

Forgetting data intelligently in data warehouses

Aliou Boly, Georges Hébrail

Laboratoire LTCI - UMR 5141 CNRS
Ecole Nationale Supérieure des Télécommunications
Paris, France
boly@enst.fr, hebrail@enst.fr

Aliou Boly, Sabine Goutier

Research and Development
Electricité de France
Clamart, France
sabine.goutier@edf.fr

Abstract— The amount of data stored in data warehouses grows very quickly so that they can get saturated. To overcome this problem, we propose a language for specifying forgetting functions on stored data. In order to preserve the possibility of performing interesting analyses of historical data, the specifications include the definition of some summaries of deleted data. These summaries are aggregates and samples of deleted data and will be kept in the data warehouse. Once forgetting functions have been specified, the data warehouse is automatically updated in order to follow the specifications. This paper presents both the language for specifications, the structure of the summaries and the algorithms to update the data warehouse.

Keywords- *Aggregation, Data cubes, Data warehouses, Forgetting functions*

I. INTRODUCTION

Although the purpose of data warehouses is to store historical data, they can get saturated after a few years of operation. To overcome this problem, the solution is generally to archive older data when new data arrive if cost of maintaining older data cannot be justified economically. This solution is not satisfactory because analyses based on long term historical data become impossible. As a matter of fact analysis of data cannot be done on archived data without re-loading them in the data warehouse; and the cost of loading back a large dataset of archived data is too high to be operated just for one analysis. The archiving of data makes them hard to manage and query efficiently. So, archived data must be considered as lost data in a business intelligence perspective.

In this paper, we propose a solution for solving this problem: a language is defined to specify forgetting functions on older data. Two main ideas are developed: (1) the specifications define what data should be present in the data warehouse at each step of time, so that differences between actual contents of the data warehouse and the specifications are candidate data to be archived, (2) some summaries of archived data are kept in the data warehouse so that many analyses can still be done by using these summaries. The summaries are of two types: aggregates and samples on older data. Aggregation and sampling are two standard and complementary ways to summarize data. Consider for example a data warehouse with click-stream data capturing user behaviour on web sites. As time passes, older detailed

data tend to become useless and can be replaced in data mining algorithms by aggregated data by day or by month. In complement to aggregated data, some samples either picked up randomly or chosen to be of particular interest, can be used to perform some analyses or just useful to explain the results of analyses. As for the summaries corresponding to aggregates of detailed data, there are defined to be more and more aggregated as they concern older data: for instance, sums of sales in a sales data warehouse can be specified to be aggregated at a daily level for data between one and three months old, then at a monthly level for data between three months and two years old, and finally at a yearly level for data older than two years. As for samples, their size is defined as a fixed size: consequently there are less and less older detailed data in each sample if the sampling is performed correctly. The specification language for forgetting function is defined in the context of relational databases: one specification can be defined for each table in the database. Once specifications have been defined, some algorithms which are presented in this paper automatically update the data warehouse according to the specifications and fill-up the corresponding summaries. Note that these algorithms can be run at any time by the database administrator. Our motivation in this work is to deal with the problem of data warehouse saturation but all the described results are applicable to relational databases supporting OLTP systems. All these algorithms have been studied and programmed in a prototype upon the ORACLE system.

The paper is organized as follows. Section II is devoted to the presentation of related work. In Section III, we present the specification language we have defined, after a formal definition of the ‘age’ of data in a data warehouse and the presentation of a motivating example. In Section IV, we show that the aggregate summaries we define can be stored in data cubes and present the algorithms for applying forgetting functions. Section V treats the conservation of samples. Section VI is a conclusion to this work and draws some perspectives.

II. RELATED WORK

Our work is first related to ‘vacuuming’ which is an approach to physical deletion [22]. The concept of vacuuming is developed by Jensen in [17] in the context of transaction-time databases: data that is older than a certain time should be

considered as inaccessible and can be removed. In the TSQL2 temporal query language [10, 24], when a particular date is specified, only data that is prior to the date should be physically deleted [17].

Our work is also related to work on data reduction as described in [23] where a technique is presented for data reduction that handles a gradual change of data from new detailed data to older and summarized data in a data warehouse. Our work offers comparable facilities but is applicable to relational databases instead of multidimensional databases. Moreover, there is no conservation of detailed data in their work compared to ours.

More recently, much work has been done around the concept of data streams where there is a strong need for summarizing data on a temporal perspective. For example in [4, 8, 25, 32, 33], the problem of computing temporal aggregates over data streams is examined and it is suggested to maintain aggregates at multiple levels of granularities depending on time: older data are aggregated using coarser granularities while more recent data are aggregated with finer detail. In our work, we also develop a model for such a feature but the way data are aggregated is specified by the data warehouse administrator instead of being controlled by the stream.

