Chapter 2
Dimension Table Selection Strategies to Referential Partition a Fact Table of Relational Data Warehouses

Ladjel Bellatreche

2.1 Introduction

Enterprise wide data warehouses are becoming increasingly adopted as the main source and underlying infrastructure for business intelligence (BI) solutions. Note that a data warehouse can be viewed as an *integration system*, where data sources are duplicated in the same repository. Data warehouses are designed to handle the queries required to discover trends and critical factors are called Online Analytical Processing (OLAP) systems. Examples of an OLAP query are: *Amazon* (www.amazon.com) company analyzes purchases by its customers to come up with an individual screen with products of likely interest to the customer. Analysts at *Wal-Mart* (www.walmart.com) look for items with increasing sales in some city. Star schemes or their variants are usually used to model warehouse applications. They are composed of thousand of dimension tables and multiple fact tables [15, 18]. Figure 2.1 shows an example of star schema of the widely-known data warehouse benchmark *APB-1 release II* [21]. Here, the fact table *Sales* is joint to the following four dimension tables: *Product*, *Customer*, *Time*, *Channel*. *Star queries* are typically executed against the warehouse. Queries running on such applications contain a large number of costly joins, selections and aggregations. They are called mega queries [24]. To optimize these queries, the use of advanced optimization techniques is necessary.

By analyzing the most important optimization techniques studied in the literature and supported by the most important commercial Database Management Systems (DBMS) like Oracle, SQL Server, DB2, etc. we propose to classify them into two main categories: (1) *optimization techniques selected during the creation of data warehouses* and (2) *optimization techniques selected after the creation of the warehouses*. The decision of choosing techniques in the first category is

L. Bellatreche (✉)
LISI/ENSMA – Poitiers University, 1, avenue Clément Ader, 86960 Futuroscope, France
e-mail: bellatreche@ensma.fr

T. Özyer et al. (eds.), *Recent Trends in Information Reuse and Integration*,
DOI 10.1007/978-3-7091-0738-6_2,

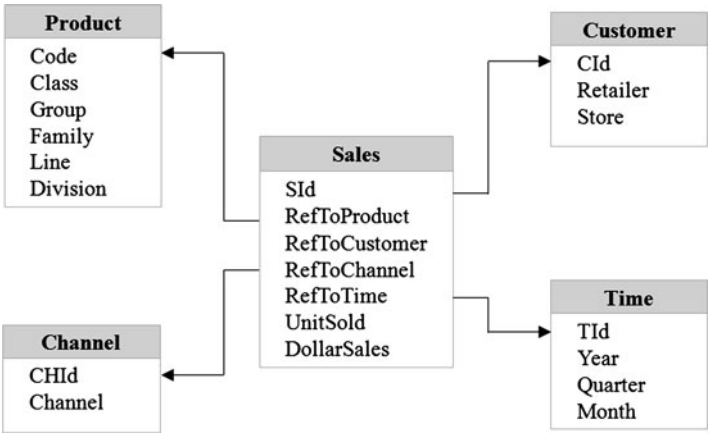


Fig. 2.1 Logical schema of the data warehouse benchmark *APB-1 release II*

taken before creating the data warehouse. Horizontal partitioning of tables and parallel processing are two examples of the first category. Techniques belonging to the second category are selected during the exploitation of the data warehouse. Examples of these techniques are materialized views and indexes, data compression, etc. Based on this classification, we note that the use of optimization techniques belonging to the first category is more sensitive compared to those belonging to the second one, since the decision of using them is usually taken at the beginning stage of the data warehouse development. To illustrate this sensitivity, imagine the following two scenarios: (1) a database administrator (DBA) decides to partition her/his warehouse using a native data definition language (DDL) proposed by his/her favorite DBMS, later on, DBA realizes that data partitioning is not well adapted for his/her data warehouse. Therefore, it will be costly and time consuming to reconstitute the initial warehouse from the partitions. (2) DBA selects indexes or materialized views (techniques belonging to the second category) and if she/he identifies their insufficiencies to optimize queries, she/he can easily drop or replace them by other optimization techniques. This sensitivity and the carefulness of optimization techniques belonging to the first category motivate us to study one of them, known as referential horizontal partitioning.

Horizontal partitioning is one of important aspect of physical design [23]. It is a divide-and-conquer approach that improves query performance, operational scalability, and the management of ever-increasing amounts of data [28]. It divides tables/views into disjoint sets of rows that are physically stored and accessed separately. Unlike indexes and materialized views, it does not replicate data, thereby reducing storage requirement and minimizing maintenance overhead. Most of today's DBMSs (*Oracle, DB2, SQL Server, PostgreSQL, Sybase* and *MySQL*) offer native Data Definition Language support for creating partitioned tables (*Create Table ... Partition*) and manipulating horizontal partitions (*Merge* and *Split*) [9, 20, 23]. Partitioning is not restricted to data tables. Indexes may also be

partitioned. A local index may be constructed so that it reflects the structure of an underlying table [23]. Notice that when horizontal partitioning is applied on materialized views and indexes, it will be considered as a technique of the second category. Its classification depends on the nature of the used access method (table, materialized view or index).

Horizontal partitioning improves performance of queries by the mean of pruning mechanism that reduces the amount of data retrieved from the disk. When a partitioned table is queried, the database optimizer eliminates non relevant partitions by looking the partition information from the table definition, and maps that onto the Where clause predicates of the query. Two main pruning are possible: partition pruning and partition-wise join pruning [9]. The first one prunes out partitions based on the partitioning key predicate. It is used when only one table is partitioned. The second one prunes partitions participating in a join operation. It is applicable when the partitioning key is same in the two tables.

Two main types of horizontal partitioning exist [5]: mono table partitioning and table-dependent partitioning. In the mono table partitioning, a table is partitioned using its own attributes. It is quite similar to the primary horizontal partitioning proposed in traditional databases [22]. Several modes exist to support mono table partitioning: *Range, List, Hash, Round Robin* (supported by Sybase), *Composite* (Range-Range, Range-List, List-List), etc.), Virtual Column partitioning recently proposed by *Oracle11G*, etc. Mono table partitioning may be used to optimize selections, especially when partitioning key matches with their attributes (partition pruning). In the data warehouse context, it is well adapted for dimension tables. In table-dependent partitioning, a table inherits the partitioning characteristics from other table. For instance a fact table may be partition based on the fragmentation schemes of dimension tables. This partitioning is feasible if a parent-child relationship among these tables exists [7, 9]. It is quite similar to the derived horizontal partitioning proposed in 1980s in the context of distributed databases [7]. Two main implementations of this partitioning are possible: native referential partitioning and simulated referential partitioning. The native referential partitioning is recently supported by Oracle11G to equi-partition tables connected by a parent child referential constraint. A native DDL is given to perform this partitioning [9] (*Create Table ... Partition by Reference ...*). It optimizes selections and joins simultaneously. Therefore, it is well adapted for star join queries.

Simulated referential partitioning is a manual implementation of referential partitioning, i.e., it simulates it using the mono table partitioning mode. This simulation may be done as follows: (1) a dimension table is first horizontally partitioned using its primary key, then the fact table is decomposed based on its foreign key referencing that dimension table. This partitioning has been used for designing parallel data warehouses [10], but it is not well adapted for optimizing star join queries [9]. Figure 2.2 classifies the proposed partitioning modes.

