Applied Intelligence (2019) 49:420–434
https://doi.org/10.1007/s10489-018-1275-2

OLAP cube partitioning based on association rules method

Khadija Letrache[1] · Omar El Beggar[1] · Mohammed Ramdani[1]

Published online: 31 August 2018

Abstract
Partitioning is an optimization method used in Business intelligence (BI) systems to improve query and processing performances. That is why most of BI vendors integrate partitioning functionality in their solutions. However, they do not provide partitioning strategies which remain a serious challenge for BI administrators. Some works in the literature have proposed algorithms and strategies for Data warehouse partitioning. Nevertheless, most of them focused on the relational data warehouse partitioning and ignore the OLAP cubes although they are the first concerned by the user multidimensional queries. To deal with this, we propose in this paper a dynamic partitioning strategy for OLAP cubes based on the association rules algorithm. The first step in the proposal consists on analyzing the user queries for a specific period with a view to finding the frequent predicates itemsets. Afterwards, we use our proposed algorithm based on the association rules method to partition the data cube according to the frequent predicates itemsets obtained in the first step. Finally, we present a case study and experiences results to evaluate and validate our approach.

Keywords Data warehouse · Partition · OLAP cube · Association rules · Cube maintenance · Cube performance

1 Introduction

One of the most important characteristics of OLAP systems is to facilitate the analysis of huge amounts of data. However, querying and processing time can become day after day too significant. Thus, some works in the literature have carried out the performance issue in data warehouse (DW) systems using partitioning solutions. Nevertheless, all of them have been focused on the relational data warehouse and ignored the OLAP layer.

Actually, partitioning is to break up data into small, manageable physical units [1]. It can be vertical or horizontal. The vertical partitioning consists on dividing a table into multiple tables based on columns, while horizontal partitioning aims to divide a table by rows into multiple tables having the same columns [2]. In this paper we focus on horizontal partitioning. In fact, using horizontal partitioning on OLAP databases can insure a great improvement of querying and processing performance [3] without changing the cube structure. Actually, querying is the interrogation of the cube while processing is the refresh process of this latter.

Besides reducing the number of rows that the system has to scan for each user query, partitioning reduces the amount of aggregations that the OLAP system recalculates on each cube refresh. The partitioning also allows parallel querying and processing, in addition to facilitating store management [4].

In fact, besides the query response time that getting slow, the cube processing time can take many hours depending on the cube size and the number of fact tables. This can represent a real issue. For instance, when the users claim daily reports (having a flash report each hour for example) this requires to quickly synchronize the cube many times per day. Another case is related to cube refresh breakdowns, to which the BI administrators can be faced.

✉ Khadija Letrache
khadijaletrache@gmail.com

Omar El Beggar
elbeggar_omar@yahoo.fr

Mohammed Ramdani
ramdani@fstm.ac.ma

[1] Informatics Department, LIM Laboratory, Faculty of Sciences and Techniques of Mohammedia, University Hassan II, B.P. 146 Yasmina str. 20658 Mohammedia, Casablanca, Morocco

Actually, the DSS system can be impacted by many external issues, like power failures or ETL breakdowns, in this case, the BI administrator should be able to immediately recover the most urgent partitions and recover the rest afterward. To deal with all of this, the partitioning can be the solution.

Even in OLAP meta-modeling standards, the partitioning is considered. In the CWM [5], an OMG BI standard, the class Region corresponds to a cube partition. Besides, in the OIM [6], an MDC standard, the partition class is the physical container of measure aggregations.

Nowadays most of OLAP vendors like Microsoft, Oracle, Cognos, Mondrian and SAS provide wizards to create and configure cube partitions. The most challenging issue however is to decide the efficient strategy of partitioning which involve specifying, primarily, the set of criteria that characterize each partition as well as defining partitions setting like the physical placement, tolerated size, mode of storage, refresh plan etc.

In this paper we propose a partitioning strategy for OLAP cube based on the association rules algorithm (Fig. 1). The first step in our approach is to analyze the user queries from the log files. The analysis is done using the apriori algorithm [7] which allows identifying the most frequent predicates itemsets. Before performing this analysis, a preprocessing step on queries predicates is required. Afterwards, the partitioning algorithm uses the frequent predicates itemsets as input to deduce the best partitioning criteria. Noting that the partitioning proposal could be periodically performed when noticing a decrease in cube performances.

The remainder of this paper is organized as follows: Section 2 presents previous works related to partitioning approaches especially in data warehouse systems. Section 3 gives a background overview and addresses our partitioning solution. Next, Section 4 describes the implementation of our approach and a case study to evaluate it. Finally, Section 5 presents the conclusion and perspectives.

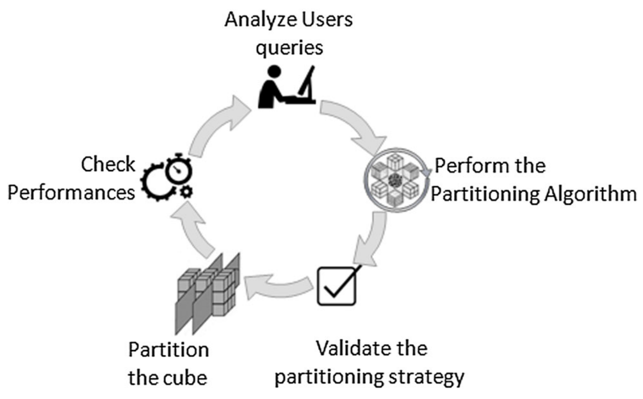

Analyze Users queries
Perform the Partitioning Algorithm
Validate the partitioning strategy
Partition the cube
Check Performances

Fig. 1 Our approach for OLAP partitioning

2 Related work

Data partitioning was discussed by many works in the literature; in our knowledge, most of them deal only with relational data warehouse. In [1] and [4], Inmon and Kimball respectively, both addressed the importance of data warehouse partitioning especially according to the time dimension, For example, if the user requires a near real time report, we define a small partition that contains only data of the current day.

In [8, 9] the authors Bellatreche et al. propose a genetic algorithm for horizontal partitioning of relational data warehouse and a cost model to measure the query performance. The proposal is to partition the fact table according to the dimension partitioning schema and then generating multiple star schemas. The selection of dimension criterion is based on previous knowledge of user frequently asked queries. It however ignores the correlation between predicates. In [10] the authors Hamdi et al. provide a two-level data partitioning approach for real-time data warehouse. The first level consists on partitioning the relational data warehouse according to query workload by using the predicates usage matrix. The second level aims to reorganizing the partitions by merging or splitting those partitions when new data is imported. Another approach for near real time OLAP is proposed by Baluch et al. [11], based also on partitioning in addition to multi-core processing. The solution uses a hot partition to absorb incoming updates and multi-core processing to speed the process of partition building, merge and querying. In [12], the authors Lima et al. propose an adaptive and dynamic virtual partitioning solution based on clustered databases. Similarly, Sun et al. [13] proposed a partitioning framework based on four steps. The first three concern the query workload analysis. It aims to construct predicates vectors that the partitioning algorithm uses in the last step. This latter performs a clustering algorithm to partition data. The approach aims to minimize the number of partitions scanned by queries by storing the partitioning schema in the system catalog. Another partitioning solution is given by Toumi et al. [14] based on multiple techniques. The proposal starts by selecting the query predicates and then using the Jaccard index to calculate the attraction between predicates, afterwards, clustering the predicates by using the Ward algorithm. Finally, the proposal uses a discrete particle swarm optimization to select the best partitioning schema. In [15] the proposed partitioning