The specifications we define for forgetting data control how older data should expire and thus be deleted from the data warehouse. In the context of materialized views (see [9, 15, 16, 19, 26]) work has been done on data expiration: in [13] it is suggested to expire (delete) unneeded materialized view tuples, so that a set of predefined views on these materialized views can be still maintained with future updates. The motivation in this work is that data warehouses collect data into materialized views for analysis and as time evolves, the materialized views occupy too much space and some of the data may no longer be of interest. This approach cannot be applied to solve the problem we address in this paper.

Still related to data expiration is the work described in [28]: the problem addressed in this paper is to study data expiration in the context of historical databases which store the history of the different states of the database. This work is not relevant here since we are not interested in the history of database states but in historical data explicitly stored in a data warehouse.

When applying forgetting functions, some tuples will be deleted from the data warehouse, possibly linked by some referential constraints. This is related to the concept of garbage-collector [5, 21], which is a form of automatic memory management. The principle is to determine what data objects in a program will not be accessed in the future and reclaim the storage used by those objects. In particular a garbage-collector must analyze relationships (links) between objects to be deleted: we show in Section 5 that a similar problem appears when tuples from different tables are linked by some foreign key constraints (see [1, 12, 14, 29]).

Compared with the related work cited above, our main

contribution is the definition of a language for specifying such forgetting functions and keeping of summaries of archived data. These specifications – defined by the database administrator – enable to automate the process of applying the forgetting functions, which includes the deletion of data to be archived (possibly linked with foreign key constraints) and the update of summaries which are aggregates and samples of detailed data.

In [3, 6, 7, 9, 11, 18], it is shown that many queries can be answered using only samples or aggregated data instead of the whole database. On the one hand, when aggregate summaries are available instead of the detailed version of data, it is still possible to answer aggregation queries with a result provided as an approximation of the exact answer. On the other hand, samples can be used to infer characteristics of a whole dataset by using sampling theory as in surveys (see [2, 27]). Moreover many data mining algorithms – such as decision trees for instance – can only consider as input aggregates over detailed data. This justifies our approach to keep summaries of archived data: a complete discussion of this aspect is beyond the scope of this paper.

III. SPECIFICATION OF FORGETTING FUNCTIONS

The specifications we define are based on the notion of ‘age’ of data stored in a data warehouse. This notion is defined formally in the next sub-section.

A. Notion of age of data

As mentioned in the introduction, our study is done in the context of relational databases. We consider that detailed data are tuples stored in relations of the data warehouse. We assume here that each tuple is associated with a date denoted t_s , which corresponds either to the value of an attribute of the relation or to a system timestamp representing the date of the last update of the tuple. The *age* of a tuple data calculated at current date t_c is defined to be the difference $t_c - t_s$. Both t_s and t_c can be expressed at different time unit levels defined to be: SECOND, MINUTE, HOUR, DAY, MONTH, QUARTER and YEAR.

Durations like the age $t_c - t_s$ can also be defined in different units, by convention we take:

$$\begin{aligned} 1 \text{ MINUTE} &= 60 \text{ s[seconds]}, \\ 1 \text{ HOUR} &= 60 * 60 \text{ s}, \\ 1 \text{ DAY} &= 24 * 3600 \text{ s} \\ 1 \text{ MONTH} &= 30 \text{ DAYS} = 30 * 24 * 3600 \text{ s}, \\ 1 \text{ QUARTER} &= 3 * 30 \text{ DAYS} = 3 * 30 * 24 * 3600 \text{ s} \\ 1 \text{ YEAR} &= 365 \text{ DAYS} = 365 * 24 * 3600 \text{ s} \end{aligned}$$

In order to compute the age $t_c - t_s$, we first transform t_c and t_s into the SECOND time level unit whatever the time unit level used for each of them. This is done by taking the first instant of the period covered by the date if it is not expressed at the SECOND level.

For instance: $to_second('17/02/06 \ 12:05')$ ='17/02/06 12:05:00', $to_second('February 2006')$ ='01/02/06 00:00:00'.

The age is first computed at the SECOND time unit as follows:

- if t_s is expressed at the SECOND level:
age = $\text{to_second}(t_c) - t_s$
- if t_s is not expressed at the SECOND level:
age = $\text{to_second}(t_c) - \text{to_second}(t_s+1)$

Secondly the age can be converted at a higher level using the conventions defined above, using the *floor* function.

- In minutes: $\text{floor}(\text{age} / 60)$,
- In hours: $\text{floor}(\text{age} / 3600)$,
- In days: $\text{floor}(\text{age} / (24 * 3600))$,
- In months: $\text{floor}(\text{age} / (30 * 24 * 3600))$,
- In quarters: $\text{floor}(\text{age} / (3 * 30 * 24 * 3600))$,
- In years: $\text{floor}(\text{age} / (365 * 30 * 24 * 3600))$

For instance: an age expressed in the DAY unit is obtained by $\text{floor}(\text{age} / (24 * 3600))$.

