suppkey, and the keys (01011996, 01121997) are replaced respectively by (Rome, Naples, Milan) of the SUPPLIER dimension, (Lyon, Paris) of the CUSTOMER dimension, and finally (1996,1997) of the DATE dimension. The output is gathered to produce the OLAP cube when the Reduce function is performed (Figure 17).

## IMPLEMENTATION AND TESTING

In order to show the feasibility of our approach and the opportunity to extend the advantage of the columnar approach (beyond the data storage) to build OLAP cubes, we have implemented the MC-CUBE operator in a columnar NoSQL environment using MapReduce, and put in place a non-relational distributed storage and processing environment. This environment is produced by using the Hadoop-2.6.0 and HBase-0.98.8 DBMS (the most popular column-oriented NoSQL system).

**Dataset:** We used the warehouse benchmark SSB described in section 5.2 which is popular for generating data for decision support systems, and we have populated it with data samples by using the data generator (SSB-dbgen) (Sundstrom, 2010). This SSB-dbgen allows to generate data sets with different sizes by specifying the scalability factor (SF: Scale Factor). Thus, we have populated the data warehouse by a sample of data comprising 6 x $10^7$ records (SF = 1).

**Queries set:** Query configuration has been decided on criterias of selectivity and dimensionality. The dimensionality concerns the number of dimensions in grouping clauses ($2^D$ = two dimensions, $3^D$ = three dimensions, $4^D$ = four dimensions and $5^D$ = five dimensions), and the selectivity is a degree of filtering of the data when the conditions of a query are applied (low (L), average (A), high (H), very high (VH)). Overall, we have used 8 queries (4 by dimensions and 4 by selectivity).

Figure 17. Gathering the aggregate levels and producing the cube

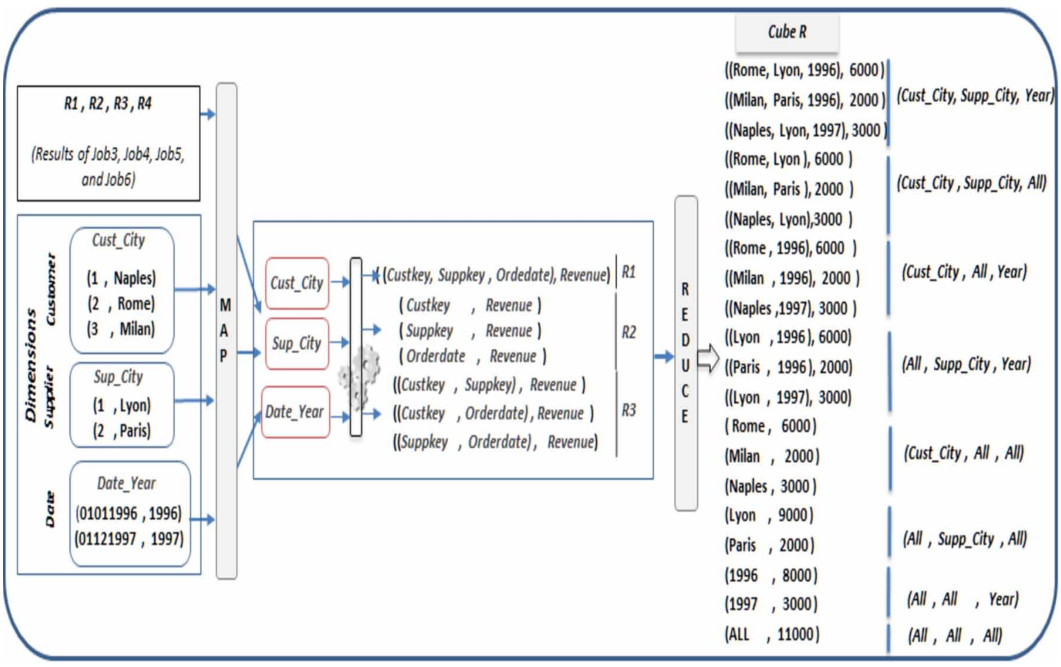


Table 1 shows the queries set by dimensionality and Table 2 shows the queries set by selectivity.

