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@ibm.us.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, we propose a new approach to incrementally update materialized summary tables. The advantage of our approach is that a) our method retrieves changes on a predicate level; b) the method utilizes time information hidden in user data, therefore, our 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. Merge only updates changed values and inserting new rows into summary table, and it is much efficient than union. By taking use of the above advantages, our method is much fast than other methods and achieves the same accurateness as state-of-the-art approaches. We compared our method with the state-of-the-art over IBM DB2 workstation version; the results showed that the proposed method outperforms the state-of-the-art in terms of elapse time.

1 INTRODUCTION

In data warehouse context, operational data are integrated into data warehouse hourly, daily, weekly or monthly, as depends on application type. The update 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 as well, 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, we use summary tables 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, we have much more opportunities to achieve a quick query response.

The update and synchronization of summary table and base table is an important issue to assure a data warehouse works 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: first, 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 for union all. Instead of combine rows from two or more tables, merge updates the existing rows in summary table with latest column values, while inserting new rows if the updated data contain rows NOT in an original summary table. Merge saves cost to a large degree when the updates are mostly against the existing rows.

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

The rest of the paper is organized as follows. In section 2, we review the related work for refresh of data warehouse, especially for incremental refresh. We propose our new method in section 3. Section 4 gives the experimental proof that our method outperforms the state-of-the-art; the paper is concluded 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 update 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][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 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, 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. Compared with static updates, incremental refresh is more complex, but more accurate than static capture. Incremental refresh is the exploring focus of this paper. We analyze the aforementioned three incremental methods next.

Application-Assisted Capture: The underlying principle of the method is that when changes are written to the operational database, changes are also written to a persistent area for later retrieval. The advantage of the method is that the changed records are available for immediate use since the latency is minimal; however, the method adds high cost to application, because, in order to write complete records to the changed queue, the application has to first go to the database and retrieve the entire record, then applies changes, and finally writes it out to the change queue.

Trigger-based Capture: This method assumes that the DBMS supports triggers, which are stored procedures that are invoked when certain conditions or events occur. In this method, the invocation of the trigger can be used to save changed records of interests to a persistent storage area for retrieval. The advantage of this approach is that both the before and the after image are available since the capture is conducted at the source of data. There are two drawbacks to this method. One is that file-based storage can not use this method; the other problem is that it requires the granularity of the information in target database matches that of the source database.

Transaction Log Capture: This approach utilizes log to capture changed information. In this method, only an application is written to monitor the log files and captures the data of interests. The cost added to DBMS is small; therefore, the method can capture changes without directly impeding the performance of the database. One disadvantage of the method is that the transaction logs must remain available until the changes of interest are captured. For example, if a database administrator (DBA) trims the transaction logs prior to the capture of the changed records, information will be lost. Compared to other methods, transaction log capture is the most efficient one because it introduces small overhead. In circumstances where DBMS are used to store operational data, transaction log capture is the most powerful and provides the most efficient approach to incremental capture.

The main challenge for incrementally updating data warehouse is to identify changes within the operational systems that are of interest to the data warehouse at the appropriate granularity. Once these changes have been identified, the problem then becomes one of determining the actions to take based on these changes. In a data warehouse environment, different actions may be taken based on the type of change identified (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, and based on the captured changes, the data warehouse is refreshed by a merge process that is much efficient than union all. The propose method are efficient in terms of both capturing changes and writing changes to a data warehouse. The performance efficiency is shown experimentally.
3 THE PROPOSED METHOD

In this section, we propose a predicate-based refresh method for a data warehouse. Our method falls into category of incremental refreshment. The basic idea of incremental refresh is to apply changes in base tables to summary tables. The problem of incremental refresh can be divided into two sub-problems: one is how to detect changes to base tables, i.e. determine δ; and the second problem is how to apply δ into summary tables, thus keeping data in summary tables up to date. We will describe our new approach with regarding to the two sub-problems.

In our approach, firstly, we introduce a new method to determine change δ. Unlike current other methods which 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 for time tag; secondly, the changes, δ, is retrieved through a query which accomplishes the retrieval on a predicate level; lastly, our method applies δ by merging. The merging process overwrites exiting rows in a summary table with latest value, while inserts new rows if a row comes for the first time during a capture. The merge process saves much cost than union all, especially when the number of new rows is small.

![Figure 2. An Example of Data Warehouse with Star Schema](image)

We give an example data warehouse 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 δ-retrieval query; sub-section 3.2 describes the calculation of integrating an incremental change, δ, which is the theory basis for merge; sub-section 3.3 summaries the proposed approach.