This process ensures that the duration between the last instant of the period covered by t_s (for instance the period is one day if it is expressed at the DAY time level), and the first instant of the period covered by t_c is at least the age expressed in its unit (for instance 2 months if the age has been converted to the MONTH unit).

Some examples are given :

$t_c = '20/01/06\ 12:34:45'$, $t_s = '20/01/06\ 10:50:54'$, $t_c - t_s = 6231\ \text{SECOND} = 1\ \text{HOUR}$

$t_c = '20/02/06'$, $t_s = '20/01/06\ 11h'$, $\text{to_second}(t_c) = '20/02/06\ 00:00:00'$, $\text{to_second}(t_s+1) = '20/01/06\ 12:00:00'$, $t_c - t_s = 2635200\ \text{SECOND} = 30\ \text{DAY} = 1\ \text{MONTH}$.

Finally, an ordering can be defined between ages:

We say $\text{age}_1 < \text{age}_2$ if $\text{in_second}(\text{age}_1) < \text{in_second}(\text{age}_2)$ where *in_second* is a function which transforms into seconds a duration expressed in other duration time units, using the conversion conventions described above. For example: $30\ \text{DAY} < 3\ \text{MONTH} < 1\ \text{YEAR}$.

B. Motivating example

We present here a motivating example designed from a real CRM (Customer Relationship Management) data warehouse at Electricité de France (EDF). Let CLIENT and ORDER be the following relations (tables) of the data warehouse:

CLIENT(clientId, City, Department, Region, Sex, Salary),
ORDER(orderID, Date_order, amount, clientId)

The first (underlined) attribute of each relation is its primary key. The CLIENT relation contains the location, sex and salary of each client. The ORDER relation contains the date, amount and client of each order. We assume that the following referential integrity constraint holds: from ORDER.clientID to CLIENT.clientID. Figure 1 presents a specification of forgetting function for the ORDER relation.

```
SUMMARY TABLE Order {
USE CLIENT JOIN BY clientId;
TIMESTAMP = Date_order;
HIERARCHY(Geography):City→Department→Region;
LESS THAN 1 MONTH : DETAIL;
LESS THAN 3 MONTH : SUM (amount) BY CLIENT.City, CLIENT.Sex, DAY;
LESS THAN 1 YEAR : SUM (amount) BY CLIENT.Department, MONTH;
LESS THAN 5 YEAR : SUM (amount) BY CLIENT.Region, YEAR;
LESS THAN 10 YEAR : SUM (amount) BY YEAR;
KEEP SAMPLE (1000) WHERE amount>4000;
}
END SUMMARY ;
```

Figure 1. Motivating example

This forgetting function shows first that the date used to compute the age of each Order tuple is the value provided by the Date_order column (TIMESTAMP = Date_order).

The ‘LESS THAN’ specifications define when detailed data (tuples from the ORDER relation) should be archived and which aggregates should be kept depending on the age of archived tuples. Tuples having an age less than 1 MONTH must be kept in the ORDER relation. Tuples older than 1 MONTH *can* be archived but some aggregates must be kept by the system:

- Tuples with an age between 1 and 3 months must be at least described with aggregates by City, Sex and DAY. Notice that City and Sex are attributes of the CLIENT relation. Aggregation by City and Sex is possible by using the specification “USE CLIENT JOIN BY clientId”.
- Tuples with an age between 3 months and 1 year must be at least described with aggregates by Department and MONTH. Aggregation by Department is possible by using the ‘Geography’ hierarchy¹.
- Tuples with an age between 1 year and 5 years must be at least described with aggregates by Region and YEAR.
- Tuples with an age between 5 years and 10 years must be at least described with aggregates by YEAR.
- Tuples with an age greater than 10 years can be completely forgotten.

Some semantic constraints in the language ensure that the different ‘LESS THAN’ specifications are consistent: ‘LESS THAN’ specifications with older age are associated with aggregates which can be obtained by a roll-up operation from ‘LESS THAN’ specifications with younger age. Finally, the “KEEP SAMPLE (1000) WHERE amount>4000” specification is a specification for keeping a random sample of archived tuples. It says to keep a random sample of 1000 tuples from the archived ORDER tuples which verify the condition “amount>4000”.

C. Specification language for forgetting functions

A language has been defined to specify a forgetting function associated with each table in a relational database. The

¹ We suppose that there exists a hierarchy named Geography between attributes city, Department and Region of the CLIENT relation.

grammar of this language is given in appendix. It includes additional features, such as the discretization of numerical attributes to use them as aggregation attributes, which are not described in this paper.

The focus in this paper is on the management of forgetting functions. Once some specifications have been defined, the application of forgetting functions is automatic: some algorithms and storage structures are needed to manage the forgetting process. The two next sections respectively study the maintenance of aggregates and the maintenance of detailed data (archiving of detailed data and update of samples).