## Experiment 1

In this experiment, we did not use a cluster to avoid its management complexity and eliminate, by now, the number of active nodes in the experimental variables. We have integrated into this environment HIVE[3] and Kylin[4], which manage large data sets. Each one provides an SQL language to query data and allows building OLAP cubes. In Figures 18 and 19, we report the time needed to compute the

Table 1. Queries set by dimensionality

| Query | Number of Dimensions | Dimension: Attribute | Predicate | Measure |
|---|---|---|---|---|
| Query 1 | 2 D | Customer: Nation<br>Supplier: Nation | Date: Year = 1993<br>Lineorder: Discount between1 and 3<br>Lineorder: Quantity < 25 | Sum (revenue) |
| Query 2 | 3 D | Customer: Nation<br>Supplier: Nation<br>Date: Year | | |
| Query 3 | 4 D | Customer: Nation<br>Supplier: Nation<br>Date: Year<br>Date: Month | | |
| Query 4 | 5 D | Customer: Nation<br>Supplier: Nation<br>Date: Year<br>Date: Month<br>Part: Brand | | |

Table 2. Queries set by selectivity

| Query | Selectivity | Dimension: Attribute | Predicate | Measure |
|---|---|---|---|---|
| Query 5 | L: 1,9 * $10^{-2}$ (116.883 rows) | Customer: Nation<br>Supplier: Nation<br>Date: Year | Date: Year = 1993<br>Lineorder: Discount between1 and 3<br>Lineorder: Quantity < 25 | Sum (revenue) |
| Query 6 | A: 2 * $10^{-4}$ (1200 rows) | | Part: Brand1 = 'MFGR#2221'<br>Supplier: Region = 'EUROPE' | |
| Query 7 | H: 9.1 * $10^{-5}$ (549 rows) | | Customer: Region = 'AMERICA'<br>Supplier: Nation = 'UNITED STATES'<br>Date: Year = (1997 or 1998)<br>Part: Category = 'MFGR#14' | |
| Query 8 | VH: 7.6*$10^{-7}$ (5 rows) | | Customer: City=('UNITED KI1' or 'UNITED KI5')<br>Supplier:City= ('UNITED KI1' or 'UNITED KI5')<br>Date: Yearmonth = 'Dec1997' | |

OLAP cubes and we compared between MC-CUBE and the CUBE operators of Hive and Kylin (often used to build OLAP cubes from big data warehouses (see section 2)).

### *Experiment Results*

This experiment allows evaluating OLAP cubes computation times with the MC-CUBE and CUBE operators of Hive and Kylin. Overall, Figure 18, we observe a slight variation in OLAP cube computation times with the MC-CUBE and CUBE operators (Hive and Kylin) when the number of dimensions was increased. However, MC-CUBE shows a better performance than CUBE of Hive and Kylin. Indeed, Hive and Kylin build the cube according to the row-oriented approach which consists to use the aggregates of the lower level of granularity having the fewest attributes when the aggregates of the above level of granularity are performed. By cons, MC-CUBE benefits from the column store, and performs the aggregates of the above levels of granularity, from the one level which is the least aggregated. This allows the MC-CUBE to generate the cube quickly in terms of execution times.

Also, Figure 19 shows that building OLAP cubes with MC-CUBE is better than CUBE operators of Hive and Kylin whatever the selectivity of queries used for performing cubes.

From the results obtained, we find that the MC-CUBE operator allows optimizing the time when the OLAP cubes are performed.