By analyzing the most important works done on referential partitioning in the data warehousing environment, we draw the two following points: (a) most of proposed algorithms control the number of generated fragments of the fact tables in order to facilitate the manageability of the data warehouse [2, 16, 25] and

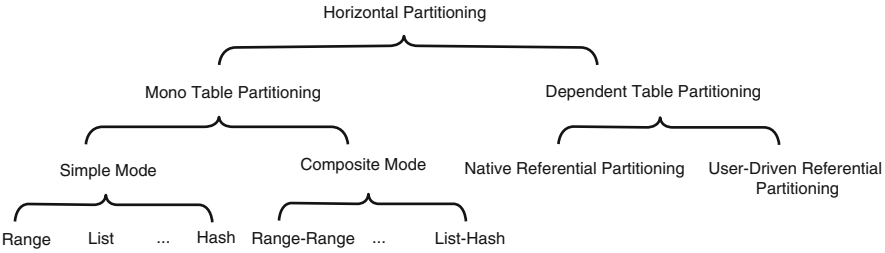


Fig. 2.2 A classification of the proposed partitioning modes

(b) the fragmentation algorithms start from a bag containing all selection predicates involved by the queries. Therefore, dimension tables referenced by these predicates have the same chance to be used to referential partition a fact table of a warehouse. This situation is not always the best choice to optimize queries. This finding is based on the following observation: To execute a star join query involving several tables in a non partitioned data warehouse, the query optimizer first shall establish an order of joins [26]. This order should reduce the size of intermediate results. To execute the same query on a horizontally partitioned warehouse, the query optimizer will do the same task as in the non partitioned scenario. The join order is somehow partially imposed by the fragmentation process, since fragments of the fact table will be first joined with the fragments of a dimension table used in the referential partitioning process in order to perform partition-wise join pruning. Based on this analysis, we notice that the choice of dimension table(s) used to partition the fact table is a crucial performance issue that should be addressed. Contrary to the existing horizontal partitioning approaches that consider all dimension tables involved by the queries, we propose to identify relevant dimension table(s) and then generate fragmentation schemes using existing algorithms. This partitioning manner will reduce the complexity of the partitioning problem which is known as a NP-complete problem [3], since selection predicates belonging to non desirable dimension table(s) will be discarded. Note that the complexity of horizontal partitioning is proportional to the number of predicates [22].

For data warehouse applications managing a reasonable number of dimension tables, DBA may identify manually dimension tables based on her/his experience, but for BI applications, where hundred of dimension tables are used, it will be hard to do it manually. Therefore, the development of automatic techniques offering DBA the possibility to select dimension tables to partition her/his fact table is necessary. In this paper, we propose a comprehensive procedure for automatically selecting dimension table(s) candidate to referential partition a fact table.

This paper is divided in six sections: Section 2.2 reviews the existing works on horizontal partitioning in traditional databases and data warehouses. Section 2.3 proposes selection strategies of dimension table(s) to support referential horizontal partitioning. Section 2.4 gives the experimental results done on the data set of the APB-1 benchmark using a mathematical cost model and on Oracle11. Section 2.5,

a tool supporting our proposal is given. It is connected to *Oracle11G* DBMS. Section 2.6 concludes the paper by summarizing the main results and suggesting future work.

2.2 Related Work

Many research works dealing with horizontal partitioning problem were proposed in traditional databases and data warehouses. In the traditional database environment, researchers concentrate their efforts on proposing algorithms for the mono table partitioning. These algorithms are guided by a set of selection predicates of queries involving the table to be partitioned. A selection predicate is defined as follows: *attribute* θ *value*, where *attribute* belongs to the table to be fragmented, $\theta \in \{>, >, =, \leq, \geq\}$ and $value \in$ domain of attribute. In [3], a classification of these algorithms is proposed: *minterm generation-based approach* [22], *affinity-based approach* [13] and *cost-based approach* [4].

Online analysis applications characterized by their complex OLAP queries motivate the data warehouse community to propose table-dependent horizontal partitioning methodologies and algorithms. [19] proposed a methodology to partition a relational data warehouse in a distributed environment, where the fact table is referentially partitioned based on queries defined on all dimension tables. [17] proposed a fragmentation methodology for a multidimensional warehouse, where the global hypercube is divided into sub cubes. Each one contains a sub set of data. This process is defined by slice and dice operations (similar to the selection and the projection in relational databases). Relevant dimensions to partition the hyper cube are chosen manually. In [3], a methodology and algorithms dealing with partitioning problem in relational warehouses are given. In this work, the authors propose a fragmentation methodology of the fact table based on the fragmentation schemas of dimension tables. Each dimension table involved in any selection predicate is a candidate to referential partition the fact table. Three algorithms selecting fragmentation schemes reducing query processing cost were proposed [3]: *genetic*, *simulated annealing* and *hill climbing*. These algorithms control of generated fragments of a fact table. The main characteristic of these algorithms is that they use a cost model to evaluate the quality of the obtained fragmentation schemes. In [5], a *preliminary work* on selecting dimension table(s) to partition a relational fact table is presented.

In [16], an approach for fragmenting XML data warehouses is proposed. It is characterized by three main steps: (1) extraction of selection predicates from the query workload, (2) predicate clustering and (3) fragment construction with respect to predicate clusters. To form clusters of predicates, the authors proposed to use the k-means technique. As in [3], this approach controls the number of fragments. [23] showed the need of integrating horizontal and vertical partitioning in physical database design, but they did not present algorithm to support referential partitioning of a data warehouse.

To summarize, most important research efforts on horizontal partitioning in the traditional database are focused on the mono table partitioning, whereas in data warehouses, the efforts are concentrated on dependent table partitioning. Proposed algorithms on fragmenting a fact table consider whether all dimension tables of a data warehouse schema or a set of these tables identified by DBA. The choice of these dimension tables to partition a fact table is a crucial issue for optimizing OLAP queries. Unfortunately, the problem of selecting dimension table(s) to partition the warehouse is not well addressed; except the work done by [17] that pointed out the idea of choosing dimension to partition a hyper cube, without giving strategies to perform this selection.

2.3 Selection of Dimension Tables

In this section, we propose first a formalization of the problem of selecting dimension tables and then strategies aiding DBA in choosing her/his relevant dimension tables used to referential partition a fact table.

2.3.1 Formalization of the Problem

Before proposing this methodology, we formalize the referential horizontal partitioning problem as follows:

Given a data warehouse with a set of d dimension tables $D = \{D_1, D_2, \ldots, D_d\}$ and a fact table F, a workload Q of queries $Q = \{Q_1, Q_2, \ldots, Q_m\}$ and a maintenance constraint W fixed by DBA that represents the maximal number of fact fragments that he/she can maintain. The referential horizontal partitioning problem consists in (1) identifying candidate dimension table(s), (2) splitting them using single partitioning mode and (3) using their partitioning schemes to decompose the fact table into N fragments, such that: (a) the cost of evaluating all queries is minimized and (b) $N \leq W$.

Based on this formalization, we note that referential partitioning problem is a combination of two difficult sub problems: *identification of relevant dimension tables* and their *fragmentation schema selection* (which is known as a NP-complete problem [2]). In the next section, we study the complexity of the first sub problem (identification of relevant dimension tables).

2.3.2 Complexity of Selecting Dimension Tables

The number of possibilities that DBA shall consider to partition the fact table using referential partitioning is given by the following equation:

$$\binom{d}{1} + \binom{d}{2} + \ldots + \binom{d}{d} = 2^d - 1 \tag{2.1}$$

Example 1. Let us suppose a warehouse schema with three dimension tables *Customer*, *Time* and *Product* and one fact table *Sales*. Seven scenarios are possible to referential partition *Sales*: *Sales(Customer), Sales(Time), Sales(Product), Sales (Customer, Time), Sales (Customer, Product), Sales (Product, Time)* and *Sales (Customer, Time, Product)*. For a data warehouse schema with a large number of dimension tables, it will be hard for DBA to choose manually relevant dimension table(s) to partition the fact table and guarantying a reduction of query processing cost. For instance, the star schema of *Sloan Digital Sky Survey database*, which is a real-world astronomical database, running on SQL Server has 18 dimension tables [12]. In this context, DBA needs to evaluate $2^{18} - 1$ (=262 143) possibilities to partition her/his fact tables.

This formalization and real partitioning scenario motivate us to propose strategies aiding DBA to select relevant dimension tables.