IV. MANAGEMENT OF FORGETTING FUNCTIONS BY AGGREGATION

A. Storage structure for aggregate summaries

As shown in the example of Section 3.2, a forgetting function contains several ‘LESS THAN’ aggregation specifications. Each of these specifications indicates an aggregation level depending on the age of data. From these ‘LESS THAN’ specifications, a cube scheme is derived and will be used to store aggregates as time passes.

A cube scheme C is a list (D_1, \dots, D_n, TL, M) , where D_1, \dots, D_n are dimension names, TL is the time dimension and M is a vector space of measures. With each dimension D_i is associated an *ordered* list of levels $L_i = \text{levels}(D_i)$, including the value *ALL* aggregating completely the dimension². These levels among dimensions represent hierarchies: we assume there is only one hierarchy per dimension. For our example, we have:

$C_ORDER = (\text{LOCATION}, \text{SEX}, \text{TL}, \text{TOTAL_AMOUNT})$
 $\text{levels}(\text{LOCATION}) = (\text{CITY}, \text{DEPARTMENT}, \text{REGION}, \text{ALL})$, $\text{levels}(\text{SEX}) = (\text{SEX}, \text{ALL})$, $\text{levels}(\text{TL}) = (\text{DAY}, \text{MONTH}, \text{YEAR}, \text{ALL})$

Then, using the cube scheme, we associate one cube of the scheme (also called *cubeoid* in some references) with each ‘LESS THAN’ aggregation specification.

A cube C of a cube scheme C is defined by (l_1, \dots, l_n, tl, M) such that $l_i \in L_i = \text{levels}(D_i)$, $i = 1, \dots, n$, $tl \in TL$. With each level l is associated a non-empty set of values $\text{dom}(l)$, representing the domain of l ; with each time level tl is associated $\text{dom}(tl)$, representing the domain of tl .

A cell c of a cube C is a tuple $c = (x_1, \dots, x_n, t, m)$, where $\forall i = 1, \dots, n$, $x_i \in \text{dom}(l_i)$, $t \in \text{dom}(tl)$ and $m \in M$.

This gives the following cubes for our example:

S_1 : LESS THAN 3 MONTH : SUM (amount) BY City, Sex, DAY ;

$\rightarrow C_1 = (\text{CITY}, \text{SEX}, \text{DAY}, \text{SUM_AMOUNT})$

S_2 : LESS THAN 1 YEAR : SUM (amount) BY Department, MONTH ;

$\rightarrow C_2 = (\text{DEPARTMENT}, \text{ALL}, \text{MONTH}, \text{SUM_AMOUNT})$

S_3 : LESS THAN 5 YEAR : SUM (amount) BY Region, YEAR ;

² For the time dimension, we generally define $\text{levels}(TL)$ as (SECOND, MINUTE, HOUR, DAY, MONTH, QUARTER, YEAR, ALL)

$\rightarrow C_3 = (\text{REGION}, \text{ALL}, \text{YEAR}, \text{SUM_AMOUNT})$

S_4 : LESS THAN 10 YEAR : SUM (amount) BY YEAR ;

$\rightarrow C_4 = (\text{ALL}, \text{ALL}, \text{YEAR}, \text{SUM_AMOUNT})$

The different ‘LESS THAN’ specifications can be ordered by age³ (ex: age of ‘LESS THAN 3 MONTH’ is 3 MONTH): $\text{age}(S_1) \leq \text{age}(S_2) \leq \text{age}(S_3) \leq \text{age}(S_4)$.

The cubes can also be ordered from the less to the most aggregated one (we use the same \leq notation): $C_1 \leq C_2 \leq C_3 \leq C_4$. Note that $C_1 \leq C_2$ means that the partition of detailed tuples induced by cube C_1 is finer than the one induced by C_2 ⁴.

B. Update of aggregate summaries

The algorithms developed for updating aggregate summaries are based on three properties:

- 1) Additivity of aggregation functions,
- 2) Disjointness of cubes,
- 3) Commutation of updates of aggregates and detailed data.

1) Additivity of aggregation functions: This property is often assumed when building cubes. In our context, this property ensures that data belonging to a cube C_2 can be computed from a cube C_1 if $C_1 \leq C_2$. In our example, the SUM aggregation function on measure amount is additive. The grammar of the language only accepts additive aggregation functions or functions which can be derived from additive ones (for instance AVG can be derived from SUM and COUNT). Note that since AVG is computed in terms of SUM and COUNT, if a specification includes an AVG function then we assume it also includes the corresponding functions for SUM and COUNT. We do not further consider AVG separately in this paper.

