A Performance Analysis of View Maintenance Techniques for Data Warehouses

Xing Wang
Dell Computer Corporation
Round Rock, Texas

Le Gruenwald
The University of Oklahoma
School of Computer Science
Norman, OK 73019

Guangtao Zhu
University of Science and Technology of China
Department of Computer Science and Technology
Beijing, P.R. China 100039

Abstract

A data warehouse stores integrated information as materialized views over data from one or more remote sources. These materialized views must be maintained in response to actual relation updates in the remote sources. Based on whether the current materialized views will be used in computing the new views, and whether the data warehouse will query the remote data sources for additional data to do the computation, the data warehouse view maintenance techniques are classified into four major categories: self-maintainable recomputation, not self-maintainable recomputation, self-maintainable incremental maintenance, and not self-maintainable incremental maintenance. This paper provides a comprehensive comparison of the techniques in these four categories in terms of the data warehouse space usage and number of rows accessed in order to propagate an update from a remote data source to a target materialized view in the data warehouse. The analysis shows that self-maintainable incremental maintenance performs the best in terms of both space usage and number of rows accessed.

1 Introduction

A data warehouse stores integrated information over data from one or more remote data sources for query and analysis [Hammer 95, Wiener 96]. The integrated information at the data warehouse is stored as materialized views. A view is a virtual relation defined using the actual relations stored in the database. A materialized view is the result relation of the evaluation of the relational algebra expression that defines the view relation [Silberschatz 97]. Using these

materialized views, user queries can be answered quickly as the information may be directly available or can be calculated using these materialized views on fly.

A problem known as *the view maintenance problem* is how to maintain the materialized views so that they can be kept up to date in response to updates of the actual relations in the remote data sources. There are numerous algorithms developed to solve the view maintenance problem for traditional database systems. In these database systems, query expressions defined views and actual relations are stored at the same database. The database systems understand view management and view definitions and know what data is needed for propagating updates to the views.

In a data warehouse, the query expressions that define views and actual relations may be stored at different database sources residing at many sites. The sources may inform the data warehouse when an update occurs but they might not be able to determine what data is needed for updating the views at the data warehouse. Therefore they may send only the actual data updates or the entire updated relations to the data warehouse. Upon receiving this information, the data warehouse may find that it needs some additional source data in order to update the views. Then it will issue some queries to some of the sources to request the additional source data. Some of the sources may have updated their data again before they evaluate the requesting queries from the data warehouse. Therefore they will send incorrect additional data to the data warehouse, which subsequently will use the incorrect data to compute the views. This phenomenon is called *distributed view maintenance anomaly*. Solving the view maintenance problem in data warehouses is thus more complicated than that in traditional database systems.

The objectives of this paper are to provide a classification of different view maintenance techniques that have been proposed in the literature and to conduct a comprehensive comparison

of these techniques in terms of space usage and number of rows accessed using the TPC benchmark for decision support queries. The rest of this paper is organized as follows. Section 2 classifies the existing view maintenance techniques. Section 3 presents a performance analysis of the techniques. Finally, Section 4 provides conclusions and future research.

2 Classification of Data Warehouse View Maintenance Techniques

Depending on whether the current materialized views in a data warehouse are used in the computation of their new views in response to updates that occur on the remote data sources, the existing data warehouse view maintenance techniques can be classified into two broad categories: recomputation and incremental view maintenance. Depending on whether the data warehouse has to query the remote data sources in order to calculate the new views, the techniques can be further classified as self-maintainable or not self-maintainable. The below subsections discuss these four categories.

2.1 The Self-Maintainable Recomputation Category

Materialized views can be computed from scratch by using the view definitions and other materialized views at the data warehouse. The current materialized views being maintained have no contribution to the calculation of the new views. Some techniques replicate all or part of the remote data at the data warehouse. We can view these replicated data as some kind of materialized views at the data warehouse. Others such as the self-maintenance warehouse approach discussed in [Quass 96] store the remote relations at the data warehouse as additional materialized views to provide data needed when the data warehouse computes the new views. Therefore, the data warehouse will never have to query the data sources for additional data.