2.3.3 Dimension Table Selection Strategies to Split the Fact Table

Referential partitioning is proposed essentially to optimize join operations and facilitate the data manageability [9, 23]. Therefore, any selection strategy should hinge on join operation which well present in OLAP queries.

1. *Largest dimension table:* a simplest way to select a dimension table to referential partition the fact table is to choose the largest dimension table. This selection is based on the following observation: joins are typically expensive operations, particularly when the relations involved are substantially larger than main memory [14].
2. *Most frequently used dimension table:* the second naive solution is to choose the most frequently used dimension table(s). The frequency of a dimension table Di is calculated by counting its appearance in the workload.

   These two naive strategies are simple to use. But, they ignore several logical and physical parameters like size of tables, buffer size, size of intermediate results of joins that has a real impact on join cost, etc. Generally, when query involves multiple joins (which is the case of star join queries), the cost of executing the query can vary dramatically depending on the query evaluation plan (QEP) chosen by the query optimizer. The join order and access methods used are important determinants of the lowest cost QEP [27]. The query optimizer estimates the intermediate results and uses this information to choose between different join orders and access methods. Thus, the estimation of join result sizes in a query is an important problem, as the estimates have a significant influence on the QEP chosen. Based on these observations, the choice of dimension table(s) should take into account these the sizes of intermediate results.
3. *Minimum share:* in data warehousing, every join operation involves the fact table. The query graph of such a query contains a root table (representing the fact table)

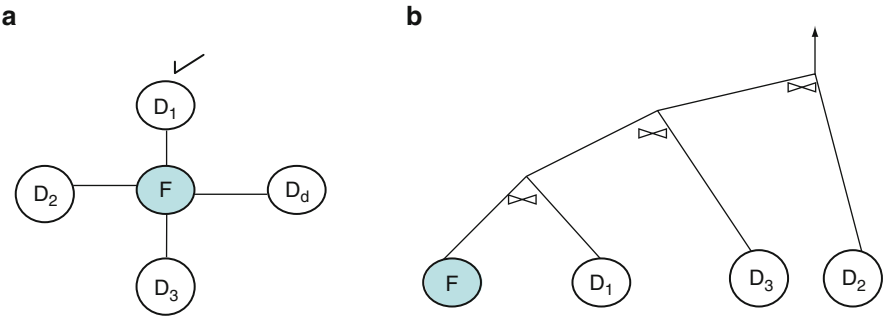


Fig. 2.3 (**a**) Star Query Graph, (**b**) Star query execution using left deep tree

and peripheral tables (representing dimension tables) (Fig. 2.3a). To optimize the execution of a star join query, an optimal sequence in which the tables are joined shall be identified. This is problem is known as *join ordering* [26]. Intuitively, this problem is quite similar to dimension table selection problem. Join ordering is specific to one query (Fig. 2.3b). An important difference between these two problems is that join order concerns only one query, but dimension table selection problem concerns several queries running on data warehouse. As in join ordering problem [26], the best choice of dimension table should reduce the size of intermediate results of the most important join star queries. Based on this analysis, we propose a strategy to referential partition the fact table reducing the size of intermediate results. The size (in terms of number of tuples) of join between the fact table F and a dimension table D_i (which is a part of a star join query), denoted by $\|F \bowtie D_i\|$ is estimated by:

$$\|F \bowtie D_i\| = \|D_i\| \times Share(F.A_i) \tag{2.2}$$

where $\|D_i\|$ and $Share(F.A_i)$ represent respectively, the cardinality of dimension table D_i and the average number of tuples of the fact table F that refer to the same tuple of the dimension table D_i via the attribute A_i. To reduce the intermediate result of a star join query, we should choose the dimension table with a *minimum share*.

If we have a star join query involving several dimension tables, a simplest join order is to use the dimension table with a minimum share to ensure a reduction of the intermediate result. Therefore, we propose to choose the dimension table with a minimum share to referential partition the fact table. Several studies were proposed to estimate the share parameter, especially in object oriented databases [8].

2.3.4 Referential Partitioning Methodology

Now, we have all ingredients to propose our methodology offering a real utilization of data partitioning in data warehouse application design. It is subdivided into three steps described in Fig. 2.4:

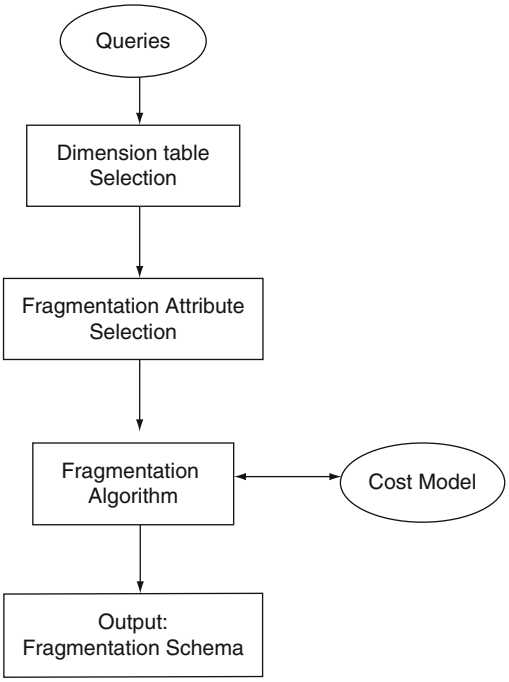


Fig. 2.4 A methodology to support referential partitioning

1. Selection of dimension table(s) used to decompose the fact table. This selection may be done using one of the previous criteria.
2. Extraction of candidate fragmentation attributes (key partitioning). A fragmentation attribute is an attribute used by a selection predicate defined on a chosen dimension table.
3. Partition each dimension table using single table partitioning type. To generate partitioning schemes of chosen dimension tables, DBA may use his/her favourite algorithm. The complete generation of fragmentation schemes (of fact and dimension tables) is ensured by using a particular coding [2]. Each fragmentation schema is represented using a multidimensional array, where each cell represents a domain partition of a fragmentation schema.

   To illustrate this coding, let us suppose the following scenario: DBA choose dimension table *Customer* and Time to referential partition the fact table Sales. Three attributes: Age, Gender, Season, where Age and Gender belong to *Customer* dimension table, whereas Season belongs to Time. The domain of these attributes are: $Dom(Age) = [0, 120]$, $Dom(Gender) = \{$'M', 'F'$\}$, and $Dom(Season) = \{$"Summer", "Spring", "Autumn", "Winter"$\}$. DBA proposes an initial decomposition of domains of these three attributes as follows: $Dom(Age) = d_{11} \cup d_{12} \cup d_{13}$, with $d_{11} = [0, 18]$, $d_{12} =]18, 60[$, $d_{13} = [60, 120]$. $Dom(Gender) = d_{21} \cup d_{22}$, with $d_{21} = \{'M'\}$, $d_{22} = \{'F'\}$. $Dom(Season) = d_{31} \cup$

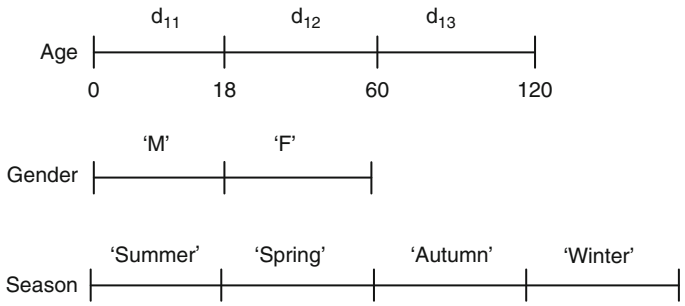


Fig. 2.5 Decomposition of attribute domains

Fig. 2.6 Coding of a fragmentation schema

Age	1	2	3	
Gender	1	2		
Season	1	2	3	4

$d_{32} \cup d_{33} \cup d_{34}$, where $d_{31} = \{$"Summer"$\}$, $d_{32} = \{$"Spring"$\}$, $d_{33} = \{$"Autumn"$\}$, and $d_{34} = \{$"Winter"$\}$. Sub domains of all three fragmentation attributes are represented in Fig. 2.5.