2) Disjointness of data cubes: We have seen in the previous section that ‘LESS THAN’ specifications can be ordered by their age. Let us consider two specifications S_{i-1} and S_i associated with cubes C_{i-1} and C_i respectively. We have $\text{age}(S_{i-1}) \leq \text{age}(S_i)$. A tuple d from the base relation (ORDER in our example) is assumed to be counted in only one cube, defined as follows:

$d \in C_i$ iff $\text{age}(S_{i-1}) \leq \text{age}(d) < \text{age}(S_i)$.

The consequence of this assumption is that all cubes of a forgetting function are disjoint: no cell of a cube is included (functionally) in a cell of a more aggregated cube. The different cubes of a forgetting function store disjoint periods over time.

3) Commutation of updates of aggregates and detailed data: The consequence of the preceding assumption and property is that updates issued from the application of forgetting functions are of only two types:

³ The age of a ‘LESS THAN’ specification is defined by the age value specified just after the ‘THAN’ keyword.

⁴ This is a total order because there is only one hierarchy per dimension.

- detailed tuples from the base relation are archived and must be counted in one (unique) cube,
- already aggregated data may be transferred from one cube to a more aggregated one.

Since detailed data are only counted in one cube, it is equivalent to first update the cubes with newly archived data and then operate the transfers between cubes, or to first operate the transfers and then counting newly archived data. We assume here that updates due to forgetting functions can be applied at any time, either evenly (for instance every day) or irregularly or even sporadically. At each time the update program is launched, new tuples of the base relation may satisfy some conditions of aggregation specifications: they must be archived and included into the corresponding cubes (which depend on when the update program is invoked). Also, aggregated data of some cubes may be transferred to another cube if they satisfy the criteria of the corresponding specification. Our algorithm operates with a bounded total space necessary to store the different cubes. This is achieved by the disjointness property and by an activation/inactivation process for the cube cells. When data should be transferred to a data cube, the cells where they were stored are not deleted but inactivated. So, when data should be transferred to a data cube, we first check if they may occupy some inactivated cells of this cube, and if so, these cells are activated.

As a consequence of the commutation property, we distinguish two update procedures which can be invoked in any order:

- (1) the procedure which transfers cells between cubes (figure 2),
- (2) the procedure which includes newly archived data in the cubes (figure 3).

In the presented algorithms, we consider that each cell in a cube is characterized by two fields: the position and the measure. The cell's position is the set of values of corresponding dimension levels that determine this cell. For example, position of cell ('Paris', 'F', 'Jan 01, 06', 2000) is ('Paris', 'F', 'Jan 01, 06') and the corresponding measure is equal to 2000.

In algorithm presented in figure 2, it is important to note that it is possible that cells of a cube C_i may not satisfy the criteria of the following cube C_{i+1} but a cube C_p such that $p > i+1$. For example, this may happen if the forgetting function has not been applied for a long period. Considering the example presented in Section 3.2, when the forgetting function has not been applied for five years, the data cells of the cube corresponding to the second specification⁵ should be directly transferred to the most aggregated cube. This explains why there are two nested loops for cubes in the algorithm.

⁵ LESS THAN 3 MONTH : SUM(amount) BY City, Sex, DAY;

```

for each cube  $C_j$  ( $j = n, n-1, n-2, \dots, 2$ )
  for each cube  $C_i$  ( $i = j-1, j-2, \dots, 1$ )
    for each cell data  $c = (x_1, \dots, x_p, t, m)$  in  $C_i$  with
       $age(S_j) \geq age(t) > age(S_i)$ 
      /*  $S_j$  and  $S_i$  the specifications associated with cubes  $C_j$  and  $C_i$ 
      respectively */
      /* search is accelerated by an index on the time dimension */
      let  $\bar{c}$  the cell in cube  $C_j$  covering  $c$ 
      if ( $\bar{c}$  is found)
        /* incrementation */
        for each aggregate measure  $M_i$  in  $\bar{c}$ 
          if  $M_i$  is COUNT or SUM
             $\bar{c}.M_i = \bar{c}.M_i + c.M_i$ 
          else if  $M_i$  is MIN
             $\bar{c}.M_i = \text{MIN}(\bar{c}.M_i, c.M_i)$ 
          else if  $M_i$  is MAX
             $\bar{c}.M_i = \text{MAX}(\bar{c}.M_i, c.M_i)$ 
        else
          /* check if there is an inactivated cell data in  $C_j$  that can replace  $\bar{c}$  */
          if (found)
             $C_{found}.position := \bar{c}.position$ 
             $C_{found}.measure := \bar{c}.measure$ 
          else
            create cell data  $\bar{c}$  into  $C_j$ 
            inactivate cell data  $c$  in the cube  $C_i$ 