Table 1 Related works summary

Paper	Partitioning paradigm	Partitioning based rule	Area	Scope	Partitioning type
L. Bellatreche et al. [8, 9]	Genetic algorithm	Frequent user queries predicate	Relational Data warehouse	Fact and Dimensions	Horizontal
Lima et al. [12]	Physical and virtual partitioning with partial replication	User queries	Relational Data warehouse	Fact and Dimensions	Horizontal
I. Hamdi et al. [10]	Two-level data partitioning: 1-partitioning according to query workload 2- adjustment of the existing partitions after new data loading	User queries and Data amount	Real-time relational data warehouse	Fact and Dimensions	Horizontal
L. Toumi et al. [14]	Jaccard index, data mining and particle swarm optimization	Predicate clusters	Relational Data warehouse	Fact and Dimensions	Horizontal
L. Sun et al. [13]	Partitioning data according to predicates partitioning at data loading time	Predicates vectors	Relational Data warehouse	Fact and Dimensions	Horizontal
M. Grund et al. [15]	Active and Passive Data	Active and Passive Data	Enterprise applications	Production tables	Vertical and Horizontal
R. Bouchakri et al. [16, 17]	Genetic algorithm and partitions reorganization when query workload evolves	User queries	Relational Data warehouse	Fact and Dimensions	Horizontal
Arres et al. [20]	Data warehouse partitioning and placement policy	Frequent user queries	Hadoop-based data warehouses	Fact and Dimensions	Vertical and Horizontal
Baluch et al. [11]	Partitioning and multi-core processing	Partition size	Soft real time OLAP	Fact and Dimensions	Horizontal

Table 1 (continued)

Paper	Partitioning paradigm	Partitioning based rule	Area	Scope	Partitioning type
Rodriguez et al. [18]	Association rules algorithm	User queries attributes	Relational Data warehouse	Fact	Vertical
Bouakkaz et al. [19]	FP-Max algorithm	User queries attributes	Relational Data warehouse	Fact	Vertical
Kim et al. [21]	Column partitioning strategy	User queries and storage constraint	Relational Data warehouse	Fact and Dimensions	Vertical

solution given by Grund et al. concerns the enterprise production databases. The proposal is according to the nature of data. In fact, identifying active and passive data allows performing horizontal and vertical partitioning to reduce the amount of data by ignoring the unused ones. The proposal corresponds, actually, to a kind of archiving strategy. Moreover, in [16, 17] the authors R. Bouchakri et al.. deal with the static partitioning of relational data warehouse when query workload evolves. The solution, based on the genetic partitioning solution [9], consists on performing new partitioning schema by merging or splitting partitions after new query execution. More recently, Arres et al. [20] proposed a data partitioning and placement for Hadoop-based data warehouses. The proposal is to partition the data warehouse tables vertically and horizontally and then place frequent dimension tables predicate in the same cluster or closest based on the frequent user queries. In addition to the k-means algorithm, other data mining algorithms have been used in the literature for DW partitioning. In [18], a vertical partitioning approach based on the association rules algorithm is given by Rodriguez et al.. It consists on using an attribute usage matrix as input of the association rules algorithm. This matrix is also used to determine the min_sup threshold which have to satisfy 71% to 75% of cases. Besides, in [19] the authors Bouakkaz et al. proposed a vertical partitioning approach based on the FP-Max algorithm inspired from the apriori algorithm. The approach starts by identifying the frequent attributes itemsets using the apririori algorithm, and then selecting the partitioning solution from all possible schemes using a block estimate function. Moreover, the authors Kim et al. [21] give a column partitioning strategy based on query workload. The proposal starts by identifying columns that appear together in user queries, then, a selection of the best partitioning schema is performed based on the storage constraint. Finally, it is worth noting that conversely to the listed approaches, the data partitioning have been widely used in the literature to improve data mining algorithms performances by means of parallel or distributed mining approaches, especially the association rules algorithm [22–24]. Table 1 resumes the listed works.

In summary, most of previous works in the literature concern the relational DW. Besides, some works require special hardware or configuration, or use several techniques. Meanwhile, some works that involve partitions reorganization by merging or splitting partitions on each data loading or new query execution cannot be adapted for OLAP cube because the process is quite time consuming. Finally, most of existing approaches are either workload or data based.To deal with this, we propose in this paper a practical and optimized partitioning solution for OLAP cube, using an association rules method which takes into account the user queries, the amount of data cube and storage constraint in addition to the criticality of queries.

3 Our approach and background overview

3.1 Background overview

Data warehouse partitioning: refers to the breakup of data into separate physical units that can be handled independently [1].

In OLAP model, a partition is a subset of a cube used for performance or storage reasons. A partition contains all of the measures and dimensions used by the partition. A horizontal partition contains all of the measures and dimensions of its cube. A vertical partition contains a subset of the measures and dimensions of its cube [6].

Therefore, considering a cube C defined by a set of dimensions D and a set of measures M, we can define a cube partition P by: $P \begin{cases} D_p \subseteq D \\ M_p \subseteq M \\ Y_p \end{cases}$ where D_p is the set of dimensions of P,M_p is the set of measures used by P and Y_p the set of dimension members used to drive P.

We note also P(Y_p) where the partition has the same dimensions and measures as the cube.

Since a data warehouse is no more than a denormalized relational database, partitioning a DW means dividing tables (facts and dimensions) into smaller tables. However, in BI architectures using an analytical layer, by means of OLAP tool, the data is organized differently. Actually, there exists multiple kind of data storage techniques, the two most popular ones are namely the MOLAP (multidimensional OLAP) and ROLAP (Relational OLAP) modes [25]. In the former, the data and pre-calculated aggregations are stored in arrays in the OLAP server, the physical provider of multidimensional metadata, while in ROLAP mode, the data remains in the relational database (DW) and the server creates additional tables in the relational database to store the aggregations [26]. Hence, in such architecture, the user queries access data (arrays or tables) from the OLAP server through OLAP tools. The partitioning process must thus concern, in this case, the OLAP cube, which will result on partitioning the aggregation tables or arrays.

Besides, in MOLAP cube the refreshment of cube's data must be done regularly [25], which we call the processing operation. This latter depends on the volume of data. Threfore, partitioning the cube also allows optimizing the processing time in OLAP systems. Indeed, when a cube is partitioned, in addition to parallel processing, we can process only the needed partitions instead of processing the whole cube [27] like processing only the partition corresponding to the current day for daily reports [4].

To conclude, the partitioning allows optimizing the query and processing time, but it must be handled on the OLAP layer. In case of using MOLAP mode, the partitioning results on subdividing the cube arrays while subdi viding the aggregation tables in case of the ROLAP one.

MultiDimensional eXpressions (MDX): is a powerful syntax that enables querying multidimensional objects and provides commands that retrieve and manipulate multidimensional data from those objects. The MDX provides functionality for creating and querying multidimensional structures, its syntax is similar to the Structured Query Language (SQL), however, its features can be more complex and robust than SQL's features [28].

The MDX manipulates multidimensional data. We call Tuple a slice of data from a cube [28]. The tuple is formed by a combination of dimension members, as long as there are no two or more members that belong to the same hierarchy [28], it can also be viewed as a cross-section or vector of member data in a cube. Considering a cube C and its D dimensions $D = \{D_1, D_2.., D_d\}$ and MB_i the set of members of D_i levels, we define a tuple as:

$T = mb_1 \otimes mb_2.. \otimes mb_n$ where $mb_i \in MB_i$

Example: T=([Date].[Year].[2015],[Customer].[Nation].[France]) is a tuple composed by the members [Date].[Year].[2015] and [Customer].[Nation].[France].

We call Set a set of tuples as $S = \{T_1, T_2.., T_n\}$ where T_i is a tuple.

Example:

$$S = \{([Date].[Year].[2015], [Product].[Name].[Coat]), ([Date].[Year].[2016], [Product].[Name].[Jeans])\}$$

Association rule: is an implication of the form $X \rightarrow Y$ which means that all the transactions in the database that contain X tend to contain Y. the formal statement of an association rule is:

Let $I = \{i_1, i_2, .., i_n\}$ be a set of items, a set of items $X \subset I$ is called itemset. Let T be a set of transaction where each transaction t in T is an itemset such that $t \subseteq I$.

We call an association rule $X \longrightarrow Y$ where $X \subset I$ and $Y \subset I$ and $X \bigcap Y = \emptyset$ with the confidence factor (min_conf) $0 \leq c \leq 1$ if at least c% of transactions in T that satisfy X also satisfy Y, and with a support s which means that s% of transactions in T contains $X \bigcup Y$ [29]. It corresponds to a statistical significance that allows considering only rules with the frequency above some minimum threshold (min_sup), by means of frequent itemsets [7]. Otherwise a rule is not worth consideration or simply less preferred [29].