Domain partitioning of different fragmentation attributes may be represented by multidimensional arrays, where each array represents the domain partitioning of a fragmentation attribute. The value of each cell of a given array representing an attribute belongs to $(1..n_i)$, where n_i represents the number of sub domain of the attribute (see Fig. 2.6). Based on this representation, fragmentation schema of each dimension table D_j is generated as follows:

- All cells of a fragmentation attribute of D_j have different values: this means that all sub domains will be used to partition D_j. For instance, the cells of each fragmentation attribute in Fig. 2.5 are different. Therefore, they all participate in fragmenting their corresponding tables (*Customer* and Time). The final fragmentation schema will generate 24 fragments of the fact table.
- All cells of a fragmentation attribute have the same value: this means that it will not participate in the fragmentation process. Table 2.1 gives an example of a fragmentation schema, where all sub domains of Season (of dimension table Time) have the same value; consequently, it will not participate in fragmenting the warehouse schema.
- Some cells has the same value: their corresponding sub domains will be merged into one. In Table 2.1, the first ([0, 18]) and the second (]18, 60[) sub domains of Age will be merged to form only one sub domain which is the union of the merged sub domains ([0, 60[). The final fragmentation attributes are: Gender and Age of dimension table *Customer*.

Table 2.1 Example of a fragmentation schema

Table	Attribute	SubDomain	SubDomain	SubDomain	SubDomain
Customer	Age	1	1	2	
Customer	Gender	1	2		
Time	Season	1	1	1	1

4. Partition the fact table using referential partitioning based on the fragmentation schemes generated by our algorithm.

To illustrate this step, let us consider an example. Suppose that our algorithm generates a fragmentation schema represented by a coding described by Table 2.1. Based on this coding, DBA can easily generate scripts using DDL to create a partitioned warehouse. This may be done as follows: *Customer* is partitioned using the composite mode (*Range* on attribute *Age* and *List* on attribute *Gender*) and the fact table using the native referential partitioning mode.

```
CREATE TABLE Customer
(CID NUMBER, Name Varchar2(20), Gender CHAR, Age Number)
PARTITION BY RANGE (Age)
SUBPARTITION BY LIST (Gender)
SUBPARTITION TEMPLATE (SUBPARTITION Female VALUES ('F'),
SUBPARTITION Male VALUES ('M'))
(PARTITION Cust_0_60 VALUES LESS THAN (61),
PARTITION Cust_60_120 VALUES LESS THAN (MAXVALUE));
```

The fact table is also partitioned into 4 fragments as follows:

```
CREATE TABLE Sales (customer_id NUMBER NOT NULL,
product_id NUMBER NOT NULL,
time_id Number NOT NULL, price NUMBER,
quantity NUMBER,
constraint Sales_customer_fk foreign key(customer_id)
references CUSTOMER(CID))
PARTITION BY REFERENCE (Sales_customer_fk);
```

2.4 Experimental Results

We have conducted an intensive experimental study to evaluate our proposed strategies for choosing dimension tables to referential partition a fact table. Since, the decision of choosing the horizontal partitioning technique is done before populating the data warehouse; the development of a mathematical model estimating the cost of processing a set of queries is required. In this study, we develop a cost model estimating the number of inputs outputs required for executing this set of queries. It takes into account size of generated fragments of the fact tables ($\|F_i\|, 1 \leq i \leq N$); the number of occupied pages of each fragment F_i, denoted by $|F_i|\left(=\frac{\|F_i\| \times TL}{PS}\right)$, where PS and TL represent respectively, the page size of disk and length of each

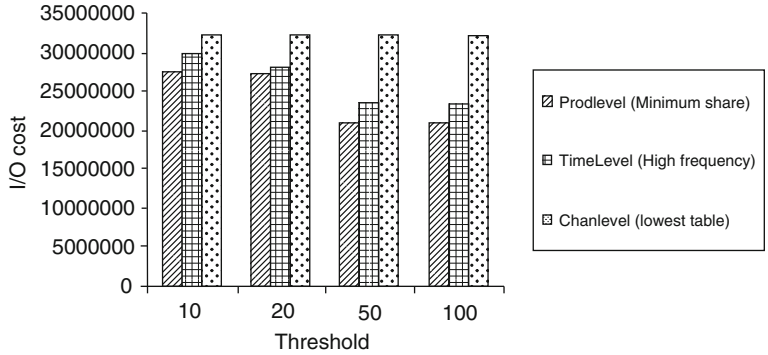


Fig. 2.7 Theoretical results

instance of the fact table), buffer size, etc. The obtained results are then validated on Oracle11G DBMS in order to evaluate the effectiveness of our cost model.

Dataset: We use the dataset from the APB1 benchmark [21]. The star schema of this benchmark has one fact table *Actvars* (33 323 400 tuples) and four dimension tables: *Prodlevel* (9,900 tuples), *Custlevel* (990 tuples), *Timelevel* (24 tuples) and *Chanlevel* (10 tuples).

Workload: We have considered a workload of 55 single block queries with 55 selection predicates defined on 10 different attributes (Class_Level, Group_Level, Family_Level, Line_Level, Division_Level, Year_Level, Month_Level, Quarter_Level, Retailer_Level, All_Level). The domains of these attributes are split into: 4, 2, 5, 2, 4, 2, 12, 4, 4, 5 sub domains, respectively. We did not consider update queries (update and delete operations). Note that each selection predicate has a selectivity factor computed using SQL queries executed on the data set of APB1 benchmark. Our algorithms have been implemented using Visual C++ performed under Intel Pentium 4 with a memory of 3 Gb.

2.4.1 Theoretical Evaluation

The used cost model computes the inputs and outputs required for executing the set of 38 queries[1]. It takes into account the size of intermediate results of joins and the size of buffer. The cost final of 55 queries is computed as the sum of the cost of each query (for more details see [6]).

We use a hill climbing algorithm to select fragmentation schema of the APB1 warehouse due to its simplicity and less computation time [3]. Other algorithms may

[1]These queries are described in Appendix Section at: http://www.lisi.ensma.fr/ftp/pub/documents/papers/2009/2009-DOLAP-Bellatreche.pdf

be used such as simulated annealing and genetic [1]. Our hill climbing algorithm consists of the following two steps:

1. Find an initial solution and iteratively improve this solution by using hill climbing heuristics until no further reduction in total query processing cost. The initial solution is represented by multidimensional array, whose cells are filled in uniform way [3]. Each cell i of fragmentation attribute (A_k) is filled using the following formula:

$$Array[i]_k = \left\lfloor \frac{i}{n} \right\rfloor \tag{2.3}$$

   where n is an integer $(1 \leq n \leq max_{1 \leq j \leq C}(n_j))$, where C is the number of fragmentation attributes. To illustrate this distribution, let us consider three fragmentation attributes: *Gender*, *Season* and *Age*, where the numbers of sub domains of each attributes are respectively, 2, 4 and 3 (see Fig. 2.8). The generated fragments of partitioning schema corresponding to $n = 2, n = 3$ and $n = 4$ are 12, 4 and 2 respectively. If DBA wants an initial fragmentation schema with a large number of fragments, she/he considers n with a low value. In this case, all initial sub domains (proposed by DBA) have the same probability to participate on the fragmentation process.
2. The previous initial solution can be improved by introducing two specialized operators, namely Merge and Split, which allow us to further reduce the total query processing cost due to evaluating queries in Q. Let us now focus on the formal definitions of these operators. Given a fragmentation attribute A_k of a dimension table D_j having $FS(D_j)$ as fragmentation scheme, Merge operation takes as input two domain partitions of $A_k \in FS(D_j)$, namely and returns as output a new fragmentation scheme for D_j, denoted by $FS'(D_j)$, where these two domains are merged into a singleton domain partition of A_k. *Merge* reduces the number of fragments generated by means of the fragmentation scheme

Gender	0	1		
Season	0	1	1	2
Age	0	1	2	

n = 2

Gender	0	0		
Season	0	0	1	1
Age	0	0	1	

n = 3

Gender	0	0		
Season	0	0	0	1
Age	0	0	0	

n = 4

Fig. 2.8 Different coding of an initial solution

$FS(D_j)$ of dimension table D_j, hence it is used when the number of generated fragments does not satisfy the maintenance constraint W.