## Experiment 2

In this experiment, we have exposed the MC-CUBE operator to scaling-up which is the raison of advent of NoSQL data models to store and manage massive data. Thus, we have generated 1 TB of data samples and involved building OLAP cubes with the queries set of previous experiment by using

Figure 18. OLAP cubes computation of queries set by dimensionality

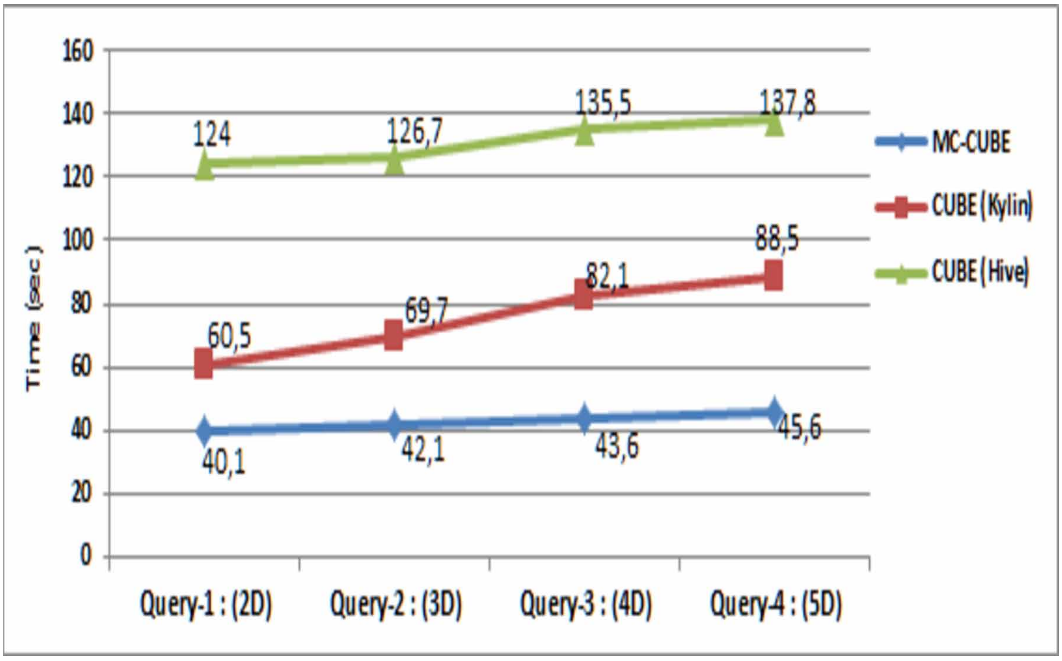
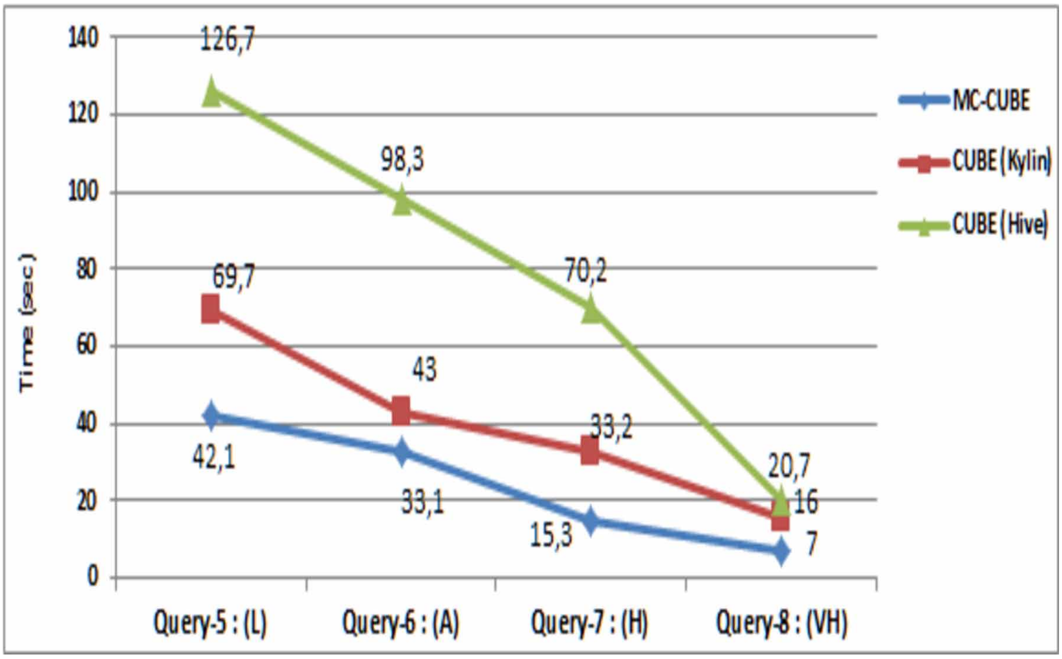