The frequency of an itemset is simply the count of the itemset [7]. The support can be defined by:

$$Support(X \rightarrow Y) = \frac{count(X \bigcup Y)}{|T|} \text{ where } |T| \text{ is the cardinality of T}$$

We note $Support(X \rightarrow Y)$ by sup(X,Y)

And the confidence is defined by [7]:

$$Confidence(X \rightarrow Y) = \frac{count(X \bigcup Y)}{Count(X)}$$

We note $Confidence(X \rightarrow Y)$ by conf(X,Y)

We call large itemsets, all combinations that have fractional transaction support above the threshold min_sup [29].

3.2 Our approach

As discussed above, the majority of previous partitioning approaches concern the relational data warehouse. However,

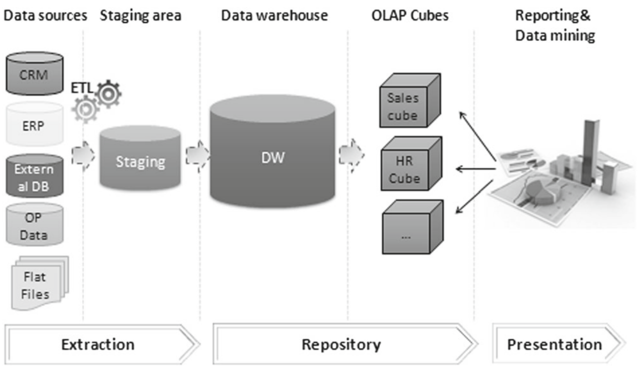


Fig. 2 Standard BI Architecture

when the decisional system uses OLAP cubes to interrogate the DW and to response the user queries, the DW partitioning cannot be the solution. In fact, as discussed above, in such architecture (Fig. 2) the data and aggregations are either extracted from the DW and stored in the OLAP server [31] in case of using the MOLAP mode of storage, or stored in relational aggregation tables when using the ROLAP one. The partitioning strategy must thereby concern the OLAP layer. In this respect, our approach aims to partition OLAP cubes in order to enhance their query and processing time. Moreover, even if the partitioning is supported by the most of OLAP vendors, defining an efficient partitioning strategy cannot be done by tools. In fact, partitioning an OLAP cube has to take into account the nature of data as well as the user requirements. For this reason, our data cube partitioning proposal is according to the most frequent predicates itemsets of the user queries. By applying the association rules algorithm to the logged user queries, we can deduce the most used predicates and also the correlation between them. The partitioning algorithm takes, thus, the large itemsets deduced from the previous step to partition the OLAP cube. The algorithm rolls up the large itemsets sorted by frequency (support) to partition the cube until attaining the minimum support defined before. At last, the OLAP cube is partitioned according to the user queries frequent predicates. However, non-frequently asked data can remain untouchable. This data corresponds to rarely used or inactive data. In this case, we propose to partition this latter mathematically to satisfy to the storage constraint if exists.

3.2.1 User queries analysis

Predicates of a user's queries can be similar to a shopping basket. Many correlations between predicates can exist, and that cannot be deduced from the data but from the user queries themselves. Using an association rules algorithm to analyze them can provide very interesting results.

Hence, the first step in our proposal is to gather the user queries from the OLAP system logs. These latter can have different format (text file, csv, relational table etc) depending on the used framework, but it always requires word processing. There exist also tools, like the SQL profiler and Jmeter, that allows tracing the user queries and gathering them in a simple format in addition to providing several useful information like the query execution time, its duration, user, cube, etc.

Hence, once the user queries are collected, our approach uses the apriori algorithm to deduce the frequent predicate itemsets. The resulting itemsets will be used as input of the partitioning algorithm in the next step.

Before applying the apriori algorithm, we have to preprocess the user queries by firstly replacing the date predicates by dynamic formula. In fact, if an itemset contains a date predicate, it will never be considered as a frequent itemset, because the date item is not a fixed value. Nonetheless, if we compare the date predicate with the query execution date we will deduce a correlation between the queries.

In Table 2 the predicate [Date].[Year-Month-Date].[Year].[2016] corresponds to the previous year compared to the query execution date. We replace the explicit value 2016 by the MDX formula: ParallelPeriod([Date].[Year-Month-Date].[Year],1, StrToMember (”[Date].[Year-Month-Date].[Year].[”+Year(Now())”+]”)) which returns to the previous year of the current date. Hence the predicate year-1 will be common to all itemsets containing this latter. Noting that the function StrToMember converts a string to an MDX member while the ParallelPeriod returns the parallel period of the current member according to the parameter hierarchy level (year, month, day, etc.).

Secondly, the logical operators are also preprocessed. In fact, in a multidimensional model, the general syntax of a user query Q_i on a cube C is ⟨ Select S_i from C where Z_i⟩ where S_i is the list of selected dimension levels (projection), C is the source cube and Z_i is the set of predicates (selection), we note thus Q_i by Q_i $\langle S_i|C|Z_i\rangle$. Z_i can be a tuple or a set of tuples, which refers to an ”OR” logical operator. We can note Z_i by:

$$Z_i = \bigcup_{j=1}^{n} T_j \text{ where } T_j \text{ is a tuple of } Z_i$$

Every tuple T_j in Z_i represents thus a predicate itemset. Before using the query Q_i we separate it onto multiple queries Q_j each one corresponding to a tuple T_j. Q_j is thus defined by $\langle S_i|C|T_j\rangle$.

For example, if the condition in a user query is:

$$Z = \{([Date].[Year].[2015], [Product].[Name].[Coat]), ([Date].[Year].[2016], [Product]. [Name].[Jeans])\}$$

Table 2 Examples of date predicates pre-processing

Query date predicate	Query execution date	Used predicate
[TIME].[DATE-MONTH-YEAR].[YEAR].&[2016]	[TIME].[DATE-MONTH-YEAR].[YEAR]&[2017]	ParallelPeriod([TIME].[DATE-MONTH-YEAR].[YEAR],1, STRTOMEMBER('[TIME].[DATE-MONTH-YEAR].[YEAR].&['+cstr(Year(Now()))+']'))
[TIME].[DATE-MONTH-YEAR].[MONTH].&[20171]	[TIME].[DATE-MONTH-YEAR].[MONTH].&[20171]	STRTOMEMBER('[TIME].[DATE-MONTH-YEAR].[MONTH].&['+cstr(Year(Now()))+cstr(Month(Now()))+']')
[TIME].[DATE-MONTH-YEAR].[DATE].&[20/01/2017]	[TIME].[DATE-MONTH-YEAR].[DATE].&[20/01/2017]	STRTOMEMBER('[TIME].[DATE-MONTH-YEAR].[DATE].&['+cstr(Now())+']'))

We divide Z into two itemsets (tuples) before using it like:

$Z_1 = ([Date].[Year].[2015], [Product].[Name].[Coat])$ and $Z_2 = ([Date].[Year].[2016], [Product].[Name].[Jeans])$

The last preprocessing task concerns the queries criticality. In fact, there may exist queries that are not frequent but critical and thus require good performance in terms of response time and availability, like those used by the top