A self-maintainable materialized V view can be defined in two ways. In the first way, the materialized view V can be defined by using other self-maintainable materialized views. In this case, the view V is defined as

$$V = \Pi_{proj}(\sigma_{cond}(v_1 \bowtie v_2 \bowtie \ldots \bowtie v_N))$$

where all v_i 's are self-maintainable materialized views stored in the data warehouse.

In the second way, view V can be defined by using the relations residing at the remote sources. In this case, the view definition is

$$V = \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_N))$$

where all r_i 's are self-maintainable relations stored in the remote data sources. When there is an update occurring at a remote data source, the data source knows what to do and will send the update with all related relations in the view definition to the data warehouse in a transaction. Therefore, the data warehouse does not have to send a query requesting additional data to the remote data sources as all necessary information is available.

However, a self-maintainable data warehouse view cannot be defined as

$$V = \Pi_{proj}(\sigma_{cond}(v_1 \bowtie v_2 \bowtie \ldots \bowtie v_k \bowtie r_{k+1} \bowtie r_{k+2} \bowtie \ldots \bowtie r_N))$$

where all relations v_i 's are self-maintainable materialized views residing at the data warehouse, all relations r_i 's are self-maintainable relations residing at the remote data sources, and there are totally N relations in the definition of view V. The reason is as follows. The data warehouse is still self-maintainable when an update of relation r_l is propagated to the data warehouse as all v_i 's are available in the data warehouse. However, when an update of view v_i is propagated to view V, the relation r_l is not available at the data warehouse. Therefore, the data warehouse has to send

a query to the remote data sources to get the relation r_l in order to calculate view V. Thus view V is not self-maintainable.

Although the above view V is not self-maintainable, we can add some more self-maintainable materialized views and redefine view V using these self-maintainable views to make it self-maintainable. For example, in the above view V where

$$V = \Pi_{proj}(\sigma_{cond}(v_1 \bowtie v_2 \bowtie \ldots \bowtie v_k \bowtie r_{k+1} \bowtie r_{k+2} \bowtie \ldots \bowtie r_N))$$

suppose we can define the following self-maintainable views

$$v_{k+1} = \Pi_{proj}(\sigma_{cond}(r_{k+1} \bowtie r_{k+2} \bowtie \ldots \bowtie r_l)) \text{ and } v_{k+2} = \Pi_{proj}(\sigma_{cond}(r_{l+1} \bowtie r_{l+2} \bowtie \ldots \bowtie r_N))$$

View V can then be redefined as

$$V = \Pi_{proj}(\sigma_{cond}(v_1 \bowtie v_2 \bowtie \ldots \bowtie v_k \bowtie v_{k+1} \bowtie v_{k+2}))$$

V is thus self-maintainable by definition.

The materialized view v_i at the data warehouse can be defined either by other self-maintainable views only, or by a collection of self-maintainable relations at the remote sites. We can view the relationships among the data warehouse materialized views as a hierarchy structure. The data warehouse may consist of many levels of materialized views. At the bottom level, there are materialized views defined by remote self-maintainable relations. The views defined by other self-maintainable materialized views are staying on top of those defined by remote relations.

The materialized views in the data warehouse have to be maintained in a correct order from the bottom level to the top level in the view hierarchy. The bottom level views have to be maintained first. Then those views in the second to the bottom level must be maintained next, and so on. Finally, the views at the top of the hierarchy are maintained. This is the order that updates from the remote data sources are propagated to the data warehouse.

An advantage of the techniques in this category is that the view maintenance anomaly problem is avoided as all necessary data are available at the data warehouse. The data warehouse knows the view definitions and what to do with the views to keep them up to date. It eliminates accesses to the remote relations, and therefore, it does not compete with the remote data sources' local resources. The data warehouse maintenance operations can then be totally separated from other OLTP operations. Whether a remote data source is available or not will not affect the data warehouse view maintenance process. However, in order to make the materialized views self-maintainable, additional materialized views that provide information necessary for view updates must be stored. Extra storage and time are thus needed to maintain these additional views.