Figure 19. OLAP cubes computation of queries set by selectivity



this time a cluster made up of 15 machines (nodes). Each machine has an intel-Core TMi3-3220 CPU@3.30 GHZ processor with 4GB RAM. These machines operate with the operating system Ubuntu-14.04 and are interconnected by a switched Ethernet 1 GBps in a local area network. One of these machines is configured to perform the role of Namenode in the HDFS system, the master and the Zookeper[5] of HBase. However, the other machines are configured to be HDFS DataNodes and the HBase RegionServers.

Figure 20 reports the time needed to compute the OLAP cubes and we compared between MC-CUBE and the CUBE operators of Hive and Kylin.

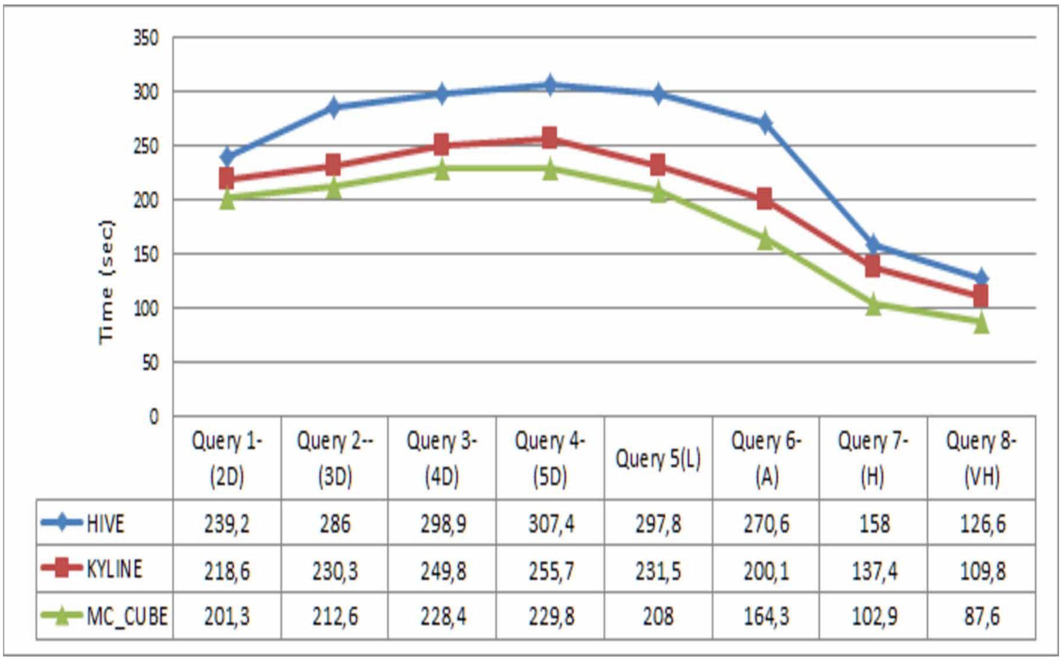
### *Experiment Results*

This experiment allowed to evaluate OLAP cubes building times with the MC-CUBE and compared it to CUBE operators of Hive and Kylin in a distributed environment which is represented by cluster made up to 15 nodes. Overall, we observed that MC-CUBE achieves a better performance than CUBE of Hive and Kylin when the cubes are built.