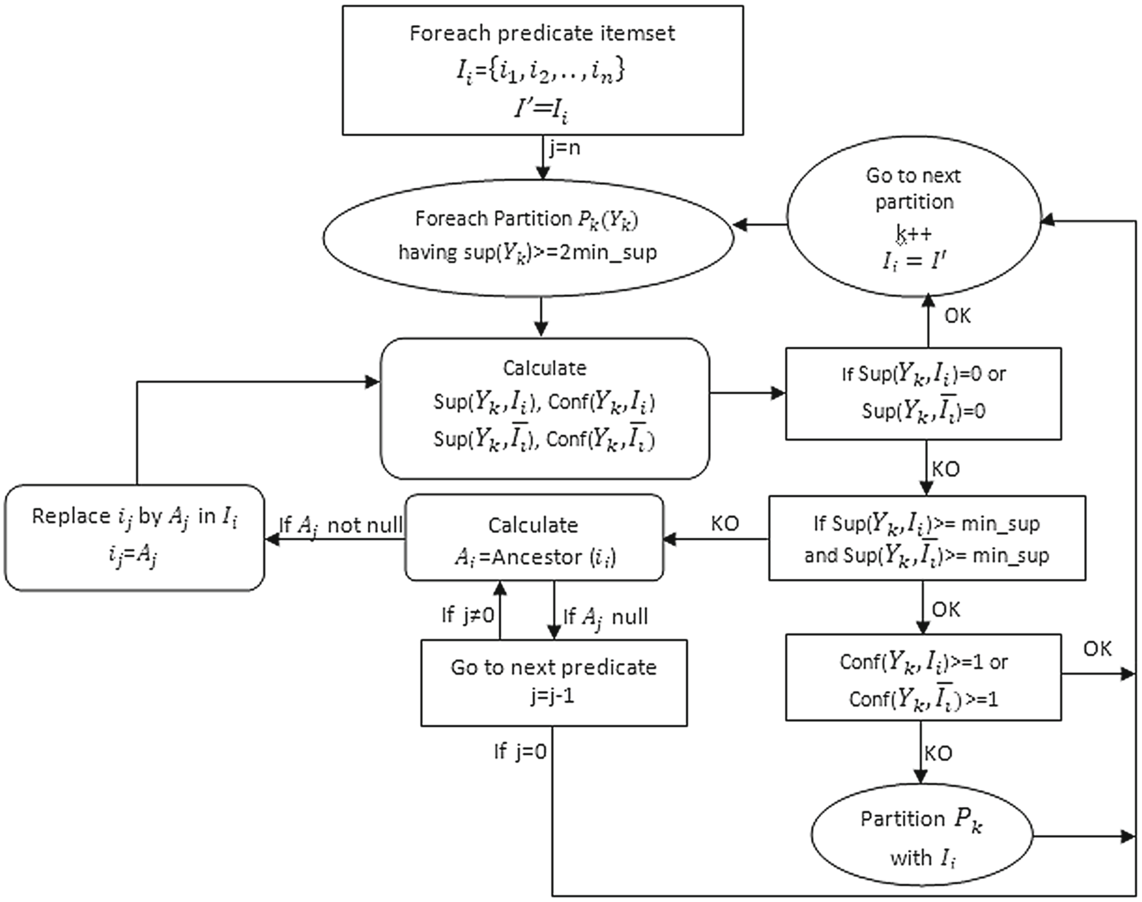

Foreach predicate itemset
$I_i=\{i_1, i_2, .., i_n\}$
$I'=I_i$
j=n
Foreach Partition $P_k(Y_k)$ having sup(Y_k)>=2min_sup
Go to next partition
k++
$I_i = I'$
OK
Calculate
Sup(Y_k,I_i), Conf(Y_k,I_i)
Sup(Y_k,$\overline{I_i}$), Conf(Y_k,$\overline{I_i}$)
If Sup(Y_k,I_i)=0 or Sup(Y_k,$\overline{I_i}$)=0
KO
Replace i_j by A_j in I_i
i_j=A_j
If A_j not null
Calculate
A_i=Ancestor (i_i)
KO
If Sup(Y_k,I_i)>= min_sup and Sup(Y_k,$\overline{I_i}$)>= min_sup
If j≠0
If A_j null
OK
Go to next predicate
j=j-1
Conf(Y_k,I_i)>=1 or Conf(Y_k,$\overline{I_i}$)>=1
OK
If j=0
KO
Partition P_k with I_i

Fig. 3 Our proposed partitioning process

management. These queries are neglected when using a workload-based algorithm. To deal with this, we introduce a new parameter that we call criticality factor and which allows increasing the frequency of critical queries. Hence, the total number of user queries T will be: $Count(T) = \sum_{i=1}^{n} \alpha Count(Q_i)$ where n is the number of user queries and α is the criticality factor of Q_i. Noting that the default value of α is 1.

Once the user queries and predicates are preprocessed, we perform the apriori algorithm [7] to calculate the frequency of each predicate itemset to find the frequent itemsets. For that, we use a small min_sup threshold, which is possible because of the small size of user queries, to let the partitioning algorithm later control the stop condition. The algorithm sorts the predicates of each itemset by frequency and also the resulting itemsets.

3.2.2 Partitioning algorithm

Our partitioning algorithm aims to partition OLAP cubes into smaller partitions to enhance query and processing time. To do that, our partitioning algorithm (see Algorithm 1), based on the association rules algorithm, loops first on the large itemsets I={I_1, I_2, .., I_n}, resulting from the queries analysis and sorted by frequency. Afterwards, the algorithm rolls up all partitions P={P_1, P_2, .., P_m}, or the cube in the first iteration, and calculates the support of the new partitions (Fig. 3).

Indeed, using an itemset I_j to split a partition P_i (or the parent cube) implies replacing P_i by two new partitions P_{i_1} and $\overline{P_{i_1}}$ as shown in the example in (Fig. 4). The $\overline{P_{i_1}}$ verifies the condition $\overline{I_j}$ like

$$P_i = P_{i_1} \bigcup \overline{P_{i_1}} \text{ and } P_{i_1} \bigcap \overline{P_{i_1}} = \emptyset$$

Hence, to avoid creating too small partitions, the algorithm calculates the support of P_{i_1} and $\overline{P_{i_1}}$. If the obtained values are superior to the threshold min_sup, thus the algorithm uses the itemset I_j to partition the current partition P_i.

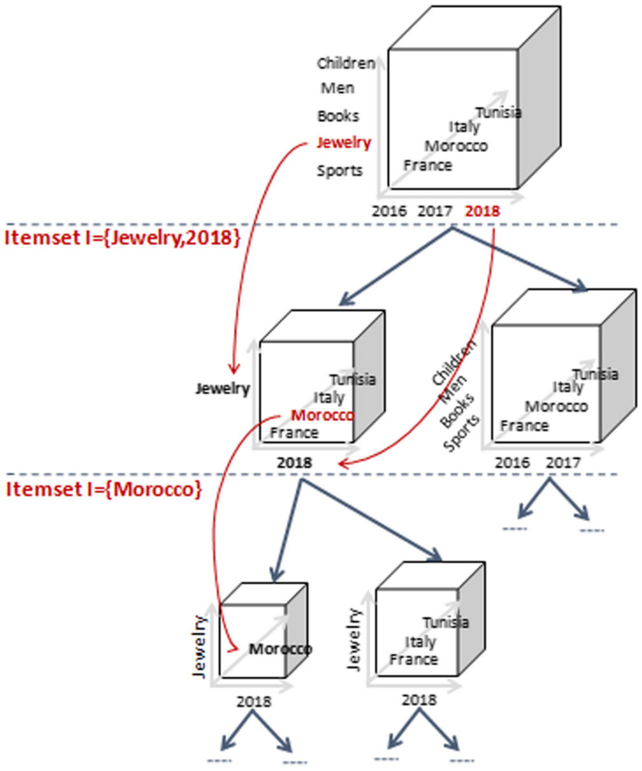


Fig. 4 Example of cube partitioning

For instance, considering an itemset $I_j = \{Current Month, Consignment, Customer Cat A\}$ to be used to divide a partition containing data of Morocco like $P_i(\{Morocco\})$. The two resulting partitions are $P_{i_1}(\{Current Month, Consignment, Customer\ Cat\ A, Morocco\})$ and $\overline{P_{i_1}}(\{Current\ Month, Consignment, Customer\ Cat\ A, \overline{Morocco}\})$. The supports of these two partitions whose formula are respectively

$$\frac{Count(\{CurrentMonth, Consignment, CustomerCatA, Morocco\})}{Count(All)} \text{ and}$$
$$\frac{Count(\{CurrentMonth, Consignment, CustomerCatA, \overline{Morocco}\})}{Count(All)} \text{ must verify the min\_sup threshold.}$$

Otherwise, the algorithm tries to enlarge the itemset scope by replacing the predicates by their ancestors. The algorithm starts thus from the last predicate which means from the predicate having the smaller support in the itemset, then replaces it by its first ancestor and so on, until the satisfaction of the min_sup threshold or until attaining the predicate root, which means that the predicate will be ignored. The algorithm skips then to the next predicate etc.

In the same example listed above, if the supports of P_{i_1} or $\overline{P_{i_1}}$ do not satisfy the min_sup then the algorithm replaces the predicate "Customer Cat A" by its first ancestor, which corresponds in this case to the root "All". This means that the predicate is ignored and I_j becomes

$$I_j = (\{CurrentMonth, Consignment, All\}) = (\{CurrentMonth, Consignment\})$$

If the supports still do not satisfy the min_sup then I_j becomes

$I_j = (\{CurrentMonth, Jewelry\})$ as shown by the (Fig. 5).

It is worth noting that the partitioning algorithm does not use a min_conf threshold because the aim is to find the predicate itemsets that represent the better criteria to partition the cube and constitute a considerable population instead of looking for the strength of the itemsets relationship.

