A Predicate-based Incremental Refresh Method for a Data Warehouse

Dongmei Ren, Guogen Zhang, William Perrizo

555 Bailey Avenue
IBM Silicon Valley Lab
San Jose, CA 95141, USA
408-463-5972
dongme@us.ibm.com

IACC 258C
Computer Science
North Dakota State University
Fargo, ND 58105
william.perrizo@ndsu.edu

Abstract

With the increasing popularity of data warehouses and data marts, the ability to refresh data in a timely fashion is more important than ever. In this paper, a new approach is proposed to incrementally update materialized summary tables. The advantage of the approach is that a) this method retrieves changes on a predicate level; b) the method utilizes time information hidden in user data. Therefore, the method does not introduce extra cost while the other current approaches do introduce overhead for time tag; c) our method applies changes to summary tables by merge. The merge only updates changed values and inserting new rows into summary table, and it is much more efficient than union. By making use of the above advantages, this method is much faster than other methods, and the same accurateness as state-of-the-art approaches can be achieved. By comparing this method with the state-of-the-art over IBM DB2 workstation version, it showed that the proposed method outperforms the state-of-the-art in terms of elapse time.

1 INTRODUCTION

In a data warehouse context, operational data are integrated into data warehouse hourly, daily, weekly or monthly, as depends on application type. The updating and synchronization of operational database unit and data warehouse unit are very important to guarantee that a correct query answer is achieved with most current operational data[1][2][3].

As it is known, pre-calculation is one of the approaches to achieve fast query response. For efficient data warehouse query purpose, data are pre-processed, and some intermediate results, such as sum, average, and stand deviation, are stored in data warehouse. In this paper, summary tables are used to store those pre-computed and summary values. The query can be answered against summary tables containing pre-computed information/summary information. Since summary tables are much, much smaller than original base tables, as exposes much more opportunities to achieve a quick query response.

The updating and synchronization of summary table and base table is an important issue to assure a data warehouse work correctly with most current data. This paper introduces a new method to incrementally update summary tables as base tables change. The method works in two steps: firstly, incremental updates are acquired by distribute summarization over a fact table and dimensional tables; secondly, a summary table is updated with the changes through a merge procedure. Merge is a new efficient alternative to union all. Instead of combining rows from two or more tables, merge updates the existing rows in a summary table with latest column values, while inserting new rows if the updated data contain rows NOT in the original summary table. Merge saves cost to a large degree when the updates are mostly against the existing rows.

In comparison with other existing methods, this method has the following advantages: 1). the method retrieves changes through a query with WHERE clause. The retrieval is based on a predicate level query; 2). the method does not introduce extra overhead cost of time tag; 3). Merge applies the detected changes from base tables into summary tables. The above three features make our approach for an incremental refreshment efficient.

The rest of the paper is organized as follows. In section 2, the related work about refreshments of a data warehouse, especially for an incremental refreshment, is reviewed. The proposed new method is discussed in section 3. The experimental proof that our method outperforms the state-of-the-art is covered in section 4. Conclusions are drawn in section 5.

2 RELATED WORK

With the increasing popularity of data warehouses and data marts, the ability to refresh data in a timely fashion is more important than ever. The approaches to refresh data warehouse fall in two broad categories: static updates and...
incremental updates. Static capture, timestamp capture, file comparison capture are static capture methods mainly discussed currently; there are three main approaches for incremental updates, which are application-assisted capture, trigger-based capture, and transaction log capture[1][2][3][4][5].

Static data capture is usually associated with taking a snapshot of the data at a particular point in time. In some cases, the complete data sets may be retrieved, but in most cases, only a subset will be used. Generally speaking, the static capture methods are relatively simple, but suffer from capturing historical data because it is difficult to capture intermediate states of data. Taking timestamp capture method as an example, if a record changed state five times since its last capture and the operational system did not maintain periodic history for the record, then the next time the information capture application would only retrieve the current state of the record, while the other four states have been lost.

Incremental data capture is a time-dependent model for capturing changes to operational systems. This approach is closely tied to database management system (DBMS), and is best applied in the situation where changes in the data are significantly smaller than the size of the data set for a capturing cycle. Figure 1 shows the basic idea of incremental refresh. In Figure 1, the table in the center represents the fact table, and four small tables around the fact table are dimensional tables, each for one dimension. The table on the right side is a summary table with summarized information. Compared with static updates, an incremental refreshment is more complex, but more accurate. An incremental refreshment is the focus of this paper. We analyze the aforementioned three incremental refreshment methods next.