Given a fragmentation attribute A_k of a dimension table D_j having $FS(D_j)$ as fragmentation scheme, *Split* takes as input a domain partition of Ak in $FS(D_j)$ and returns as output a new fragmentation scheme for D_j, denoted by $FS(D_j)$, where that domain is split into two distinct domain partitions. Split increases the number of fragments generated by means of the fragmentation scheme $FS(D_j)$ of D_j.

On the basis of these operators running on fragmentation schemes of dimensional tables, the hill climbing heuristic still finds the final solution, while the total query processing cost can be reduced and the maintenance constraint W can be satisfied.

The first experiments evaluate the quality of each criterion: *minimum share*, *high frequency*, and *largest size*. The size and frequency of each candidate dimension to partition the fact table are easily obtained from the data set of APB1 benchmark and the 38 queries. The share criterion is more complicated to estimate, since it requires more advanced techniques such histograms [11]. For our case, we calculate the *share* using SQL queries executed on the data set of APB1 benchmark created on Oracle11G. For instance, the *share* of the dimension table *CustLevel* is computed by the following query:

```
Select avg(Number) as AvgShare
    FROM (Select distinct customer_level, count(*) as Number
          From actvars
          Group By Customer_level);
```

For each criterion, we run our hill climbing algorithm with different values of the threshold (representing the number of generated fragments fixed by DBA) set to 10, 20, 50 and 100. For each obtained fragmentation schema, we estimate the cost of evaluating the 38 queries. At the end, 12 fragmentation schemes are obtained and evaluated. Figure 2.7 summarizes the obtained results. The main lessons behind these results are: (1) the minimum share criterion outperforms other criteria (frequency, maximum share), (2) minimum share and largest dimension table criteria give the same performance. This is because, in our real data warehouse, the table having the *minimum share* is the largest one and (3) the threshold has a great impact on query performance. Note that the fact of increasing the threshold does not mean getting a better performance, since when it is equal to 50 and 100; we got practically the same results. Having a large number of fragments increases the number of union operations which can be costly, especially when the size of manipulating partition instances is huge.

2.4.2 Validations on ORACLE 11G

To validate the theoretical results, we conduct experiments using Oracle11G. We choose this DBMS because it supports referential horizontal partitioning. The data

set of ABP1 benchmark is created and populated using generator programs offered by APB1 [21]. During this validation, we figure out that *Composite partitioning* with more than two modes is not directly supported by Oracle11G DDL[2]. To deal with this problem, *we propose the use of virtual partitioning column proposed by Oracle11G*. A virtual column is an expression based on one or more existing columns in the table. The virtual column is only stored as meta-data and does not consume physical space. To illustrate the use of this column, let D_i be a partitioned table in N_i fragments. Each instance of this table belongs to a particular fragment. The identification of the relevant fragment of a given instance is done by matching partitioning key values with instance values. To illustrate this mechanism, suppose that dimension table *ProdLevel* is partitioned into eight partitions by the hill climbing algorithm using three attributes. A *virtual column PROD_COL* is added into *ProdLevel* in order to facilitate its partitioning using the *List* mode.

mode:

```
CREATE TABLE Prodlevel(
CODE_LEVEL CHAR(12) NOT NULL, CLASS_LEVEL CHAR(12) NOT NULL, GROUP_LEVEL CHAR(12) NOT NULL,
FAMILY_LEVEL CHAR(12) NOT NULL, LINE_LEVEL CHAR(12) NOT NULL, DIVISION_LEVEL CHAR(12) NOT NULL,
PROD_COL NUMBER(5) generated always as (case when Class_Level
IN('ADX8MBFPWVIV','OC2WOOC8Q
6','LB2RKO0ZQCJD')
and  Group_Level='VTL9DOE3RSWQ' and Family_Level
in('JIHR5NBAZWGU','OX3BXTCVRRKU','M32G5M3AC4T5', 'Y45VKMTJDNYR') then 0
when Class_Level IN ('ADX8MBFPWVIV','OC2WOOC8QIJ6','LB2RKO0ZQCJD') and
Group_Level='VTL9DOE3RSWQ' and Family_Level  not in
('JIHR5NBAZWGU','OX3BXTCVRRKU','M32G5M3AC4T5', 'Y45VKMTJDNYR')
then 1 when Class_Level IN
('ADX8MBFPWVIV','OC2WOOC8QIJ6','LB2RKO0ZQCJD') and  Group_Level
not in ('VTL9DOE3RSWQ') and Family_Level in
('JIHR5NBAZWGU','OX3BXTCVRRKU','M32G5M3AC4T5','Y45VKMTJDNYR')
 then 2 when Class_Level IN ('ADX8MBFPWVIV','OC2WOOC8QIJ6','LB2RKO0ZQCJD')
 and Group_Level not in ('VTL9DOE3RSWQ') and
Family_Level  not in('JIHR5NBAZWGU','OX3BXTCVRRKU','M32G5M3AC4T5','Y45VKMTJDNYR')
then 3  when Class_Level NOT IN ('ADX8MBFPWVIV','OC2WOOC8QIJ6','LB2RKO0ZQCJD')
and  Group_Level='VTL9DOE3RSWQ' and Family_Level  in
('JIHR5NBAZWGU','OX3BXTCVRRKU','M32G5M3AC4T5','Y45VKMTJDNYR')
then 4 when Class_Level NOT IN ('ADX8MBFPWVIV','OC2WOOC8QIJ6','LB2RKO0ZQCJD')
and  Group_Level='VTL9DOE3RSWQ' and Family_Level  not in
('JIHR5NBAZWGU','OX3BXTCVRRKU','M32G5M3AC4T5','Y45VKMTJDNYR') then 5
Else  7  end), PRIMARY KEY (CODE_LEVEL))
PARTITION BY LIST(PROD_COL)
(PARTITION PT1 VALUES(0) TABLESPACE HCTB,PARTITION PT2 VALUES(1) TABLESPACE HCTB,
PARTITION PT3 VALUES(2) TABLESPACE HCTB, PARTITION PT4 VALUES(3) TABLESPACE HCTB,
PARTITION PT5 VALUES(4) TABLESPACE HCTB, PARTITION PT6 VALUES(5) TABLESPACE HCTB,
PARTITION PT7 VALUES(6) TABLESPACE HCTB, PARTITION PT8 VALUES(7) TABLESPACE HCTB);
```

No materialized views and advanced indexes are created, except indexes on primary and foreign keys.

Figure 2.9 compares the *native referential partitioning* and *user driven referential partitioning*. In both cases, the fact table is partitioned into eight partitions based on *ProdLevel* table. In user driven referential partitioning, the *ProdLevel* is horizontally partitioned using the *hash mode* on the primary key (Code_Level) and the fact table is horizontally partitioned using also the *hash mode* on the foreign key (the number of partitions using hash must be a power of 2). Both partitioning are compared to the non partitioning case. The result shows that native horizontal partitioning

[2]For example, we can partition a table using three fragmentation attributes.