2.2 The Not Self-Maintainable Recomputation Category

The data warehouse can recompute the materialized views from scratch using the view definitions, possibly some other materialized views at the data warehouse, and actual source relations periodically or whenever the source data is updated. That is, when an update occurs at the data source or periodically, the source will inform the data warehouse. According to the query expression that defines the view, the data warehouse may get part of data it wants from other materialized views at the data warehouse, and issue queries to the sources to get the other data it does not have. The sources send the query results back to the data warehouse. Based on the query results, the data warehouse calculates the views and stores the results as materialized views in the data warehouse.

The current materialized views have no contribution to the calculation of the new views. The data warehouse may replicate part of the remote relations in the data warehouse. However, these data are not enough for maintaining the materialized views. Therefore, the data warehouse

will have to query the remote data sources for additional data in order to maintain the views. An extreme case is where the data warehouse does not replicate any remote relations.

If the view maintenance process is not designed carefully, the distributed view maintenance anomaly problem will occur. Suppose that there is a data warehouse system where the remote data sources send updated relations to the data warehouse whenever an update occurs at the data sources. Upon receiving the information, the data warehouse is ready to compute the new views. But now let us assume that the data warehouse finds that it needs some other relations at some remote data sources to compute the new views. It will issue queries to these data sources. Suppose the data sources that sent the updated relations to the data warehouse update the relations again before they receive the queries from the data warehouse. The data sources answer the query and send the results to the data warehouse. These results might contain extra information that is incorrect. The data warehouse will then use the incorrect data to compute the new views, which will result in incorrect new views.

There are a lot of solutions to the distributed view maintenance anomaly problem. The simplest solution is as follows. Instead of sending the updated relations to the data warehouse, the data sources just simply inform the warehouse that there is an update that occurs at the data sources. The data warehouse will issue queries to the data sources to request all relations required in the view definitions. After the data sources receive the queries, they will send all requested relations to the data warehouse. The anomaly problem is thus avoided.

The not self-maintainable recomputation approach is simple. The anomaly problem can be avoided easily. However, the recomputation process is also time and resource consuming. The data warehouse sends queries back to the sources and waits for answers in order to computer

the new views. Processing these queries consumes the sources' local resources. If the sources are unavailable, the data warehouse will not get the answers it needs.

2.3 The Self-Maintainable Incremental Maintenance Category

In this category, the data warehouse views are maintained by using the view definitions, the materialized views, and the view updates. The data warehouse will never query the remote data sources as the information at the data warehouse is enough for maintaining the views. The data warehouse computes the view updates, then adds them to the materialized views. The process is incremental. Normally, only necessary remote relations, or views of the remote relations are stored at the data warehouse as materialized views. In the extreme case, all remote relations can be replicated at the data warehouse. The self-maintainable warehouse approaches discussed in [Cui 99], [Hull 96] and [Quass 96] belong to this category.

Let us discuss how to maintain a view V that is defined as

$V = \Pi_{proj}(\sigma_{cond}(v_1 \bowtie v_2 \bowtie \ldots \bowtie v_N))$

where each v_i is a materialized view and is defined as either

$v_i = \Pi_{proj}(\sigma_{cond}(v_{11} \bowtie v_{12} \bowtie \ldots \bowtie v_{1N}))$ where each v_{1j} is a view defined by other auxiliary materialized views,

or

$v_i = \Pi_{proj}(\sigma_{cond}(r_{11} \bowtie r_{12} \bowtie \ldots \bowtie r_{1N}))$ where each r_{1j} is a base relation.

In the first case, each view v_{1j} is defined either as

$v_{1j} = \Pi_{proj}(\sigma_{cond}(v_{21} \bowtie v_{22} \bowtie \ldots \bowtie v_{2N}))$ where each v_{2j} is a view defined by other auxiliary materialized views,

or

$v_{1j} = \Pi_{proj}(\sigma_{cond}(r_{21} \bowtie r_{22} \bowtie \ldots \bowtie r_{2N}))$ where each r_{2j} is a base relation.

Finally, at the lowest level of the view hierarchy discussed earlier in this paper, view v_{Mj} can only be defined by relations at the remote data sources as follows:

$v_{Mj} = \Pi_{proj}(\sigma_{cond}(r_{M1} \bowtie r_{M2} \bowtie \ldots \bowtie r_{MN}))$ where each r_{Mj} is a base relation.