In this method, the invocation of the trigger can be used to capture intermediate states of data. Taking trigger capture method as an example, if a record changed state five times since its last capture and the operational system did not maintain periodic history for the record, then the next time the information capture application would only retrieve the current state of the record, while the other four states have been lost.

Incremental data capture is a time-dependent model for capturing changes to operational systems. This approach is closely tied to database management system (DBMS), and is best applied in the situation where changes in the data are significantly smaller than the size of the data set for a capturing cycle. Figure 1 shows the basic idea of incremental refresh. In Figure 1, the table in the center represents the fact table, and four small tables around the fact table are dimensional tables, each for one dimension. The table on the right side is a summary table with summarized information. Compared with static updates, an incremental refreshment is more complex, but more accurate. An incremental refreshment is the focus of this paper. We analyze the aforementioned three incremental refreshment methods next.

The main challenge of incrementally updating data warehouse is to identify changes within the operational systems that are of interest to a data warehouse at an appropriate granularity. Once these changes have been identified, the problem then becomes how to determine the actions to take based on these changes. In a data warehouse environment, different actions may be taken based on the types of change identified (e.g. insert, update or delete). For example, if we have created a retail data warehouse, and over the course of time, one of our products is retired, we would capture that change (a delete) and update the information in our warehouse correspondingly. In this particular case, we would most likely have the product information in a dimension table and the subsequent action would be to update the product dimension record with a termination date.

In this paper, we propose a new method to detect changes. Based on the captured changes, the data warehouse is refreshed by a merge process that is much more efficient than union all. The propose method is
efficient in terms of both capturing changes and writing changes to a data warehouse. The performance efficiency is shown experimentally.

3 THE PORPOSED METHOD

In this section, a predicate-based refreshment method for a data warehouse is proposed. This method belongs to the category of incremental refreshment. The basic idea of an incremental refreshment is to apply changes in base tables to summary tables. The problem of an incremental refreshment can be divided into two sub-problems: one is how to detect changes to base tables, i.e. determine \( \delta \); and the second problem is how to apply \( \delta \) into summary tables, thus keeping data in summary tables up to date. Our new approach will be discussed with regarding to the two sub-problems.

In this approach, firstly, a novel method is introduced to determine a change \( \delta \). Unlike other existing methods that need some overhead such as a time stamp, a page number, and a slot number, to remember the time spot of last change, our method uses time information from user data, thus eliminating overheads of time tags; secondly, the changes, \( \delta \), is retrieved through a query which accomplishes the retrieval on a predicate level; lastly, our method applies \( \delta \) by merging. The merging process overwrites exiting rows in a summary table with latest value, while inserts new rows if a row comes in the summary table for the first time during a capture. The merge process saves much more cost than union all, especially when the number of new rows is small.

An example data warehouse is given, shown in figure 2, to help explain our method. In the example, the data warehouse has a star schema. The fact table is a Sales facts object table (Sales), and three dimensions tables are a Product dimension table, (Product), a Market dimension table (Market) and a Time dimension table (Time) correspondingly. The queries in the rest of this paper are based on this exemplary data warehouse.

This section includes three sub-sections. Sub-section 3.1 introduces a definition of a summary table, a definition of data update, and a definition of \( \delta \)-retrieval query; sub-section 3.2 describes the calculation of integrating an incremental change, \( \delta \), which is the theory basis of merging; sub-section 3.3 summaries the proposed approach.

3.1 Definitions and Notations

In this subsection, we introduce several definitions to ease explanation of our algorithm [2][4].

Definition 1 : Definition of a Summary Table

Similar to a regular view, the content of a summary table is defined by a SELECT expression. A definition of a summary table for the data warehouse in figure 2 can be

```
CREATE SUMMARY TABLE ast_demo AS
(SELECT  TimeID, StoreID, ProduceID,
sum(sales), sum(totalExp), sum(profit),
COUNT(*) AS count,
FROM Sales
GROUP BY storeID, ProduceID,
TimeID,
)
DATA INITIALLY DEFERRED REFRESH IMMEDIATE;
```

The above expression defines a three-dimensional hierarchical data cube for Product, Market, and Time. From the query, we can see that a summary table may contain an explicit specification of its physical layout similar to a regular base table, i.e. it can be partitioned, indexed. Also, a refreshment mode “REFRESH IMMEDIATE” is assigned to a summary table. The incremental update strategy will be discussed in a definition of a \( \delta \)-Retrieval Query.