```

Figure 2. Algorithm Transfer of data between cubes

We now present the algorithm for the inclusion of tuples from the base relation to the cubes. Notice that the order of processing of data cubes is not important, each tuple of detail belonging to only one cube.

```

for each cube  $C_j$  ( $j = 1, 2, \dots, n$ )
  let  $C_j^{active} = \{d \in R \text{ (base relation) with } age(S_j) \geq age(d) > age(S_{j-1})$ 
  and aggregated at levels  $C_j\}$ 
  /* we suppose there is an index on the time dimension;  $S_j$  and  $S_{j-1}$ 
  are the specifications associated with cubes  $C_j$  and  $C_{j-1}$  respectively;
   $S_0$  corresponding to the specification of DETAIL6 */
  for each cell data  $c$  in  $C_j^{active}$ 
    let cell data  $c' =$  cell data of the cube  $C_j$  having the same values
    for dimensions of  $C_j$ 
    if ( $c'$  is found)
      /* incrementation */
      for each aggregate measure  $M_i$  in  $c'$ 
        if  $M_i$  is COUNT or SUM
           $c'.M_i = c'.M_i + c.M_i$ 
        else if  $M_i$  is MIN
           $c'.M_i = \text{MIN}(c'.M_i, c.M_i)$ 
        else if  $M_i$  is MAX
           $c'.M_i = \text{MAX}(c'.M_i, c.M_i)$ 
        else
          /* check if there is an inactivated cell data in  $C_j$  that can replace
           $\bar{c}$  */
          if (found)
             $C_{found}.position := c'.position$ 
             $C_{found}.measure := c'.measure$ 
          else
            create cell data  $c'$  into  $C_j$ 

```

Figure 3. Algorithm for inclusion of tuples from the base relation to cubes

C. Management of referential constraints

In the preceding sections, we have assumed that only one

base relation is associated with a forgetting specification. Our approach enables the definition of several specifications, each of them being associated with one relation. The problem which arises when defining several forgetting specifications is that tuples from base relations may be linked together by foreign keys. For instance, consider in our example, a relation *BILL*(*billID*, *Date_bill*, *amount*, *orderID*) with the following referential integrity constraint: from *BILL.billID* to *ORDER.billID*. Let us assume that the timestamp of the forgetting function associated to the *BILL* relation to calculate the age of each *Bill* tuple is provided by the *Date_bill* column and the following specifications are defined:

```
LESS THAN 4 MONTH: DETAIL;
LESS THAN 6 MONTH: SUM(amount) BY MONTH;
```

The problem appears when a tuple t_1 to be archived in a base relation (*ORDER* in our example) is referenced by a tuple in another relation (*BILL* in our example), say t_2 which is not yet archived. In our example, the *Order* tuples can be specified to be archived but the corresponding *Bill* tuples are not yet archived. This may happen when forgetting functions are applied every month (for instance) since *Bill* tuples having an age less than 4 MONTH must be kept in the *BILL* relation while the *Order* tuples older than 1 MONTH must be specified to be archived. We propose the following solution in this case: t_1 is not archived but marked to be archived and will be checked to be archived in later executions of the forgetting function. So, the *Order* tuples in our example are marked to be archived and they will be archived when the corresponding *Bill* tuples are archived. Note that no further update of the tuple will be allowed.

V. CONSERVATION OF SAMPLED DETAILED DATA

As seen in the example of Section 3.2, the specification language enables to specify to keep samples of archived data (thus deleted from the base relations). This is the *KEEP SAMPLE* specification which indicates that a random sample of fixed size is maintained whenever detailed tuples are archived. Such samples are stored in a separate table which has the same structure as the base relation, with some additional attributes for the management of forgetting functions.

At every application of the forgetting functions, new tuples may be archived and should be considered to update the sample. For example, we suppose that a sample of 1000 individuals is kept among tuples to archive and we have 100 new tuples to archive since the last update of forgetting functions.

For maintaining the sample, it is necessary to use an incremental technique: we use a reservoir sampling algorithm due to Waterman (see Vitter [30]). This approach enables to sample data ‘on the fly’, without knowing in advance the number of individuals in the whole population. It operates by decreasing over time the probability of picking up a tuple.

Notice that the size of samples is bounded by the definition of the *KEEP SAMPLE* specifications.

As indicated at the end of Section 2, random samples from a population can be used to infer information about a whole population by only observing values available in a sample of it (see [2, 27]). For instance, it is possible to infer the sum of amounts of orders by measuring it from a sample of orders and then calibrate this measure. Such estimations (and corresponding confidence intervals) can only be done if the size of the whole population is known. So it is necessary to store the number of archived tuples corresponding to each sample. This is achieved by always defining a *COUNT* measure in the storage structure of aggregate summaries (or to define one if no aggregate summary has been specified).

VI. CONCLUSION AND PERSPECTIVES

We have proposed a solution for dealing with the saturation problem in data warehouses and more generally in relational databases. A language has been defined for specifying a policy on how to archive data in data warehouses and keep summaries of archived data. These summaries are of two types: aggregates and samples. These summaries are designed to occupy a bounded amount of space in the database. Once forgetting specifications have been defined, the algorithms presented in the paper update automatically both the data warehouse and the summaries. A prototype system, based on the *ORACLE* database software, has been developed to prove the concept. Aggregates and samples are known to contain enough information to perform many data mining analysis and to answer queries in an approximate way. Our current work focuses on the exploitation of the summaries combining aggregates and samples.