The above view V is thus defined by M levels of the materialized views in the view hierarchy.

In the second case, the view can only be defined by base relations r_{1j}.

Each view v_{2j} can be defined by v_{3j}. Views can be defined by those views at one level lower than it. The lowest level views v_{Mj} can only be defined by relations at the remote data sources.

Suppose an update U_{ij} occurs at a data source, where i represents the view level in the view hierarchy and j represents the relation in the view definition where the update occurs. The data source sends the update U_{ij} along with the related base relations r_{i1}, r_{i2}, … and r_{iN} except r_{ij} to the data warehouse. The data warehouse calculates the view update $\Delta v_{(i-1)j}$ using the update U_{ij} and relations r_{i1}, r_{i2}, … and r_{iN} except r_{ij} as follows:

$$\Delta v_{(i-1)j} = \Pi_{proj}(\sigma_{cond}(r_{i1} \bowtie r_{i2} \bowtie \ldots \bowtie U_{ij} \bowtie \ldots \bowtie r_{iN}))$$

Then the view update $\Delta v_{(i-1)j}$ is added to the view $v_{(i-1)j}$ to produce the new view. The view $v_{(i-1)j}$ is thus maintained. The view update $\Delta v_{(i-1)j}$ is propagated to the views at one level higher than it. In this case, it is $v_{(i-2)j}$. The view $v_{(i-2)j}$ is defined by self-maintainable views as follows:

$$v_{(i-2)j} = \Pi_{proj}(\sigma_{cond}(r_{(i-1)1} \bowtie r_{(i-1)2} \bowtie \ldots \bowtie v_{(i-1)j} \bowtie \ldots \bowtie r_{(i-1)N}))$$

We can calculate the view update using the following formula:

$$\Delta v_{(i-2)j} = \Pi_{proj}(\sigma_{cond}(r_{(i-1)1} \bowtie r_{(i-1)2} \bowtie \ldots \bowtie \Delta v_{(i-1)j} \bowtie \ldots \bowtie r_{(i-1)N}))$$

This process is repeated until the top view V in the view hierarchy is updated.

All intermediate materialized views can be viewed as auxiliary views. These auxiliary views are self-maintainable. The materialized view *V* is self-maintainable by using the update information and additional information from the auxiliary views. The data warehouse views, including views such as *V* and auxiliary views, can be maintained starting with those views that do not depend on any other auxiliary views, working up to the final original view *V*.