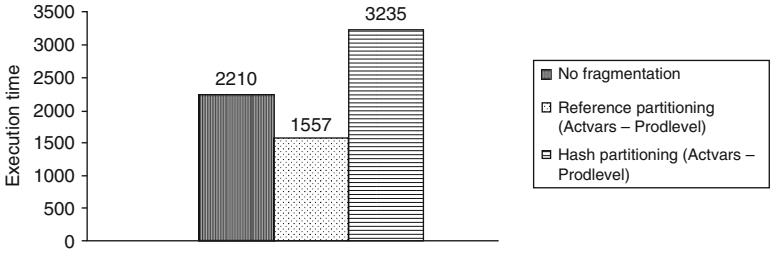


Fig. 2.9 Native and user driven referential partitioning comparison

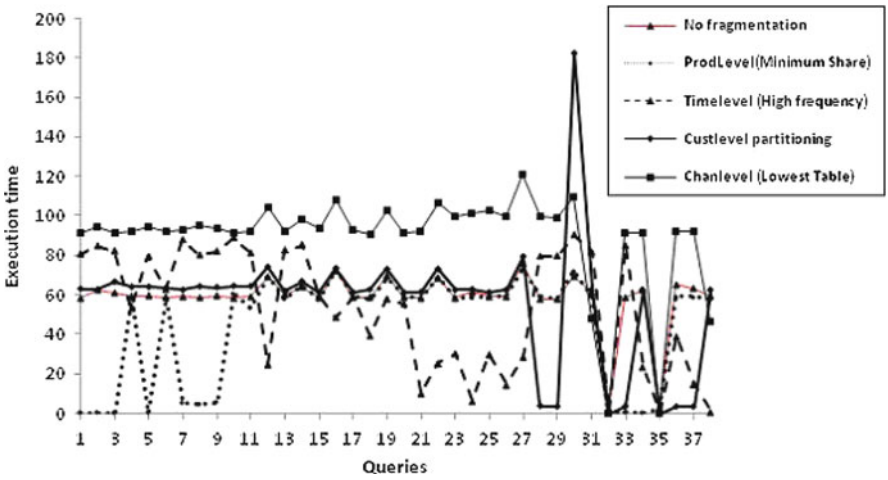


Fig. 2.10 Individual evaluation of impact of criterion on query

outperforms largely the two other cases. *The user driven referential partitioning is not well adapted to optimize star join queries, where the non partitioning mode outperforms it*.

Figure 2.10 compares the quality of solution generated by each criteria: *Minimum Share, Highest frequency* and *Lowest table*. These criteria are also compared with the non partitioned case. To conduct this experiment, we execute three times our climbing algorithm with a threshold equal to 10. Each query is executed individually on each generated fragmentation schema (one schema per selection criterion) and its execution time (given in second) is computed using *Enterprise manager of Oracle 11G*. Based on the obtained results, we propose to DBA the following recommendations:

- To improve performance of queries containing only one join and selection predicates involving only one dimension table, DBA has to partition the fact table based on that dimension table; without worrying about its characteristic (minimum share, highest frequency, etc.). More clearly, queries 1, 2, 5 optimized

with the *minimum share strategy* and queries 21, 22 and 23 with high frequency, etc. This case is very limited in real data warehouse applications, where queries involve all dimension tables.

- To speed up queries involving several dimension tables, but only one of them has selection predicates, DBA has to partition his/her fact table based on the dimension table involving selection predicates.
- To accelerate queries involving several dimension tables and each one contains selection predicates (this scenario is most representative in the real life), DBA has to partition his/her fact table based on the dimension table having a *minimum share*. *Query 34* of our query benchmark is an example of this scenario, where two dimension tables are used *ProdLevel* (with class_level = 'LB2RKO0ZQCJD) and *TimeLevel* (with two predicates Quarter_level = 'Q2' and Year_level = '1996'). Another example is about the *query 38 of our benchmark*, where two dimension tables are used, and each one has only one selection predicate. In this case, *TimeLevel* strategy (representing high frequency and also the table having a second minimum share) outperforms *ChanLevel* strategy representing the lowest table.

To complete our understanding, we use "*Set autotrace on*" utility to get execution plan of the query 34. First, we execute it on the non partitioned data warehouse and we generate its execution plan. We identify that Oracle optimizer starts by joining fact table with *ProdLevel* table (which has the minimum share criterion)[3]. Secondly, we execute the same query on a partitioned data warehouse obtained by fragmenting the fact table based on *ProdLevel*. The query optimizer starts by joining a partition of fact table with one of *ProdLevel* table and then does

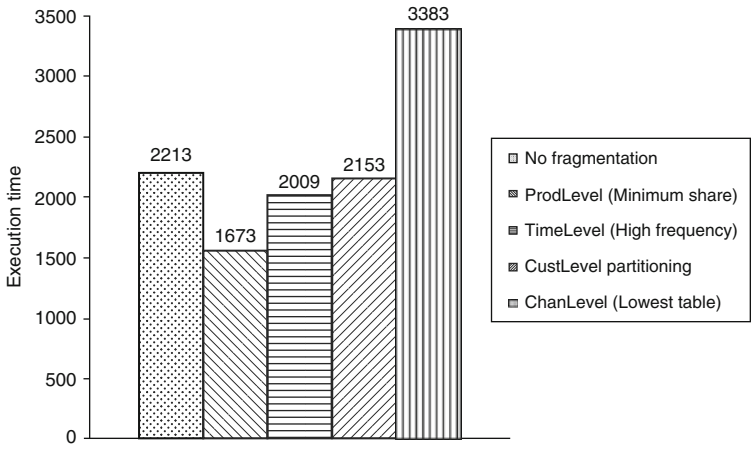


Fig. 2.11 Evaluation of all strategies for overall queries

[3]The Oracle cost-based optimizer recognizes star queries.

other operation. Based on these two execution plans, we can conclude that query optimizer uses the same criteria (minimum share) for ordering join. Referential partitioning gives a partial order of star join queries problem. Figure 2.11 summarizes the performance of each criterion for all queries.

2.5 Advisor Tool

To facilitate the use of our partitioning strategies of a given relational data warehouse, we develop a tool, called, *ParAdmin* offering nice graphical interfaces to DBA. It supports primary and referential partitioning. It may be connected to any DBMS. For our study, we choose Oracle, since it supports referential partitioning. Based on a workload of queries Q and the maintenance constraint representing the number of final fragments, it offers DBA several choices to simulate the partitioning. If they are satisfied, they can generate all the scripts necessary to partition the data warehouse to DBA. It is developed with under Visual C++. ParAdmin consists of a set of five modules: (1) *meta-base querying module*, (2) *managing queries module*, (3) *horizontal partitioning selection module (HPSM)* and (4) *horizontal partitioning module* and (5) *query rewriting module*.

The meta-base querying module is a very important module which allows the tool to work with any type of DBMS. From a type of DBMS, user name and password the module allows to connect to that account and to collect some information from the meta-base. These information concerns logical and physical levels of the data warehouse. Information of logical level includes tables and attributes in these tables. Information of physical level includes optimization techniques used and a set of statistics on tables and attributes of the data warehouse (number of tuples, cardinality, etc.) (Fig. 2.12).

The managing queries module allows DBA to define the workload of queries (Q) used to partition the data warehouse. An DBA has two possibilities to edit her/his queries: manually or by importing them from external files. This module offers DBA the possibility to add, delete, and update queries. It integrates a parser in order to check the syntax of queries and to identify errors (based on the used tables and attributes by each query).

Horizontal partitioning selection module (HPSM) selects partitioning schema of the data warehouse. It requires as inputs a schema of data warehouse, a workload and a threshold W. Using these data; HPSM selects a partitioning schema (PS) that minimizes the cost of the workload and that respect the maintenance constraint W.

Horizontal partitioning module fragments physically the data warehouse using partitioning schema obtained from HPSM if the DBA is satisfied with the proposed solution. It partitions dimension table(s) and propagates their partitioning to fact table.