APPENDIX

The key words are in majuscule, constructions with ‘[‘ and ‘]’ are facultative. The symbol ‘|’ means other possible form. The elements whose name begins by *id* denote an alphanumeric sequence of characters.

```
<forgetting function> ::=
    SUMMARY TABLE <table_name> {
        <list_use_linked_table>;
        <specif_timestamp>;
        <specif_discretise>;
        <specif_hierarchy>;
        <specif_detail>;
        <list_specif_aggregation>;
        <specif_KEEP>;
    }
END SUMMARY ;
```

```

<table_name> ::= idAtt | QUARTER
| YEAR

<list_use_linked_table> ::= <use_linked_table>
| <list_use_linked_table> ',' <use_linked_table>

<use_linked_table> ::= /* empty */
| USE <table_name> JOIN BY <columns>

<specif_timestamp> ::= TIMESTAMP = SYSTEM
| <column_name> ;

<columns> ::= <column_name>
| <columns> ',' <column_name>

<column_name> ::= idAtt

<specif_discretise> ::= /* empty */
| <discretisation>
| <specif_discretise> ',' <discretisation>

<discretisation> ::=
DISCRETISE ('(<column_name>') =
idAtt: ('(<interval> idAtt;') + ')

<interval> ::= ([[]] <numeric value>, <numeric value> ([[]])
<numeric value> ::= ([+] | -) <number>
<number> ::= <chiffre>
| <number> <chiffre>
<chiffre> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<specif_hierarchy> ::= /* empty */
| <hierarchy>
| <specif_hierarchy> ',' <hierarchy>

<hierarchy> ::= HIERARCHY ('(<dimension name>') ' '
'(<levels hierarchy>')
<dimension name> ::= idAtt
<levels hierarchy> ::= <level hierarchy>
| <levels hierarchy> '→' <level
hierarchy>
<level hierarchy> ::= idAtt

<specif_detail> ::= /* empty */
| <criteria> ':' DETAIL

<list_specif_aggregation> ::= <specif_aggregation>
| <list_specif_aggregation> ','
<specif_aggregation>

<specif_aggregation> ::= <criteria> ':' <aggregation data>
<criteria> ::= LESS THAN <value> <time level>
<value> ::= <chiffre>
| <value> <chiffre>
<time level> ::= SECOND
| MINUTE
| HOUR
| DAY
| MONTH

```

```

<aggregation data> ::= <measures> <list BY>
<measures> ::= <aggregation>
| <measures> ',' <aggregation>

<aggregation> ::= COUNT (' '*' ')
| <AGG> ('(<column_name>')

<AGG> ::= COUNT
| SUM
| MIN
| MAX
| AVG

<list BY> ::= /* empty */
| BY <list levels>
<list levels> ::= <level>
| <list levels> ',' <level>

<level> ::= [<table_name> '.' ] <column_name>
| <name_discretise>
| <time level>

<specif_KEEP> ::= /* empty */
| KEEP SAMPLE ('(<value>') [WHERE
<condition>]
<condition> ::= <expression>
| <condition> <op logic>
<expression> ::= [<table_name> '.' ] <column_name> <op>
| <simple expression>
<op> ::= <|> | ≥ | ≤ | = | <
<simple expression> ::= idAtt
| <numeric value>
<op logic> ::= OR | AND

```

REFERENCES

- [1] Akoka J., Comyn-Wattiau, Conception des bases de données relationnelles en pratique, Vuibert, Ed. France, 2003.
- [2] Ardilly P., Les Techniques de sondage, Technip, Ed. France, 1994.
- [3] Babcock B., Chaudhuri S., Das G., Dynamic Sample Selection for Approximate Query Processing, SIGMOD 2003, June 9-12, San Diego, CA.
- [4] Babcock B., Datar M., Motwani R., and Widom J. Models and issues in data stream systems. In Proceedings 2002 ACM Symposium Principles of Database Systems (PODS'02), pages 1-16, Madison, WI, June 2002.
- [5] Boehm H., Bounding Space Usage of Conservative Garbage Collectors, Proceedings of the 2002 ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages, pages 93-100, January 2002.
- [6] Chaudhuri S., Das G., Srivastava: Effective Use of Block-Level Sampling in Statistics Estimation. Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 287-298, Paris, France, June 13-18, 2004.
- [7] Chaudhuri S., Das G., Motwani R., Narasayya V., A Robust Optimisation-Based Approach for Approximate Answering of