Definition 2 : Definition of Data Update

A data update denotes a table of changes of the extent of one of the information sources, i.e. base tables. It could be tuples to be added to that information source, or tuples to be dropped form that base table, denoted by positive or negative counters respectively. We say that the set of data updates at information source j at time m, denoted by DU
An insertion (delete) set, if DU (m)[j] contains only tuples that have been added to (deleted from) IS[j].

**Definition 3: Definition of δ-Retrieval Query**

In our method, the change is retrieved by a SELECT query. We define a SELECT query which can retrieve the changes between time spots LOWlimit and UPlimit, as a retrieval query with parameter LOWlimit and UPlimit denoted as RQ (LOWlimit, UPlimit). We assume that, both LOWlimit and UPlimit as a time factor, denoted as TF, are valid at those two spot, i.e. $0 \leq TF(LOWlimit) \leq \infty$ holds and $0 \leq TF(UPlimit) \leq \infty$ holds.

For the summary table in definition 1, the corresponding RQ (LOWlimit, UPlimit) is defined as:

```
SELECT TimeID, StoreID, ProduceID, 
sum(sales), sum(totalExp), sum(profit), 
COUNT(*) AS count, 
FROM Sales 
WHERE TimeID > LOWlimit AND TimeID < UPlimit 
GROUP BY storeID, ProduceID, TimeID, 
```

### 3.2. Calculation of the Incremental Refresh

![Figure 3. An Illustration of Applying Changes to a Summary Table](image)

In this subsection, we will describe how to integrate $\delta$ after capturing it. Our approach is developed from the following calculations. It is assumed that $S_{UPlimit}$ denotes the data in a summary table at time TF(UPlimit), $S_{LOWlimit}$ denotes data in a summary table at time TF(LOWlimit), $P_{LOWlimit}$ represents data in a fact table at spot TF (LOWlimit); $P_{LOWlimit}$ denotes data in a dimension table. In the calculation, we only use one dimensional table for simplicity. However, the calculation can be generalized to multiple dimensional tables; $\cup_A$ denotes union all, $\delta$ means changes to base tables, $\delta_S$ is the summary of changes, and $A \bowtie \delta$ denotes a join operator [7].

The data in summary table at TF(LOWlimit) can be represented by

$$S_{LOWlimit} = \sum_{(gc, sum)} (F_{LOWlimit} \bowtie \delta_F) \bowtie \delta_P_{LOWlimit} \quad \text{--- formula 1}$$

where gc and sum are columns of tables, then data in a summary table at TF(UPlimit) can be denoted as

$$S_{UPlimit} = \sum_{(gc, sum)} (F_{LOWlimit} \bowtie \delta_F) \bowtie \delta_P_{LOWlimit} \quad \text{--- formula 2}$$

Therefore, data in a summary table at TF(UPlimit) can be calculated by

$$S_{UPlimit} = \sum_{(gc, sum)} (S_{LOWlimit} \cup_A \delta_S) \quad \text{--- formula 3}$$

where

$$\delta_S = \sum_{(gc, sum)} (\delta_F \cup_A \delta_P_{LOWlimit}) .$$

Formula 3 is the theory basis for a merge process. The integration of $\delta$ is implemented by a merge process. The process updates the existing rows in a summary table with the latest values, and it inserts new rows if the updated data contain rows NOT in the summary table during a capture time. The following tiny example in figure 4 helps to illustrate the merge process. In the example, the red rows in an original table, S, and the red row in change capture table, $\delta$, are aggregated (using summary in this example), and the value in the merged table is $1.0 + 2.5 = 3.5$; the green row is inserted into the merged S table as a new row; and the blue row keeps unchanged in table S before and after merging.

![Figure 4. An Example of Merging Process](image)
3.3. The Proposed Algorithm

This section describes the proposed predicate-based incremental refreshment method for a data warehouse formally. Given two time spots, TF(LOWlim), and TF(UPlim), the algorithm works in two steps.

Step 1: Capture $\delta$ between $\text{TF}(\text{LOWlim})$ and $\text{TF}(\text{UPlim})$ by a $\delta$-retrieval query, $\text{RQ}(\text{UPlim}, \text{LOWlim})$, with a WHERE clause against base tables. The WHERE predicts is $\text{LOWlim} \leq \text{TF} \leq \text{UPlim}$.  