The maximum number of created partitions will be $N = 2^n$ where n is the number of large predicate itemsets.

The support of an itemset I and a partition $P_i(Y_{p_i})$ where $Y_{p_i} = \{y_1, y_2, .., y_m\}$ is the set of predicates used to drive P_i, is the occurrence of $Y_{p_i} \bigcup I$ in the cube C. To calculate it in the data cube, we use the pre-calculated aggregations of the Count measure [30] like:

$$Sup(Y_{p_i}, I) = \frac{Count(Y_{p_i} \bigcup I)}{|C|}$$

If the supports of the new partitions are like $Sup(Y_{p_i}, I) = 0$ or $Sup(Y_{p_i}, \overline{I}) = 0$ then I is not interesting in $P(Y_{p_i})$. The algorithm skips then to the next partition.

Besides, if the support of P_i is such as:

$$Sup(P_i) = Sup(Y_{p_i}) = \frac{Count(Y_{p_i})}{|C|} < 2min\_sup$$

the algorithm skips, in this case also,to the next partition. Indeed, in that case, P_i cannot be divided yet on two new partitions that both verify the min_sup threshold, because this implies that:

$Sup(Y_{p_i}, I) >= min\_sup$ and $Sup(Y_{p_i}, \overline{I}) >= min\_sup$

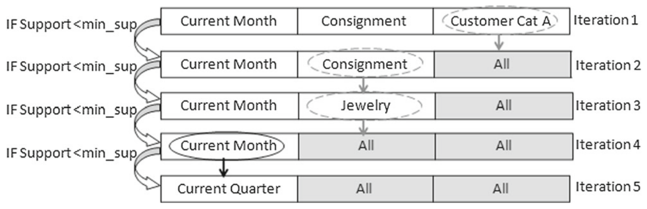


Fig. 5 Illustration of the scope enlargement mechanism

and thus, to be partitioned, P_i must verify $Sup(Y_{p_i}) >= 2min\_sup$

On the other hand, the confidence of I calculates its occurrence in the partition P_i:

$$Conf(Y_{p_i}, I) = \frac{Count(Y_{p_i} \bigcup I)}{Count(Y_{p_i})}$$

If the confidence of I is equal to 1 (or 100%) thus $Y \subseteq I$, which means that I (or its descendants) is already used to obtain P_i, I is then ignored.

Furthermore, as already discussed, the partitioning can be used to help in storage management, that is why our partitioning algorithm allows defining a tolerated size (max_size) for the new partitions and which can be used to estimate the appropriate value of the min_sup threshold.

In fact, we can define the size of a cube partition P_i by [32]:

$$Size(P_i) = \frac{|P_i|}{|C|} Size(C) \tag{1}$$

and $Size(C) = \sum_{i=1}^{n} Size(P_i)$ where n is the number of partitions of the cube C.

If the partition size must not exceed a max_size threshold then we must have:

$$Size(P_i) \leq max\_size \tag{2}$$

(1) and (2) implies that we need to have the following condition verified:

$$\frac{|P_i|}{|C|} \leq \frac{max\_size}{Size(C)}$$

We need to have then

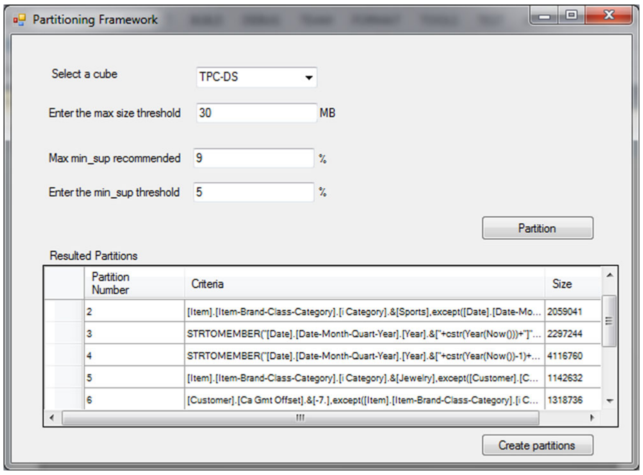
$$Sup(P_i) \leq \frac{max\_size}{Size(C)} \tag{3}$$

On the other hand, to stop partitioning a partition $P_i(Y_{p_i})$, two cases exist. The first one, discussed above, and which is the ideal one, corresponds to the case where:

$$Sup(P_i) < 2min\_sup \tag{4}$$

Hence, by choosing a min_sup like:

$$min\_sup <= \frac{max\_size}{2Size(C)} \tag{5}$$



Select a cube: TPC-DS

Enter the max size threshold: 30 MB

Max min_sup recommended: 9 %

Enter the min_sup threshold: 5 %

Resulted Partitions

Partition Number	Criteria	Size
2	[Item].[Item-Brand-Class-Category].[i Category].&[Sports],except([Date].[Date-Mo...	2059041
3	STRTOMEMBER("[Date].[Date-Month-Quart-Year].[Year].&["+cstr(Year(Now()))+"]"...	2297244
4	STRTOMEMBER("[Date].[Date-Month-Quart-Year].[Year].&["+cstr(Year(Now())-1)+...	4116760
5	[Item].[Item-Brand-Class-Category].[i Category].&[Jewelry],except([Customer].[C...	1142632
6	[Customer].[Ca Gmt Offset].&[-7.],except([Item].[Item-Brand-Class-Category].[i C...	1318736

Fig. 6 Our partitioning framework

and from (4) and (5) we will have:

$$Sup(P_i) < \frac{max\_size}{Size(C)}$$

Thereby, we insure that (3) and then (2) are verified.

The second case is where the algorithm loops on all the itemsets. In this case, even if the condition (5) is set, the condition (2) is not necessarily verified. This means that the user needs are satisfied but not the size constraint, if exists. In that case, a range partitioning according to time dimension can be used as described in Section 3.2.3.

3.2.3 Range partitioning algorithm for inactive data

In data warehouses containing old or inactive data, the latter does not appear in the users queries. Hence, in any workload-based partitioning algorithm, the partitioning will concern only currently used data, while the rarely or unused data are not affected. Using our partitioning algorithm, these data are isolated in a separate partition that we call "Rest". This data should be partitioned, in case of storage constraint, with a minimum maintenance cost. To deal with this, we propose a complementary algorithm to the principal partitioning algorithm. The proposed solution is to partition the "Rest" data mathematically using the time dimension until satisfying the storage size constraint. The algorithm starts by identifying the appropriate time range of partitioning according to data size. Afterwards, the algorithm constructs a first partition using the given range. For example, if the range is the year, the first constructed partition will be P(y) like y=Max(year,Rest)=2016, if 2016 is the maximal year in the "Rest" data. After that, the algorithm increments the partition P, by range intervals, until reaching the tolerated partition size (see Algorithm 2). This algorithm can be also used to further fragment the partitions resulting from the first partitioning phase based on user queries, if they do not satisfy the size constraint.