- Aggregate Queries. International Conference on Management of Data, Proceedings of the 2002 ACM SIGMOD, pages 295-306, Santa Barbara, California, United States, 2001.
- [8] Chen Y., Dong G., Han J., Wah B. W., and Wang J., Multi-dimensional regression analysis of time-series data streams. In Proc. 2002 Int. Conf. Very Large Data Bases (VLDB'02), pages 323-334, Hong Kong, China, Aug. 2002.
- [9] Chirkova R., Li C., Materializing Views with Minimal Size to Answer Queries, Principles of Database Systems 2003, June 9-12, 2003, San Diego, CA.
- [10] Dumas M., Fauvet M.-C., Scholl P.-C., Chapter V "Temporels Models". In "Bases de Données et Internet", G. Jomier, A. Doucet (eds.), Hermès Science publications, 2002, ISBN 2-7462-0283-2.
- [11] Ganti V., Lee M.L., Ramakrishnan R., ICICLES : self-tuning samples for Approximate Query Answering. Proc. Of VLDB, 2000.
- [12] Garcia-Molina H., Ullman J., and Widom J., Database Systems: the Complete Book, Prentice Hall; 1st edition, 2001.
- [13] Garcia-Molina H., Labio W. J., Yang J., Expiring data in a warehouse, Proceedings of the 24th VLDB Conference, pages 500-511, New York, USA, 1998.
- [14] Gardarin G., Bases de Données objet et relationnel, Eyrolles 1999.
- [15] Gupta H., Mumick I., Selection of Views to Materialize in a Data Warehouse. In IEEE Transactions on Knowledge and Data Engineering, TKDE, volume 17 (1/2005), pages 24-43, 2005.
- [16] Gupta A., Mumick I.S., Maintenance of Materialized Views: Problems, Techniques, and Applications, IEEE Data Engineering Bull. 18(2): pages 3-18, 1995.
- [17] Jensen C. S., "Vacuuming", in the TSQL2 Temporal Query Language, R. T. Snodgrass, editor, Chapter 23, pp. 451-462, Kluwer Academic Publishers, 1995.
- [18] Jin R., Glimcher L., Jermaine C., Agrawal G., New Sampling-Based Estimators for OLAP Queries, Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA.
- [19] Paraboschi S., Sindoni G., Baralis E., Teniente E., Materialized Views in Multidimensional Databases. Maurizio Rafanelli (Ed.): Multidimensional Databases: Problems and Solutions, pages 222-251. Idea Group 2003.
- [20] Ramakrishnan R., and Gehrke J., Database Management Systems, Mc Graw-Hill, third edition, 2003.
- [21] Serrano M., Boehm H., "Understanding Memory Allocation of Scheme Programs", Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming, pages 245-256, Montreal, Canada, 2000.
- [22] Skyt J., Jensen C. S., and Mark L., A Foundation for Vacuuming Temporal, Data and Knowledge Engineering, volume 44, issue(1), January 2003.
- [23] Skyt J., Jensen C.S., Pedersen T. B., Specification-Based Data Reduction in Dimensional Data Warehouses, TimeCenter TechReport TR-61, 2001.
- [24] Snodgrass R. T., The TSQL2 Temporal Query Language, Kluwer Academic Publishers, 1995.
- [25] Tatbul N., Zdonik S., Window-Aware Load Shedding for Aggregation Queries over Data Streams, Proceedings of the 32nd International Conference on Very Large Data Bases, September 12-15, 2006, Seoul, Korea.
- [26] Theodoratos D., Xu W., Constructing search spaces for materialized view selection. In Proceedings ACM Seventh International Workshop on Data Warehousing and OLAP, DOLAP, pages 112-121, 2004.
- [27] Tillé Y., Théorie des sondages, Dunod, Ed., France, 2001.
- [28] Toman D., Expiration of Historical Databases. Proceedings of the Eighth International Symposium on Temporal Representation and Reasoning (TIME'01): 128-135, IEEE Press, 2001.
- [29] Ullman J.D., Principles of Databases and Knowledge Base Systems, volume 1 and 2. Computer Science Press, 1989.
- [30] Vitter J. S., Random Sampling with a Reservoir. ACM Transactions on Mathematical Software, 11(1): pages 37-57, March 1985.
- [31] Widom J.. Special issue on materialized views and data warehousing. IEEE Bull. on Data Engineering, 18(2), 1995.
- [32] Zhang D., Gunopulos D., Tsotras V. J., and Seeger B., "Temporal and Spatio-Temporal Aggregations over Data Systems, vol. 28, no. 1-2, pages 61-84, 2003. Streams using Multiple Time Granularities", Journal of Information.
- [33] Zhang D., Gunopulos D., Tsotras V. J., Seeger B., Temporal Aggregation over Data Streams Using Multiple Granularities, Proceedings of the 8th International Conference on Extending Database Technology: Advances in Database Technology, pages 646-663, March 25-27, 2002.