Query rewriting module rewrites each global query on the generated fragmentation schema. It starts with identifying valid fragments for each query; rewrites the

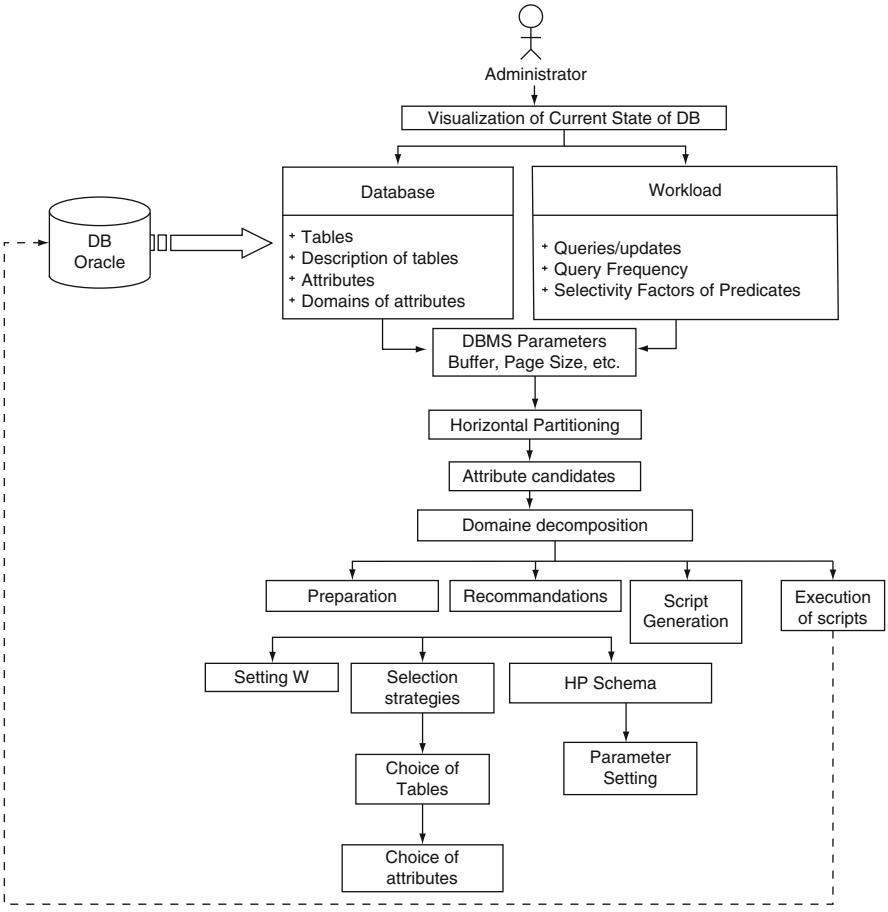


Fig. 2.12 Steps of our tool

global query on each relevant fragment. Finally, it performs union of the obtained results.

The main functionalities of ParAdmin are described in Fig. 2.12. They were identified incrementally by our Ph.D. students of our laboratory (LISI) following my advanced databases course.

1. The first functionality is to display the current state of the database (the schema, attributes, size of each table, definition of each attribute, etc.) and the workload (description of queries, number of selection operations, selection predicates, etc.). Figure 2.13 shows an example of visualisation proposed by our tool.
2. The second functionality consists in preparing the partitioning process, by offering different choices to select dimension table(s) (minimum share, largest table, most frequently table) to be used to partition the fact table, fragmentation attributes of the chosen table(s), number of final fragments, etc. (Fig. 2.14).

Fig. 2.13 Visualization of current state of the database

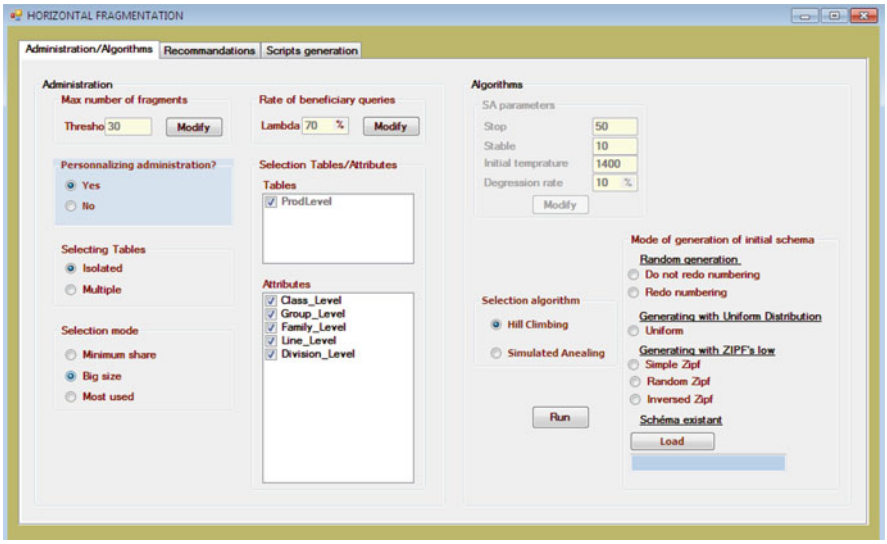
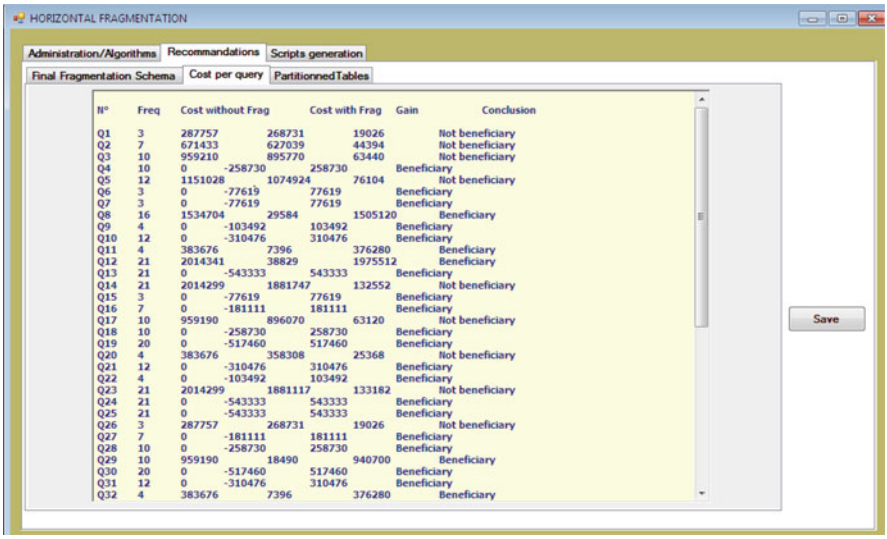


Fig. 2.14 Choice of dimension table(s)

3. Once the preparation phase done, DBA may run the partitioning algorithm that proposes recommendations. DBA has the possibility to observe the cost of query before and after partitioning. As consequence, she/he can decide on considering or not the proposed partitioning. Her/his decision is based on the gain offered by horizontal partitioning for each query established using a threshold (Fig. 2.15).



N°	Freq	Cost without Frag	Cost with Frag	Gain	Conclusion
Q1	3	287757	268731	19026	Not beneficiary
Q2	7	671433	627039	44394	Not beneficiary
Q3	10	959210	895770	63440	Not beneficiary
Q4	10	0	-258730	258730	Beneficiary
Q5	12	1151028	1074924	76104	Not beneficiary
Q6	3	0	-77619	77619	Beneficiary
Q7	3	0	-77619	77619	Beneficiary
Q8	16	1534704	29584	1505120	Beneficiary
Q9	4	0	-103492	103492	Beneficiary
Q10	12	0	-310476	310476	Beneficiary
Q11	4	383676	7396	376280	Beneficiary
Q12	21	2014341	38829	1975512	Beneficiary
Q13	21	0	-543333	543333	Beneficiary
Q14	21	2014299	1881747	132552	Not beneficiary
Q15	3	0	-77619	77619	Beneficiary
Q16	7	0	-181111	181111	Beneficiary
Q17	10	959190	896070	63120	Not beneficiary
Q18	10	0	-258730	258730	Beneficiary
Q19	20	0	-517460	517460	Beneficiary
Q20	4	383676	358308	25368	Not beneficiary
Q21	12	0	-310476	310476	Beneficiary
Q22	4	0	-103492	103492	Beneficiary
Q23	21	2014299	1881117	133182	Not beneficiary
Q24	21	0	-543333	543333	Beneficiary
Q25	21	0	-543333	543333	Beneficiary
Q26	3	287757	268731	19026	Not beneficiary
Q27	7	0	-181111	181111	Beneficiary
Q28	10	0	-258730	258730	Beneficiary
Q29	10	959190	18490	940700	Beneficiary
Q30	20	0	-517460	517460	Beneficiary
Q31	12	0	-310476	310476	Beneficiary
Q32	4	383676	7396	376280	Beneficiary

Fig. 2.15 Recommendations given to DBA

4. Finally, our tool proposes a functionality that generates scripts for primary and referential partitioning (if DBA is satisfied) corresponding to the target DBMS. They can be directly executed to really partition the warehouse.