Algorithm 1 Our partitioning algorithm

```
Require: C cube to partition,
  $min\_sup$ the minimum support threshold
  $Predicates\_Itemsets$ list of frequent predicate itemsets

Ensure: P list of resulting partitions
  $P = \{C\}$  // P contains initially the cube
  $P' = P$
  for all $I_i = \{i_1, i_2, ., i_n\} \in Predicates\_Itemsets$ do
    $I' = I_i$  // Save the initial value of $I_i$
    $j = n$  // n is the number of predicates in $I_i$
    $P = P'$
    for all $p_k(Y_k) \in P = \{p_1, p_2, ., p_m\}$ do
      if $\frac{Count(Y_k)}{|C|} >= 2min\_sup$ then
        while $j \geq 1$ do
          $sup(Y_k, I_i) = \frac{Count(Y_k \bigcup I_i)}{|C|}$  // calculates the frequency of $I_i$ in the Cube C
          $conf(Y_k, I_i) = \frac{Count(Y_k \bigcup I_i)}{Count(Y_k)}$  // calculates the frequency of $I_i$ in the partition $p_k$
          $sup(Y_k, \overline{I_i}) = \frac{Count(Y_k \bigcup \overline{I_i})}{|C|}$  // calculates the frequency of $\overline{I_i}$ in the Cube
          $conf(Y_k, \overline{I_i}) = \frac{Count(Y_k \bigcup \overline{I_i})}{Count(Y_k)}$  // calculates the frequency of $\overline{I_i}$ in the partition $p_k$
          if $sup(Y_k, I_i) = 0$ or $sup(Y_k, \overline{I_i}) = 0$ then
            $j = 0$  // Go to next partition
          Else
            if $sup(Y_k, I_i) \geq min\_sup$ and $sup(Y_k, \overline{I_i}) \geq min\_sup$ then
              if $conf(Y_k, I_i) >= 1$ or $conf(Y_k, \overline{I_i}) >= 1$ then
                j=0  // Go to next partition
              Else
                //Partition $p_k$ using $I_i$ and add the new partitions to $P'$
                $P'.add(p_k.Partition\_with(I_i))$
                $P'.delete(p_k)$  // delete $p_k$ from $P'$
                $j = 0$ // Go to next partition
              end if
            Else  // loop on predicates ancestors
              while $A$ is null and $j \geq 1$ do
                $A = Ancestor(i_j, 1)$
                if A is not null then
                  $I_i.Replace(i_j, A)$  // replace $i_j$ by its ancestor
                Else
                  $j = j - 1$  // Go to next predicate
                end if
              end while
            end if
          end if
        end while
      end if
    end for
    $I_i = I'$
  end for
```

Algorithm 2 Range partitioning algorithm for inactive data

```
Require: Rest partition,
  Partitions: list of resulting partitions
Ensure: Partitions: list of resulting partitions
  while $|Rest| > \frac{max\_size}{Size(C)}|C|$ do
      int i=0
      Range=Range_Definition(Rest)
      Max_Val=Max(Range,Rest)
      z= {Max_Val}    // the max value of range in the Rest data
      y={}
      while $|P(z)| < \frac{max\_size}{Size(C)}|C|$ do
          y.add(z[i])
          i++
          Prev_Period=Previous_Period(Max_Val,
          Range,i,Rest) // Return the i th previous period
                        //according to the Range interval
          z.add(Prev_Period)
      end while
      Partitions.add(P(y))
      Rest.delete(P(y))
  end while
  Partitions.add(Rest)
  ________________________________

  Function Range_Definition(Part partition) {
  R={year,semester,quarter,month}
  Range=R[0]
  y=Max(Range,Part)  //Return the max value of the Range in the partition Part
  int i=0;
  while $|P(y)| > \frac{max\_size}{Size(C)}|C|$ and $i < 3$ do
      i++
      Range=R[i]
      y=Max(Range,Part)
  end while
  Return Range
  }
```

4 Approach implementation and evaluation

4.1 Implementation and experimental study

To implement the algorithms of user analysis and cube partitioning, we used the C# language, which allows manipulating multidimensional objects through the ADOMD.NET library (Fig. 6). In addition, we considered the TPC-DS database, a decision support benchmark [33] to verify our approach. We created the associated OLAP cube using the Microsoft SQL Server Analysis Services. The created cube contains one fact table Store_Sales with 24M records and five regular dimensions as shown in (Fig. 7): Date (73K) containing the hierarchy date-month-quarter-year, Customer (100K) containing the hierarchy customer-city-state-country, Item (18K) with a hierarchy item-brand-class-category, Promotion (300) and Store (12) containing the hierarchy store-city-state-country. The initial size of the created cube is 175,17MB. Finally, we performed our experiments on an i3 processor machine.

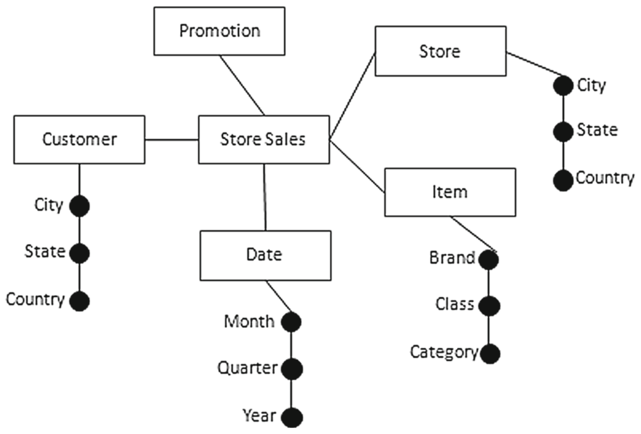


Fig. 7 The multidimensional model of our case study

We generated a sample of 100 queries using the TPC-DS templates, that we converted to MDX language to perform the query workload analysis. After preprocessing the MDX queries as described in Section 3.2.1, by replacing the date predicates by dynamic formula and separating sets of tuples and setting the criticality factor to 1 for all queries, we then performed the apriori algorithm to deduce the most frequent predicate itemsets. We fixed the support to 5%. The apriori algorithm identified 24 frequent itemsets. The results show that the most asked statistics concern the last two years, especially data of the sports, books and jewelry item categories.

Next, we performed our partitioning algorithm using the resulting frequent predicate itemsets. If we consider that the

Table 3 Our case study resulting partitions

Partition	Description	Size(records)	Size(MB)
P_1	Current quarter	2040456	12.78
P_2	Current year(except current quarter and sports category)	2297244	14.71
P_3	Last year(except sports)	4116760	27.28
P_4	Category sports (except current quarter)	2059041	15.19
P_5	Category Jewelry (except 2 last years)	1142632	8.24
P_6	OFF-7(except 2 last years)	1318736	7.98
P_7	Rest	9311584	64.44

Table 4 Approaches comparison parameters

Algorithm	Fact table size	Query workload size	Number of attributes
GA[9]	24M	60	12
HC[9]	24M	60	12
SA[9]	24M	60	12
EMeD-Part[14]	24M	100	12
Our approach	24M	100	12

maximum partition size threshold is 30MB, the min_sup has to be inferior to 9%. We fixed the min_sup to 5% to test the ability of the partitioning algorithm to enlarge the scope.

In the first iteration, the predicate itemset is the current month, which corresponds to the most used predicate. However the support of this latter is inferior to the min_sup. The algorithm tries, thus, to enlarge the population, instead of ignoring the itemset, by replacing the month predicate by its first ancestor in the date hierarchy. The month is then replaced by the quarter which satisfies the min_sup constraint. This results on the partition P_1. The partition P_5 is also obtained by replacing the item class "consignment" by its ancestor in the Jewelry category. We obtained 7 partitions as described by (Table 3):

The partition named "Rest" is the non-partitioned data, which corresponds to rarely or unused data. This partition is partitioned mathematically to respect the storage constraint, otherwise the partition is maintained. The reminder is then partitioned per year to 3 other partitions (P_7, P_8, P_9).

4.2 Results discussion and evaluation

We conducted a set of experiments to measure the efficiency of our approach. We compared our experiments results with results obtained by GA, HC,SA[9] and EMeD-Part[14]. The most important setup parameters are described in the table (Table 4). We started by measuring the execution time of 100 queries before and after partitioning. We noted a great enhancement in queries performances as shown in the (Fig. 8). The enhancement attains 52% depending on the query complexity. We also computed the I/O cost before and after partitioning by calculating the number of pages needed to load the cube or the partitions in the memory to respond to a specified query as it is shown by the following formula: $\sum_{i=1}^{n} \frac{Size(P_i)}{PS}$ Where n is the number of partitions P_i needed to respond to a user query and PS is the system page size. In fact, the (Fig. 9) shows the percentage of I/O cost reduction calculated against the non-partitioned cube. The result is 11.04% of query I/O cost compared to the non-partitioned cube, which is very satisfactory compared to other approaches (see (Table 5)).