## CONCLUSION

This paper aims at building OLAP cubes from big data warehouses. Indeed, as big data continues down its path of growth, a major challenge of the decisional information systems has become how to deal with the explosion of data and its analysis when the data warehouses are implemented and the OLAP cubes are built. Consequently, the implementations of data warehouses are oriented towards the new technologies in order to allow more scalability and flexibility for storing and handling data. The solutions that meet with the needs of big data must take into account the distributed storage of data and the parallelized process of data treatment. For such data intensive database management systems, the NoSQL databases infrastructure that is based on “key/value” model is very well adapted to the heavy demands of big data.

**Figure 20. OLAP cube computations faced with variations in dimensions number**



| | Query 1-(2D) | Query 2--(3D) | Query 3-(4D) | Query 4-(5D) | Query 5(L) | Query 6-(A) | Query 7-(H) | Query 8-(VH) |
|---|---|---|---|---|---|---|---|---|
| HIVE | 239,2 | 286 | 298,9 | 307,4 | 297,8 | 270,6 | 158 | 126,6 |
| KYLINE | 218,6 | 230,3 | 249,8 | 255,7 | 231,5 | 200,1 | 137,4 | 109,8 |
| MC_CUBE | 201,3 | 212,6 | 228,4 | 229,8 | 208 | 164,3 | 102,9 | 87,6 |

Fortunately, this work is focalized on the implementation of big data warehouses by using the columnar NoSQL model and proposed the MC-CUBE that is an aggregate operator which allows to generate OLAP cubes according to the column-wise approach. MC-CUBE benefits from column-oriented storage of data when aggregates are performed. In order to take into account the distributed environment for storing data, MC-CUBE uses the MapReduce paradigm to parallelize handling of data and performs the aggregates of different level of granularity that compose the OLAP cube. Contrary to the row-oriented approach where the higher levels of aggregations (2, 3, and 4) are obtained sequentially, the column-wise approach allows to the MC-CUBE operator to obtain the higher levels of aggregations in a parallel way; as a consequence, the time corresponding to the OLAP cube building is considerably reduced.

In the coming extended work, we look ahead to use spark instead of MapReduce in order to give more performance when the cubes are performed. The reason is that Spark uses memory instead of disk to relocate the processed data between the two steps of "Map" and "Reduce".

The results of this work are intended to show the feasibility of our approach and the opportunity to extend the advantage of the columnar approach, beyond the data storage to create analysis contexts (OLAP cubes) when the NoSQL model is chosen to store and analyse the big data. However, we also plan to continue this work by extending further the column-based approach in the study of OLAP manipulation operators and create in a short term, as perspective, some operators such as drill-down & roll-up which allow exploring the cube and navigating in it in accordance with the characteristics of columnar NoSQL model.

Surely, the use of NoSQL technologies for implementing OLAP systems is a promising direction. Hence, we think that building the OLAP cubes from big data warehouses implemented via documents-oriented or graph-oriented models will be an interesting research issue which is worth to be pursued of we want to reach the expected benefits of big data.

## REFERENCES

Abadi, D., Madden, S., & Hachem, N. (2008). Column stores vs. row stores: how different are they really? In *Proceedings of the 2008 Special Interest Group on Management of Data international conference* (pp. 967-980). Vancouver, Canada. Academic Press. doi:10.1145/1376616.1376712

Abello, A., Ferrarons, F., & Romero, O. (2011). Building cubes with MapReduce. *14th international workshop on Data Warehousing and OLAP*. (pp. 17-24). Glasgow, Scotland, UK.

Barber, R., Bendel, P., Czech, M., Draese, O., Ho, F., Hrle, N., & Lee, J. et al. (2012). Business analytics in (a) blink. *A Quarterly Bulletin of the Computer Society of the IEEE Technical Committee on Data Engineering*, *35*(1), 9–14.