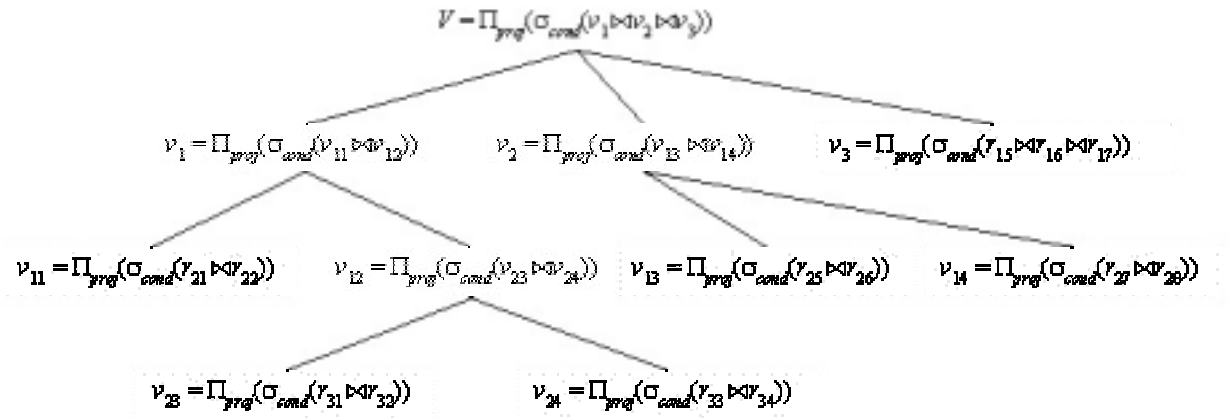

Figure 1. View Hierarchy Example

All related materialized primary views, auxiliary views and base relations can be drawn in a hierarchy structure as shown in Figure 1. All leaves in the hierarchy structure are those materialized views defined by the base relations. In this example, *V* is the primary materialized view. Views v_1, v_2 and v_{12} are materialized auxiliary views defined by other materialized auxiliary views. Views v_3, v_{11}, v_{13}, v_{14}, v_{23} and v_{24} are materialized auxiliary views defined by the base relations. All relations r_{ij} 's are the base relations. The views in the leaves should be maintained first. Suppose an update for r_{33} occurs in the data source. View v_{24} should be maintained first. Then views v_{12} and view v_1 must be maintained next in that order. Finally, the primary view *V* is maintained.

The data warehouse never needs to query the remote data source to get additional data. The data warehouse maintenance operations can be totally separated from other OLTP

operations. Whether the remote data source is available or not will not affect the data warehouse view maintenance process. However, in order to make the materialized views self-maintainable, the auxiliary views are stored in the data warehouse to provide the additional information. Extra storage and time overhead are therefore required to maintain the auxiliary views themselves. How to design materialized views at the data warehouse so that only necessary information are stored at the data warehouse is a major issue [Quass 96, Huyn 96a, 96b, 97b, 97c].

2.4 The Not Self-Maintainable Incremental Maintenance Category

Instead of recomputing every view from scratch, only parts of the warehouse that change are computed. However, the data warehouse has to query the remote data sources whenever necessary because the information at the data warehouse is not enough to maintain the view. A number of existing approaches fall under this category. Among them are the unrestricted base access [Zhuge 95, 96, 97a] and runtime warehouse self-maintenance [Huyn 97a].

2.4.1 Unrestricted Base Access

In the Unrestricted Base Access approach [Zhuge 95, 96, 97a], the data warehouse accesses the actual relations from the data sources whenever necessary in order to maintain the materialized views. There are many proposed algorithms that follow this approach. The Eager Compensating Algorithm (ECA) is the simplest among them. It is also the fastest algorithm that will let the data warehouse remain in a consistent state [Zhuge 98a and Zhuge 98b]. In this algorithm, compensating queries are sent back to the sources to offset the effects of concurrent updates [Zhuge 95]. They are used only when the next update occurs at the sources before the sources receive the data warehouse queries.

The data warehouse keeps a temporary table called COLLECT to keep the intermediate answers it receives from the data sources. It also keeps a set called Unanswered Query Set

(UQS) to keep track of those queries it sent to the data sources but has not received their answers yet.

Suppose an update U_i occurs at a data source. The data source sends U_i to the data warehouse. Suppose the data warehouse wants to update the materialized view that is defined as

$$V = \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_N))$$

where all relations r_i 's reside at a single data source. The data warehouse determines the query for calculating the delta view as

$$Q_i = \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_N))$$

It then creates a temporary COLLECT table and UQS set for processing this specific query, and sets both the COLLECT table and UQS to empty. The data warehouse writes the query Q_i to the UQS and sends the query Q_i to the data source. Suppose there is another update U_j that occurs at the same data source. The data source sends the update U_j to the data warehouse before it receives the query Q_i. The data warehouse now receives the update U_j. It knows that the upcoming answer for Q_i from the data source will contain extra information caused by simultaneous U_j update at the data source. Therefore, it has to offset this extra information. The data warehouse determines the query Q_j for updating U_j as follows

$$Q_j = \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie r_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie U_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N)) -$$
$$\Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie U_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N))$$

The first part of Q_j is the same as before; the second part is used to compensate for the extra information that query Q_i will receive. The data warehouse is eager to compensate before it receives the answer for query Q_i. Then it writes query Q_j to UQS and sends query Q_j to the data source. Now the UQS set contains two entries, i.e., Q_i and Q_j. The data source then processes

query Q_i and sends the answer to the data warehouse. The query answer A_i will contain the following extra information due to the update U_j,

$$\Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie U_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N))$$

The data warehouse receives the answer A_i, removes entry Q_i from UQS, writes the answer for Q_i to the temporary table COLLECT. Now UQS contains only one entry Q_j. The COLLECT table thus contains the following data:

$$A_i = \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie r_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N)) + \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie U_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N))$$

The data warehouse could not integrate the query answer into the data warehouse at this time as it will cause the data warehouse to contain inconsistent data. Therefore it has to wait for the query answer for query Q_j to come back. The COLLECT table is used to temporarily store query answers. After it receives the answer for the last unanswered query in UQS (in our case, Q_j), it can integrate the data in the COLLECT table into the data warehouse.