Step 2: Apply $\delta$ to the summary table through the MERGE process. We will use the aforementioned data warehouse to further explain step 2. For the data warehouse in figure 2, MERGE can be accomplished by the following query:

MERGE INTO SumaryTable S
USING (SELECT TimeID, StoreID, ProduceID, SUM(sales), SUM(totalExp), SUM(profit), COUNT(*) AS count,
FROM Sales
WHERE TimeID > LOWlim AND TimeID < UPlim
GROUP BY storeID, ProduceID, TimeID) D
ON (TimeID > LOWlim AND TimeID < UPlim)
WHEN MATCHED THEN UPDATE SET sum(sales),
sum(totalExp), sum(profit)
WHEN NOT MATCHED THEN INSERT (S.StoreID, S.ProduceID, S.sum(sales), S.sum(totalExp),
S.sum(profit))
VALUES (D.StoreID, D.ProduceID, D.sum(sales),
D.sum(totalExp), D.sum(profit))

In fact, the USING expression is just the $\delta$-retrieval query. Therefore, the whole process is accomplished through a MERGE query.

As we can see that our approach to incremental refresh is neat. Also, the functionality can be either inside a data warehouse engine, or a user can write such a query to accomplish the refreshment.

The experimental examination in section 4 will show that our method is much faster than the state-of-the-art, the transaction log capture approach.

4 EXPERIMENTAL DEMONSTRATION

In this section, some experiments are designed to compare our methods with the transaction log capture approach [3]. Our purpose is to show our approach’s efficiency in terms of elapse time. The framework of implementing this predicate-based incremental refreshment process consists of the followings:

Determine the time column granularity in the fact table and the refresh frequency requirement: The time column could be at day level, minute level, second level, or even finer levels. The refreshment frequency typically cannot be finer than the time column level. For example, if the time column is at the day level, refresh frequency will be at daily level after the data for a whole day is loaded or inserted. If the time column is at the minute level, refreshment can be done twice a day, or hourly, or every 30 minutes, which depends on the data input frequency and their volume. If data is loaded in batch, refreshment can be followed with each load.

Determine the time point value used as the upper limit in the predicate: the framework consistently uses greater than or equal to ($\geq$) for the lower limit and less than ($<$) for the upper limit. To guarantee this scheme work well, all the data fall into time spots which are less than the upper limit should be loaded or inserted already. For example, daily refreshment should happen after today’s data are loaded, and the date for tomorrow is used as the upper limit. For hourly refreshment after 12:00pm, it should be performed with all the data falling into time spots before 12:00pm. If the refreshment is part of every load, the upper limit can be the time for the last load.

Once the refreshment time point is given, incremental refreshment can be performed automatically based on the algorithm described above. The language support for this is a new clause for the SQL REFRESH TABLE statement. For example, REFRESH TABLE mySumTable WITH DATA BEFORE ‘2006-5-20:12:00’, where mySumTable is a summary table, ‘2006-5-20:12:00’ is $\text{Uplim}$, and $\text{Lowlim}$ is equal to $\text{Uplim}$ of the immediate previous refreshment.

As for delayed data after refreshment point, one solution is to use trickle insert, and to have immediate refreshment to cover the data. Since this kind of data tends to be very small in amount, immediate refreshment will not cause performance impact to the system. The immediate refresh can be triggered by the time column with a value that is smaller than the lower limit of the batch refreshment time.

The two aforementioned approaches were developed in the framework, and were tested over IBM DB2 workstation version 8 installed on an IBM client/workstation with 2.00GHz Intel Processor, 2GB main memory, running a Windows XP 2.00 [1]. The data were generated and the incremental changes to a data warehouse were simulated. An original summary table size is 2 gage bytes, and 1 mega bytes data were inserted into a fact table, dimension tables in batch but at random time spots. The experimental results show that the proposed method outperforms the transaction log capture method, the most optimal one so far, by around six times in terms of average elapse time.
5 CONCLUSION AND FUTURE WORK

In this paper, a new method is proposed for an incremental refreshment of a data warehouse. The proposed method is neat and much faster than the most favored approach, the transaction log capture method. The method can be implemented inside a database engine; also user can write a query to accomplish the refreshment, as provides flexibility for maintenance of a data warehouse.

Encouraged by the results in this paper, we will take exploring incremental updates approaches when a schema of a data warehouse changes as our further work.

6 REFERENCES