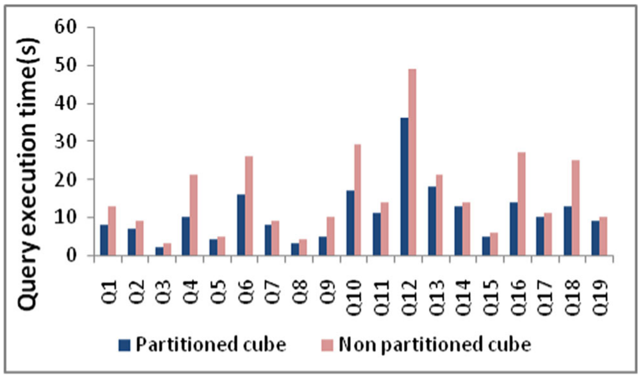


Fig. 8 Query execution time

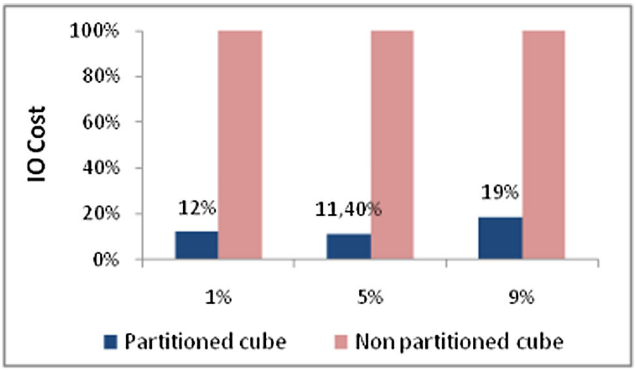


Fig. 9 IO cost reduction

Besides, we conducted multiple experiments to measure the performance of our algorithm. We thus measured the number of iterations of the partitioning algorithm for different min_sup values as shown in (Fig. 10). The experiments show that the number of iterations remains less than 500 independently of the min_sup value, meanwhile it is in the order of thousands in the other approaches[14](2500 in the best case). Therefore, our algorithm needs 5 times iterations less than EMeD-Part and genetic approaches. We also measured the number of generated partitions according to the min_sup threshold (Fig. 11). We noted that the number of partitions is always reasonable which means a low maintenance cost. For

Table 5 Our approach evaluation

Algorithm	Average query time enhancement	I/O cost vs. no partitioning	Number of iteration	number of resulting partitions
GA[9]	13%	37.5%	≅ 5500	80
HC[9]	10.07%	37.78%	–	96
SA[9]	26.72%	38.13%	–	80
EMeD-Part[14]	–	12.5%	≅ 2500	8
Our approach	28.1%	11.04%	198	9

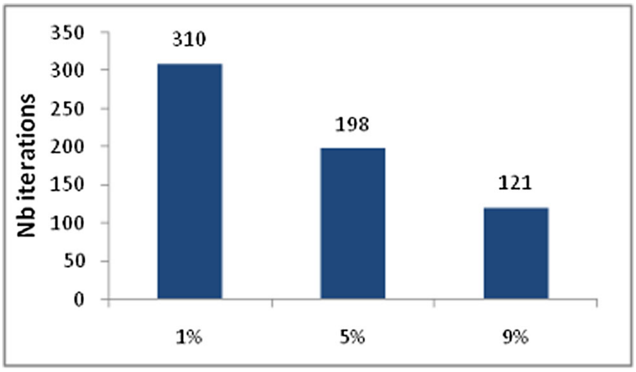


Fig. 10 Number of iterations vs. min_sup

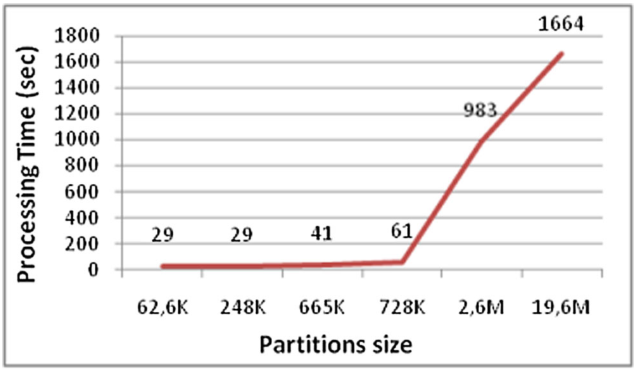


Fig. 12 Processing Time vs. partition siz

instance, our proposal produced 9 partitions for the case study while the genetic-based algorithms [9] produced 80. The table (Table 5) resumes the results comparison between our approach and existing partitioning approaches for the same amount of data and workload, which do not require special configurations.

In addition to query performances, our approach generates dynamic partitions according to the time dimension, which is not supported by the existing approaches, since they treat the date predicates as static values. Moreover, to diminish the number of partitions and hence the maintenance cost, the proposal provides a complementary partitioning strategy based on time range for non-frequently asked data (the rest). Conversely, the other approaches partition the whole cube data by queries predicates which could generate a huge number of partitions. Moreover, in the approaches based on a cost model [9, 14], the DBA has to define complicated parameters like maximum number of IO and the maximum number of partitions thresholds. Whereas our approach does not require such complicated parameters. On the other hand, the partitioning of OLAP cubes enhances also the processing time which can vary from seconds to many hours as shown in the (Fig. 12) depending on the partition size. Actually, reducing the processing time can also help in defining an appropriate cube refresh strategy. For instance, in our case study (Fig. 13) the partition P_1 related to data of the current quarter and which processing time is about 10 seconds, can be used for daily reports and also to recover immediately the urgent ones in case of synchronization breakdowns, and hence being processed many times per day if needed or simply once in the nighttime, while P_2, P_3 can be processed once a month and P_4, P_5, P_6, P_7, P_8 and P_9 once a year.

Finally, we conducted additional experiments to measure the sensitivity of our approach. Indeed, our partitioning algorithm is partially a query workload based approach, which means that while the user queries do not change considerably, the cube performance remains good. Actually, query changes could concern projection or selection columns. To experiment the former case, we considered a query template Q' whose we modified its projection columns. We noted that the performance enhancement remains above to 28% as shown by the (Fig. 14). We next modified the query predicates(selection columns), in this case we noticed performance regression especially when modifying the query completely (see (Fig. 15)). In our case study, the regression varies from 3% to 28%. Therefore, the cube performance (querying and processing) has to be checked periodically to identify performance regression, due for instance to business perspectives changes or data

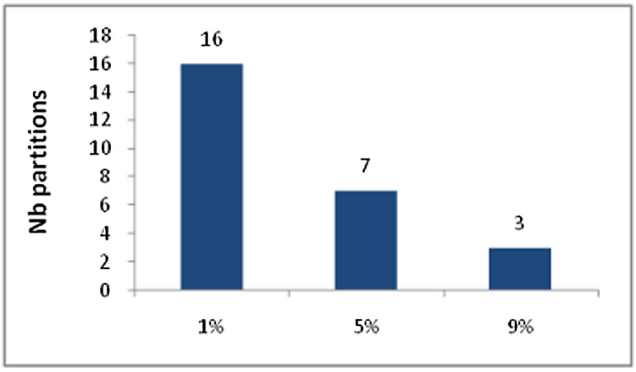


Fig. 11 Number of partitions vs. min_sup

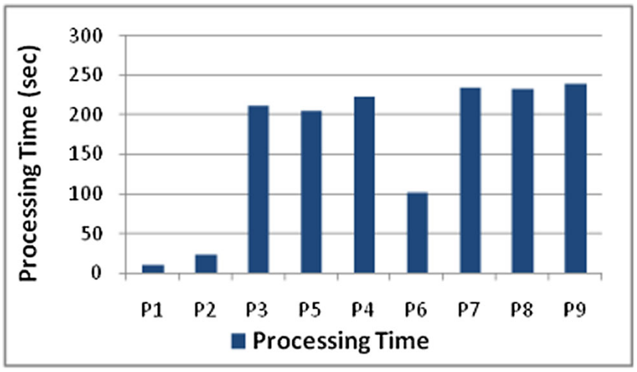


Fig. 13 Processing time of our case study

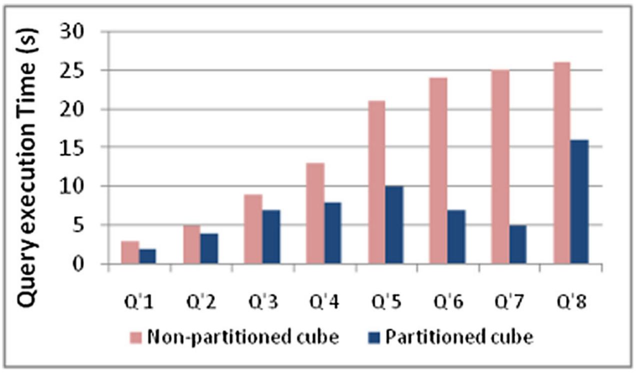


Fig. 14 Sensibility vs query projection columns changes

volume scale increase, and thereby re-run the partitioning algorithm.