The data source processes query Q_j and sends answer A_j to the data warehouse. A_j contains the following data:

$$A_j = \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie r_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie U_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N)) - \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie U_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N))$$

The data warehouse receives the answer and removes the entry Q_j from the UQS set. Now UQS is empty. The answer A_j is integrated with the data in the COLLECT table as follows:

$$\Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie r_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N)) +$$
$$\Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie U_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N)).+$$
$$\Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie r_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie U_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N)) -$$
$$\Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie U_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N))$$
$$= \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie r_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N)) +$$
$$\Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie r_i \bowtie r_{i+1} \bowtie \ldots \bowtie r_{j-1} \bowtie U_j \bowtie r_{j+1} \bowtie \ldots \bowtie r_N))$$

Now the COLLECT table contains the correct answer that the data warehouse wants. The data warehouse can then integrate the data in the COLLECT table into the data warehouse.

The Unrestricted Base Access approach is simple. Instead of discarding old materialized views and calculating new views from scratch, this approach calculates view updates then adds them to the old views in order to get the new views. However, the data warehouse has to access N –1 remote source actual relations in order to propagate one source update. In the eager compensating algorithm discussed above, all N actual relations have to be accessed if compensating queries are used.

In this approach, the data warehouse may have to send queries back to the sources and waits for answers in order to compute the view updates. Therefore, this approach has the same limitation as the not self-maintainable recomputation approach. Computing these queries consumes remote sources' local resources, and will slow down other OLTP operations. If the remote sources are unavailable, the data warehouse will not get the answers it needs.

2.4.2 Runtime Warehouse Self-Maintenance

In the self-maintenance warehouse approach, at design time, we determine views to be self-maintainable and decide to add more auxiliary views for additional information. Design-time self-maintainability is not flexible. It may be difficult or impossible for us to know the exact contents of the views and their updates at design time. To solve this problem, a run time warehouse maintenance approach has been introduced [Huyn 97a].

The basic idea of the runtime self-maintenance approach is that the data warehouse generates the self-maintainable test for the views to determine whether the views are self-maintainable for a particular update. At run time, the self-maintainable test determines the views for self-maintainability. If a view is self-maintainable, then the view can be maintained by the

update information, the view itself, and the query expression that defines the view. In this case, the run-time self-maintainable approach is similar to the self-maintainable warehouse approach discussed in the previous section. However, the data warehouse does not store and maintain any auxiliary views. If the view is not self-maintainable, then the data warehouse has to query the remote data sources for those relations it needs in order to update the view. In this case, this approach is similar to the unrestricted base access approach.

The runtime warehouse self-maintenance approach uses the view maintainability test first before doing any maintenance. This creates the overhead in the data warehouse. However it maintains the views by using the views in the data warehouse or some base relations only, without requiring auxiliary views that are necessary in the self-maintainable incremental maintenance approach. This saves the storage space in the data warehouse.

3 Performance Analysis

We conduct an analysis to compare the performance of different algorithms in the four categories. We consider only the problem of single view maintenance in a single source environment because the ECA algorithm in the not self-maintainable incremental maintenance category can only be used in this environment [Zhuge 97a, Zhuge 96]. For self-maintainable view maintenance techniques, we also consider the case where all actual relations are replicated at the data warehouse.

3.1 Performance Measurements

In our analysis, only Select-Project-Join views are considered. We measure the performances of the techniques in terms of space and number of row accesses, which are defined as follows:

- Space: total space needed to store the data in the data warehouse, including space for auxiliary views. We do not consider indices.
- Number of rows accessed: the number of rows that must be accessed in the data warehouse and the data sources in order to integrate the updates into the data warehouse.