Beyer, K., & Ramakrishnan, R. (1999). Bottom up computation of sparse and iceberg cube. In *Proceedings of the International Conference on Management Of Data,* Philadelphia, PA (pp.359-370). Academic Press. doi:10.1145/304182.304214

Bhogal, J., & Choksi, I. (2015). Handling big data using NoSQL. In *Proceedings of the 29th International Conference on Advanced Information Networking and Applications Workshops* (pp. 393-398). Academic Press.

Cattell, R. (2011). Scalable SQL and NoSQL data stores. *Special Interest Group on Management of Data Record Journal.*, *39*(4), 12–27.

Chang, F., Dean, S., Ghemawat, W. C., Hsieh, D., Wallach, M., Burrows, T., & Gruber, R. et al. (2008). Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems Journal*, *26*(2), 4–26.

Chaudhuri, S., & Dayal, U. (1997). An overview of data warehousing and olap technology. *Special Interest Group on Management of Data Record Journal*, *26*(2), 65–74.

Chavan, V., & Phursule, R. (2014). Survey paper on big data. *International Journal of Computer Science and Information Technologies*, *5*(6), 7932–7939.

Chevalier, R., Malki, M., Kopliku, A., Teste, O., & Tournier, R. (2015). Implementation of multidimensional databases in column-oriented NoSQL systems. In *Proceedings of the Conference on Advances in Databases and Information Systems* (pp. 79-91). Academic Press. doi:10.1007/978-3-319-23135-8_6

Dean, J., & Ghemawat, S. (2004). Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation* (vol. 6, pp. 137-149). Berkeley, CA: USENIX Association.

Dehdouh, K., Bentayeb, F., Boussaid, O., & Kabachi, N. (2014). Columnar NoSQL cube: Aggregation operator for columnar NoSQL data warehouse. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, San Diego, CA (pp. 3828-3833). IEEE. doi:10.1109/SMC.2014.6974527

Dehdouh, K., Bentayeb, F., Boussaid, O., & Kabachi, N. (2015). Using the column oriented NoSQL model for implementing big data warehouses. In *Proceedings of the 21st International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV (pp. 469-475). Academic Press.

Farber, F., Cha, S., Primsch, J., Bornhovd, C., Sigg, S., & Lehner, W. (2012). SAP HANA database: Data management for modern business applications. *Special Interest Group on Management of Data Record Journal*, *40*(4), 45–51.

Gandomi, A., & Haider, M. (2015). Beyond the hype: Big data concepts, methods, and analytics. *International Journal of Information Management*, *35*(2), 137–144. doi:10.1016/j.ijinfomgt.2014.10.007

Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., & Pirahesh, H. et al. (1997). Data cube: A relational aggregation operator generalizing group by, crosstab, and sub-totals. *Data Mining and Knowledge Discovery*, *1*(1), 29–53. doi:10.1023/A:1009726021843

Idreos, S., Groen, F., Nes, N., Manegold, S., Mullender, S., & Kersten, M. (2012). Monetdb: Two decades of research in column-oriented database architectures. *A Quarterly Bulletin of the Computer Society of the IEEE Technical Committee on Data Engineering*, *35*(1), 40–45.

Imho, C., Geiger, J., & Galemmo, N. (2003). *Relational Modeling and Data Warehouse Design*. New-York, USA: John Wiley & Sons.

Inmon, W. (1992). *Building the data warehouse*. New-York, USA: John Wiley & Sons.

Jerzy, D. (2012). Business intelligence and NoSQL databases. *Information Systems Management*, *1*(1), 25–37.

Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., & Doshi, L. (2012). The vertica analytic database: C-store 7 years later. *Proceedings of the Very Large Database Endowment*, *5*(12), 1790–1801.

Larson, A., Hanson, E., & Price, S. (2012). Columnar storage in SQL server. *A Quarterly Bulletin of the Computer Society of the IEEE Technical Committee on Data Engineering*, *35*, 15–20.