2.6 Conclusion

Horizontal data partitioning is one of important aspects of physical design of advanced database systems. Two horizontal partitioning types are identified: *mono table partitioning* and *dependent-table partitioning*. The first one mainly used in traditional databases to optimize selections. The second one is well adapted for business intelligence applications in order to optimize selections and joins defined in mega queries involving a large number of tables. We propose a comprehensive procedure for using referential partitioning. It first selects relevant dimension table(s) to partition the fact table and discards some; even they are associated with selection predicates. To perform this selection, we propose three strategies based on three factors: *share*, *frequency* and *cardinality* of each dimension table. The share criterion reduces the size of intermediate results of joins; therefore, it may be used in cascade over dimension tables. Two types of experimental studies were conducted to validate our findings: one using a mathematical cost model and another using *Oracle11G* with the APB1 benchmark. The obtained results are encouraging and recommendations are given to assist administrators on using the referential partitioning. Our procedure can be easily incorporated in any DBMS supporting referential partitioning such as *Oracle11g*. A tool, called *ParAdmin* is presented

to assist DBAs to select the relevant dimension table(s) to partition the fact table. It gives recommendations to measure the quality of the fragmentation schemes proposed by our algorithm.

An interesting issue needs to be addressed is to extend this work to the problem of selecting bitmap join indexes, where dimension table(s) are candidate to define each index.

References

1. Bellatreche, L., Boukhalfa, K., Abdalla, H.I.: SAGA: A Combination of Genetic and Simulated Annealing Algorithms for Physical Data Warehouse Design. In: Proceedings of BNCOD'06, pp. 212–219 (2006)
2. Bellatreche, L., Boukhalfa, K., Richard, P.: Data Partitioning in Data Warehouses: Hardness Study, Heuristics and ORACLE Validation. In: Proceedings of DaWaK'2008, pp. 87–96 (2008)
3. Bellatreche, L., Boukhalfa, K., Richard, P., Woameno, K.Y.: Referential Horizontal Partitioning Selection Problem in Data Warehouses: Hardness Study and Selection Algorithms. In IJDWM. **5**(4), 1–23 (2009)
4. Bellatreche, L., Karlapalem, K., Simonet. A.: Algorithms and Support for Horizontal Class Partitioning in Object-Oriented Databases. In the Distributed and Parallel Databases Journal, **8**(2), 155–179 (2000)
5. Bellatreche, L., Woameno, K.Y.: Dimension Table Driven Approach to Referential Partition Relational Data Warehouses. In: ACM 12th International Workshop on Data Warehousing and OLAP (DOLAP), pp. 9–16 (2009)
6. Boukhalfa, K.: De la Conception Physique aux Outils d'Administration et de Tuning des Entrepts de Donnes. Poitiers University, France, PhD. Thesis. (2009)
7. Ceri, S., Negri, M., Pelagatti, G.: Horizontal Data Partitioning in Database Design. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. SIGPLAN Notices, pp. 128–136 (1982)
8. Cho, W.S., Park, C.M., Whang, K.Y., So, S.H.: A New Method for estimating the number of objects satisfying an object-oriented query involving partial participation of classes. Inf. Syst. **21**(3), 253–267 (1996)
9. Eadon, G., Chong, E.I., Shankar, S., Raghavan, A., Srinivasan, J., Das, S.: Supporting Table Partitioning By Reference in Oracle. In: Proceedings of SIGMOD'08, pp. 1111–1122 (2008)
10. Furtado, P.: Experimental evidence on partitioning in parallel data warehouses. In: Proceedings Of DOLAP, pp. 23–30 (2004)
11. Gibbons, P.B., Matias, Y., Poosala, V.: Fast incremental maintenance of approximate histograms. ACM Trans. Database Syst. **27**(3), 261–298 (2002)
12. Gray, J., Slutz, D.: Data Mining the SDSS SkyServer Database. Microsoft Research, Technical Report MSR-TR-2002-01 (2002)
13. Karlapalem, K., Navathe, S.B., Ammar, M.: Optimal Redesign Policies to Support Dynamic Processing of Applications on a Distributed Database System. Information Systems, **21**(4), 353–367 (1996)
14. Lei, H., Ross, K.A.: Faster Joins, Self-Joins and Multi-Way Joins Using Join Indices. In Data and Knowledge Engineering, **28**(3), 277–298 (1998)
15. Legler, T., Lehner, W., Ross, A.: Query Optimization For Data Warehouse System With Different Data Distribution Strategies, In BTW, pp. 502–513 (2007)
16. Mahboubi, H., Darmont, J.: Data mining-based fragmentation of XML data warehouses. In: Proceedings DOLAP'08, pp. 9–16 (2008)
17. Munneke, D., Wahlstrom, K., Mohania, M.K.: Fragmentation of Multidimensional Databases. In: Proceedings of ADC'99 pp. 153–164 (1999)

18. Neumann, T.: Query simplification: graceful degradation for join-order optimization. In: Proceedings of SIGMOD'09, pp. 403–414 (2009)
19. Noaman, A.Y., Barker, K.: A Horizontal Fragmentation Algorithm for the Fact Relation in a Distributed Data Warehouse. In: Proceedings of CIKM'99, pp. 154–161 (1999)
20. Oracle Data Sheet: Oracle Partitioning (2007) White Paper: http://www.oracle.com/technology/products/bi/db/11g
21. OLAP Council: APB-1 OLAP Benchmark, Release II. http://www.olapcouncil.org/research/bmarkly.htm (1998)
22. Özsu, M.T., Valduriez, P.: Principles of Distributed Database Systems, Second Ed. Prentice Hall (1999)
23. Sanjay, A., Narasayya, V.R., Yang, B.: Integrating Vertical and Horizontal Partitioning Into Automated Physical Database Design. In: Proceedings of SIGMOD'04, pp. 359–370 (2004)
24. Simon, E.: Reality check: a case study of an EII research prototype encountering customer needs. In Proceedings of EDBT'08, pp. 1 (2008)
25. Stöhr, T., Märtens, H., Rahm, E.: Multi-Dimensional Database Allocation for Parallel Data Warehouses. In: Proceedings of VLDB2000, pp. 273–284 (2000)
26. Steinbrunn, M., Moerkotte, G., Kemper, A.: Heuristic and Randomized Optimization for the Join Ordering Problem. In VLDB Journal. **6**(3), 191–208 (1997)
27. Swami, A.N., Schiefer, K.B.: On the Estimation of Join Result Sizes. In: Proceedings of EDBT'04, pp. 287–300 (1994)
28. Sybase: Sybase Adaptive Server Enterprise 15 Data Partitioning. White paper (2005)

http://www.springer.com/978-3-7091-0737-9

Recent Trends in Information Reuse and Integration
Özyer, T.; Kianmehr, K.; Tan, M. (Eds.)
2012, XVI, 400 p., Hardcover
ISBN: 978-3-7091-0737-9