3.2 Analysis Parameters

The analysis parameters and their default values are listed in Table 1. The default values are calculated based on the TPC benchmark for decision support queries [TPC 99].

Meaning	**Symbol**	**Default value**	**Range**
Cardinality of view V	$Card(V)$	914	0 ~ 100,000
Tuple size of view V (in bytes)	$ts(V)$	43	10 ~ 250
Number of auxiliary views per view	Nav	3	1 ~ N
Number of base relations in the view definition	N	3	1 ~ 7
Cardinality of base relation r	$Card(r)$	108,000,000	0 ~ 1,000,000,000
Tuple size of base relation r (in bytes)	$ts(r)$	116	100 ~ 180
Selectivity: the fraction of tuples that satisfy the select condition	σ	0.003	0.00001 ~ 1.0
The join factor is roughly the fraction of tuples in a relation that join with others	j	0.73	0.00001 ~ 1.0
Number of tuples in a relation that join with others	$J = j \times Card(r)$	Calculated	Calculated
Number of interfering updates for each query	I	0.5	0 ~ 100
Cardinality of update	$Card(U)$	1	
Number of tuple insertions in a base relation at data source	$Nupdate$	1	100

Table 1. Analysis Parameters

3.3 Comparison Based on Space Needed in the Data Warehouse

3.3.1 Self-Maintainable Recomputation

The techniques in this category do not query the remote data source for additional data in order to maintain the data warehouse materialized views. The data warehouse can replicate all or part of the remote base relations at the data warehouse. These additional data take space at the data warehouse. Here we consider the case where the materialized views are defined by other materialized views (auxiliary views) at the data warehouse, and all auxiliary views are replicated remote relations. A view V is defined as

$$V = \Pi_{proj}(\sigma_{cond}(v_1 \bowtie v_2 \bowtie \ldots \bowtie v_N))$$

where all v_i 's are materialized views stored in the data warehouse.

In the average case, the amount of space needed is:

$$Card(V)\ ts(V) + \sum_{i=0}^{Nav} Card(AV_i)\ ts(AV_i)$$

where $0 \le Nav \le N$. If all auxiliary views are the same, the above formula can be reduced to

$$Card(V)\ ts(V) + Nav\ Card(AV)\ ts(AV)$$

$Nav = 0$ when none of the auxiliary views is needed. It means that the materialized view is a replicated remote relation. This is the best case. The formula can be rewritten as follows:

$$Card(V)\ ts(V)$$

If all auxiliary views are needed to maintain the view, then $Nav = N$. This is the worst case. The formula can be rewritten as follows:

$$Card(V)\ ts(V) + \sum_{i=0}^{N} Card(AV_i)\ ts(AV_i)$$

If all auxiliary views are the same, the above formula can be reduced to

$$Card(V)\ ts(V) + N\ Card(AV)\ ts(AV)$$

3.3.2 Not Self-Maintainable Recomputation

Here we consider only the case where the data warehouse does not replicate any base relations. Therefore, the data warehouse always has to query the remote data sources. The data warehouse stores only materialized views. In this extreme situation, the amounts of space needed in the best case, the average case and the worst case are the same, and are equal to $Card(V)\ ts(V)$.

3.3.3 Self-Maintainable Incremental Maintenance

Similar to the self-maintainable recomputation techniques, the techniques in this category can replicate all or part of the remote data at the data warehouse. Here we consider only the case where the materialized views are defined by other materialized views (auxiliary views) at the data warehouse, and all auxiliary views are replicated remote relations. A view V is defined as

$$V = \Pi_{proj}(\sigma_{cond}(v_1 \bowtie v_2 \bowtie \ldots \bowtie v_N))$$

where all v_i 's are materialized views stored in the data warehouse. The amount of space needed at the data warehouse is the same as that of the self-maintainable recomputation techniques.