Lezak, D., & Eastwood, V. (2009). Data warehouse technology by infobright. In *Proceedings of the Special Interest Group on Management of Data international conference* (pp. 841-846). Academic Press.

Lezak, D., Wrblewski, J., Eastwood, V., & Synak, P. (2008). Brighthouse: an analytic data warehouse for adhoc queries. In *Proceedings of the Very Large Database Endowment* (pp.1337-1345).

O'Neil, P., O'Neil, B., & Chen, X. (2007). *The star schema benchmark (SSB)*. Retrieved from http://www.cs.umb.edu/~poneil/StarSchemaB.PDF

Rabuzin, K., & Modruan, N. (2014). Business intelligence and column-oriented databases. In *Proceedings of the Central European Conference on Information and Intelligent Systems* (pp. 12-16). Academic Press.

Sundstrom, D. (2010). *Star schema benchmark dbgen*. Retrieved from https://github.com/electrum/ssb-dbgen

Zukowski, M., & van de Wiel, M., & Boncz, Peter A. (2012). Vectorwise: A vectorized analytical dbms. In *Proceedings of the 28th International Conference on Data Engineering* (pp. 1349-1350). Academic Press.

## ENDNOTES

1 https://hbase.apache.org/
2 http://hadoop.apache.org/
3 https://hive.apache.org/
4 http://kylin.apache.org/
5 http://hbase.apache.org/0.94/book/zookeeper.html

*Khaled Dehdouh is a university lecturer at the Military Academy of Cherchell, Algeria; ex- researcher at the university of Lyon 2, France. Currently, he is the Head of the Computer Engineering Department, at the military Academy of Cherchell. He received his doctorate Degree in Business Intelligence (BI) information system (2015) from the university of Lyon 2, Lyon, France. His interest in research focuses on the evolution of BI in BIG DATA, analysis of social graphs cubes, community detection &evaluation, designing distributed data warehouses using NoSQL databases &MapReduce paradigm for parallel processing and massive data analysis. He is a member of "Big data & Cloud Computing" team in the Military Academy of Cherchell Laboratory. Also, he collaborated in some research projects with different universities: University of Lyon 2 (ERIC Laboratory), France, The Polytechnic Military School of El Bordj-El-Bahri (Algiers) and Saad Dahleb University, Blida 1, Algeria. Concerning his activity in the educational field, he delivers lectures and teaches different modules; mainly, "Decision Support Systems" for Master 2 and "Object-oriented Programming" for bachelor's classes.*

*Omar Boussaid is a full professor in computer science at the Institute of Communication of the University of Lyon2, France. His main work is on Business Intelligence (BI) field, and specifically complex data warehousing and mining. His current research focuses on the evolution of BI in the Big Data. Semantic analysis (Semantic OLAP), multidimensional modeling of textual data, analysis of social graphs cubes, community detection and evaluation, designing distributed data warehouses using NoSQL databases and MapReduce paradigm for parallel processing, massive data analytics, are examples of which are currently based his scientific animation work and scientific supervision. In addition, he is the head of the Master of Business Intelligence & Big data.*

*Fadila Bentayeb received a qualification for supervising research (HDR - Habilitation à Diriger des Recherches in French) from the University of Lyon 2, France in November 24, 2011. Before, she received a Ph.D. in computer science from the University of Orléans, France in 1998. She joined the University of Lyon 2, France in 1999 as a temporary assistant professor and became associate professor in 2001. She is head of Complex Data Warehousing and OLAP research team and board member of the ERIC lab. Her current research interests regard (1) NoSQL data warehouses, (2) Text Warehouses and (3) Personalization & Recommendation. She is also head of the first year of master's degree of computer science (Master 1 Informatique) at ICOM Institute (ICOM: Institut de la Communication in french) of the university of Lyon 2, France since 2014.*