3.1 Definitions and Notations

In this subsection, we introduce several definitions to help 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

```sql
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 refresh mode “REFRESH IMMEDIATE” is assigned to a summary table. The incremental update strategy will be discussed in a definition of δ-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 from that 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 (m)[j], is 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 \( \text{Low}_{\text{limit}} \) and \( \text{Up}_{\text{limit}} \), as a retrieval query with parameter \( \text{Low}_{\text{limit}} \) and \( \text{Up}_{\text{limit}} \), denoted as \( \text{RQ}(\text{Low}_{\text{limit}}, \text{Up}_{\text{limit}}) \). We assume that both \( \text{Low}_{\text{limit}} \) and \( \text{Up}_{\text{limit}} \) make time factor, denoted as \( \text{TF} \), valid at those two spots, i.e.

\[
0 \leq \text{TF}(\text{Low}_{\text{limit}}) \leq \infty \quad \text{and} \quad 0 \leq \text{TF}(\text{Up}_{\text{limit}}) \leq \infty.
\]

For the summary table in definition 1, the corresponding \( \text{RQ}(\text{Low}_{\text{limit}}, \text{Up}_{\text{limit}}) \) is defined as:

SELECT TimeID, StoreID, ProduceID, sum(sales), sum(totalExp), sum(profit), COUNT(*) AS count, FROM Sales WHERE TimeID > \( \text{Low}_{\text{limit}} \) AND TimeID < \( \text{Up}_{\text{limit}} \) GROUP BY storeID, ProduceID, TimeID,

### 3.2. Calculation of the Incremental Refresh

In this subsection, we will describe how to integrate \( \delta \) after capturing it. Our approach is developed from the following calculations.

It is assume that \( S_{\text{Up}_{\text{limit}}} \) denotes the data in a summary table at time \( \text{TF}(\text{Up}_{\text{limit}}) \), \( S_{\text{Low}_{\text{limit}}} \) denotes data in a summary table at time \( \text{TF}(\text{Low}_{\text{limit}}) \), \( F_{\text{Low}_{\text{limit}}} \) represents data in a fact table at spot \( \text{TF}(\text{Low}_{\text{limit}}) \), \( P_{\text{Low}_{\text{limit}}} \) denotes a dimension table. In the calculation, we only use one dimensional table for simplicity. However, the calculation can be generalized to multiple dimensional tables; and \( U_\Lambda \) denotes union all, \( \delta \) means changes to base tables, \( \delta S \) is the summary of changes, and \( \bowtie \bowtie \) denotes a join operator [7].

The data in summary table at \( \text{TF}(\text{Low}_{\text{limit}}) \) can be represented by

\[
S_{\text{Low}_{\text{limit}}} = \sum_{(gc,\text{sum})} (F_{\text{Low}_{\text{limit}}} \bowtie \bowtie P_{\text{Low}_{\text{limit}}}) \quad \text{--- formula 1}
\]

where \( gc \) and \( \text{sum} \) are columns of tables, then data in a summary table at \( \text{TF}(\text{Up}_{\text{limit}}) \) can be denoted as

\[
S_{\text{Up}_{\text{limit}}} = \sum_{(gc,\text{sum})} (F_{\text{Low}_{\text{limit}}} U_\delta \bowtie \bowtie P_{\text{Low}_{\text{limit}}}) \quad \text{--- formula 2}
\]

Therefore, data in a summary table at \( \text{TF}(\text{Up}_{\text{limit}}) \) can be calculated by

\[
S_{\text{Up}_{\text{limit}}} = \sum_{(gc,\text{sum})} (S_{\text{Low}_{\text{limit}}} \bowtie \bowtie \delta S)
\]

where

\[
\delta S = \sum_{(gc,\text{sum})} (\delta F U_\delta \bowtie \bowtie P_{\text{Low}_{\text{limit}}})
\]

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 insert new rows if the updated data contain rows NOT in the summary table. The following tiny example in figure 4 helps to illustrate the merge process. In the example, the red rows in original table \( S \) and the red row in \( \delta \) are aggregated (using summary in this example), and the value in merged table is 1.0 + 2.5 = 3.5; the green row is inserted into merged \( S \) table as a new row; and the blue row keep unchanged in table \( S \) after merge.

**Figure 3. An Illustration of Applying Changes to Summary Table**

**Figure 4. An Example of Merge**
3.3. The Proposed Algorithm

This section describes the proposed predicate-based incremental refresh method for a data warehouse formally. Given two time spots, TF(LOW \_limit), and TF(UP \_limit), the algorithm works in two steps.

Step 1: Capture \( \delta \) between TF(LOW \_limit) and TF(UP \_limit) by a \( \delta \)-retrieval query, RQ(UP \_limit, LOW \_limit), with a WHERE clause against base tables. The WHERE predicts is LOW \_limit\( \leq \)TF \( \leq \)UP \_limit.

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 SummaryTable S
USING (SELECT TimeID, StoreID, ProduceID,
    sum(sales), sum(totalExp), sum(profit),
    COUNT(*) AS count,
    FROM Sales
    WHERE TimeID > LOW \_limit AND TimeID<UP \_limit
    GROUP BY storeID, ProduceID, TimeID) D
ON (TimeID > LOW \_limit AND TimeID<UP \_limit)
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 database engine, or a user can write a query to accomplish the refresh, which provides flexibility for maintenance of a data warehouse.

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, we design experiments 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.

We tested the two approaches over IBM DB2 workstation version 8 installed on an IBM client/workstation with 2.00GHz Intel Processor, 2GB main memory, running Windows XP 2.00 [1]. We generated data and simulated the incremental changes to a data warehouse. 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, which is the most optimal one so far, by 6.7 times in terms of elapse time.

5 CONCLUSION AND FUTURE WORK

In this paper, we propose a new method for incremental refresh for a data warehouse. Our method is neat and is much faster than the most favored approach, the transaction log capture method. Our method can be implemented inside a database engine; also user can write a query to accomplish the refresh, which 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