5 Conclusion

In our previous works we proposed an MDA architecture to model and implement OLAP cubes [34, 35]. In this paper, and to help BI administrators maintaining those resulting cubes, we presented an approach for OLAP cube partitioning inspired from the association rules algorithm. Contrary to existing approaches, which are based on either workload or data, our approach includes both workload, data, and criticality of queries. Indeed, the proposal starts by analyzing the user queries to deduce the frequent predicate itemsets. Afterwards, by using the resulting predicate itemsets, the proposal identifies the appropriate partitioning strategy according to minimum and maximum partitions size (min_sup and max_size). This also allows controlling and insuring regularity of partitions size which helps in store management beside enhancing cube refresh time and management also depending on partitions size. The user queries analysis phase includes a preprocessing step that especially allows integrating the criticality factor for critical but non-frequent queries and also dealing with date predicates to provide later dynamic time partitioning.

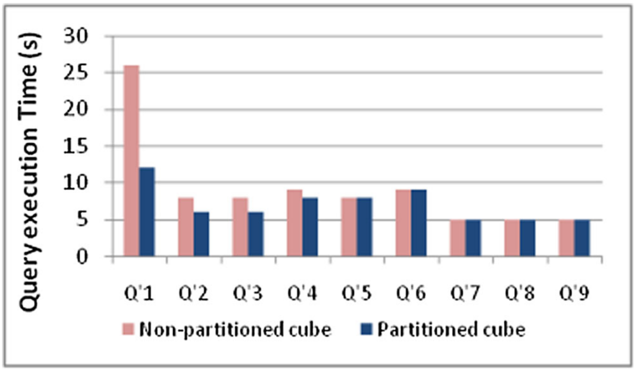


Fig. 15 Sensitivity vs query selection columns changes

Besides, our partitioning algorithm, and in order to provide the best queries performance, tends to create partitions that fit with the user queries or at least encompass them by using the dimension hierarchy to enlarge the scope of the frequent predicate itemset, if needed, instead of ignoring it.

Moreover, in our approach, contrary to existing ones, the non-frequently asked data are isolated in a separate partition that can be partitioned according to storage constraints, and thereby reducing the maintenance cost, instead of using the same partitioning schema as the frequently asked data.

Our proposal comprises the three main advantages of partitioning which are namely: performance improvement, refresh management (maintenance) and store management. Finally, the experimental results show the great enhancement of queries and processing performances after partitioning.

In our future works, we first envisage to automate the check process to be integrated in our partitioning approach. Next, we plan to combine additional optimization method with our proposal especially indexes and parallelism. Afterwards, we intend to adapt and improve our algorithm to support big data and unstructured databases.

References

1. Inmon WH (2005) Building the data warehouse. Wiley, New York
2. Vaisman A, Zimányi E (2014) Data warehouse systems design and implementation. Springer, Berlin
3. AlHammad N, Taha Y (2016) Performance Evaluation Study of Data Retrieval in Data Warehouse Environment. ICCIP '16 ACM, Singapore
4. Kimball R, Ross M (2002) The data warehouse toolkit second edition the complete guide to dimensional modeling. Wiley, New York
5. Common Warehouse Metamodel (CWM) Specification Version 1.1, Volume 1 (March 2003)
6. Meta Data Coalition Open Information Model Version 1.1 (August, 1999)
7. Han J, Kamber M (2006) Data Mining. Elsevier, Amsterdam
8. Bellatreche L, Boukhalfa K (2005) An evolutionary approach to schema partitioning selection in a data warehouse. In: Proceedings of the 7th International Conference DaWaK. LNCS, vol 3589. Springer, Berlin, pp 115–125
9. Bellatreche L, Boukhalfa K, Richard P (2009) Referential horizontal partitioning selection problem in data warehouses: hardness study and selection algorithms. Int J Data Warehouse Min 5(4):1–23
10. Hamdi I, Bouazizi E, Alshomrani S, Feki J (2015) 2LPA-RTDW: a two-level data partitioning approach for real-time data warehouse. Computer and Information Science (ICIS). IEEE, Las Vegas
11. Baluch O, Eavis T (2014) Soft real-time OLAP: exploiting modern hardware without breaking the bank. In: 43rd international conference IEEE parallel processing workshops (ICCPW),

12. Lima A, Furtado C, Valduriez P, Mattoso M (2009) Parallel OLAP query processing in database clusters with data replication. Distrib Parallel Databases 25:97–123
13. Sun L, Krishnan S, Xin RS, Franklin MJ (2014) A partitioning framework for aggressive data skipping. In: International conference on very large data bases, Hangzhou
14. Toumi L, Moussaoui A, Ugur A (2015) EMeD-part: an efficient methodology for horizontal partitioning in data warehouses. In: ACM IPAC '15. Batna
15. Grund M, Krueger J, Mueller J, Zeier A, Plattner H (2011) Dynamic partitioning for enterprise applications. In: Proceedings of IEEE IEEM, pp 1010–1015
16. Bellatreche L, Bouchakri R, Cuzzocrea A, Maabout S (2013) Horizontal partitioning of very-large data warehouses under dynamically-changing query workloads via incremental algorithms. In: SAC'13 proceedings of the 28th annual ACM symposium on applied computing, pp 208–210
17. Bouchakri R, Bellatreche L, Faget Z, Breß S (2014) A coding template for handling static and incremental horizontal partitioning in data warehouses. J Decis Syst 23:4, 481–498
18. Rodriguez L, Li X (2011) A support-based vertical partitioning method for database design. In: 2011 8th international conference on electrical engineering computing science and automatic control (CCE), pp 1–6
19. Bouakkaz M, Ouinten Y, Ziani B (2012) Vertical fragmentation of data warehouses using the FP-Max algorithm. In: 2012 international conference on innovations in information technology (IIT), pp 273–276
20. Arres B, Kabachi N, Boussaid O (2015) A data pre-partitioning and distribution optimization approach for distributed datawarehouses. In: Proceedings of the international conference on parallel and distributed processing techniques and applications (PDPTA), Athens, pp 454–461
21. Kim JW, Cho SH, Kim I-M (2016) Workload-based column partitioning to efficiently process data warehouse query. Int J Appl Eng Res 11(2):917–921
22. Ahmed S, Coenen F, Leng P (2006) Tree-based partitioning of date for association rule mining. Knowl Inf Syst 315–331
23. Patil DV (2015) Reducing data skew with round robin horizontal partitioning of data for distributed association rule mining of large data set. IJETT
24. Le-Khac NA, Kechadi MT, Carthy J (2006) ADMIRE framework: distributed data mining on data grid platforms. In: Proceedings of the first international conference on software and data technologies. ICSOFT
25. Gorla N (2003) Features to consider in a data warehousing system. Commun ACM 46(11):111–115
26. Cheung DW, Zhou B, Kao B, Kan H, Lee SD (2001) Towards the building of a dense-region-based OLAP system. Data Knowl Eng 36:1–27
27. Partitions (Analysis Services - Multidimensional Data) https://msdn.microsoft.com/en-us/library/ms175688.aspx. Accessed: 21 Sep 2017
28. SAS 9.1.3 OLAP Server: MDX Guide, Second Ed - SAS Support, MDX Introduction and Overview
29. Agrawal R, Imielinski T, Swami A (1993) Mining association rules between sets of items in large databases. In: Proceedings of the 1993 ACM SIG MOD Conference. Washington DC, USA
30. Ben Messaoud R, SL Rabasda, Boussaid O, Missaoui R (2006) Enhanced mining of association rules from data Cubes. In: DOLAP'06, November 10, 2006. Arlington, USA
31. Ponniah P (2001) Data warehousing fundamentals: a comprehensive guide for IT professionals
32. Shukla A, Deshpande P, Naughton JF (1996) Storage estimation for multidimensional aggregates in the presence of hierarchies, http://ai2-s2-pdfs.s3.amazonaws.com
33. TPC-DS database: http://www.tpc.org/tpcds. Accessed: 21 Nov 2017
34. Letrache K, El Beggar O, Ramdani M (2017) The automatic creation of OLAP cube using an MDA approach. Softw: Pract Exp 47(12):1887–1903
35. El Beggar O, Letrache K, Ramdani M (2017) CIM for data warehouse requirements using an UML profile. IET Softw 11(4): 181–194