In the average case, the space needed is as follows:

$$Card(V)\ ts(V) + \sum_{i=0}^{Nav} Card(AV_i)\ ts(AV_i)$$

where $0 \le Nav \le N$. If all auxiliary views are the same, the above formula can be reduced to

$$Card(V)\ ts(V) + Nav\ Card(AV)\ ts(AV)$$

In the best case, no auxiliary views are needed as the view is self-maintainable. The formula can be rewritten as follows:

$$Card(V)\ ts(V)$$

In the worst case, all auxiliary views are needed to maintain the view. The formula can be rewritten as follows:

$$Card(V)\ ts(V) + \sum_{i=0}^{N} Card(AV_i)\ ts(AV_i)$$

In the worst case, the auxiliary view's tuple size and cardinality are equal to those of the remote relation. If all auxiliary views are the same, the above formula can be reduced to the following:

$$Card(V)\ ts(V) + N\ Card(AV)\ ts(AV)$$

3.3.4 Not Self-Maintainable Incremental Maintenance

Here we consider the Eager Compensating Algorithm (ECA) for this category. In ECA, a temporary table COLLECT is used to store intermediate query answers. For every update, the queries including compensated queries are sent to the data source. Note that the COLLECT table is empty only when there is no query to the data source, or the answers for all the queries are returned to the data warehouse before a new update occurs at the data source. This is the best case. The space needed is as follows:

$Card(V)\ ts(V)$

In the average case, the size of the COLLECT table for a specific update with all its interfering updates I is equal to the sum of the cardinalities of the final query answers for all queries. The total number of queries sent to the data source can be calculated as below [Zhuge 97b]:

$$Nq = \sum_{k=1}^{N-1}\left(I^{k-1}\sigma^{k} j^{k-1} \sum_{i=0}^{N-k-1}\left(\sigma^{i} J^{i}\right)\right)$$

In the worst case, the number of queries sent to the data source is:

$$Nq = \sum_{k=1}^{N-1}\left(I^{k-1} \sum_{i=0}^{N-k-1}\left(Card(r)^{i}\right)\right).$$

Let us derive the formula for the multiple tuple update case. We follow the method used in [Zhuge 97b]. At first, we have to find the number of wrapper queries sent from a view

manager to the query processor [Zhuge 95, Zhuge 97b]. In the single tuple update case, $Card(A^1) = \sigma_1 \times 1 = \sigma_1$. In the multiple tuple update case, $Card(A^1) = \sigma_1 Card(U)$.

The total number of wrapper queries to evaluate the update query is

$$Nq = \begin{cases} 0 & : \quad N = 1 \\ \sigma_1 Card(U) & : \quad N = 2 \\ \sigma_1 Card(U) \times \sum_{k=2}^{N} \left(\prod_{i=2}^{k-1} \sigma_i J_i \right) & : \quad N > 2 \end{cases}$$

Assume all the σs and *J*s are the same, the formula is simplified to the following:

$$Nq = \begin{cases} 0 & : \quad N = 1 \\ \sigma Card(U) \times \sum_{i=0}^{N-2} \left(\sigma^i J^i \right) & : \quad N > 1 \end{cases}$$

The compensating query is defined as:

$$CQ = \Pi_{proj}(\sigma_{cond} (U_1 \bowtie U_2 \bowtie \ldots \bowtie U_n \bowtie R_{n+1} \bowtie R_{n+2} \bowtie \ldots \bowtie R_N))$$

There are *n* multiple tuple updates and *N*-*n* actual (base) relations. The number of wrapper queries to evaluate the query is:

$$Nq = \begin{cases} 0 & : \quad n = N \\ \sigma^n j^{n-1} Card(U)^n \sum_{i=0}^{N-n-1} \left(\sigma^i J^i \right) & : \quad N > 1 \end{cases}$$

Then, we estimate the number of compensating queries as $\sum_{i=0}^{N-1} I^i$, where *I* is defined as the maximum number of interfering updates that occur between the time when the data warehouse sends a query to the remote data source and the time when the data warehouse receives the corresponding answer [Zhuge 98b]. In the worst case, there are *I* interfering updates for each query. The number of wrapper queries is:

$$Nq = \sum_{k=1}^{N-1}\left(I^{k-1}\sigma^{k} j^{k-1} Card(U)^{k} \sum_{i=0}^{N-k-1}\left(\sigma^{i} J^{i}\right)\right)$$

When $Card(U) = 1$, the above formula is reduced to the following:

$$Nq = \sum_{k=1}^{N-1}\left(I^{k-1}\sigma^{k} j^{k-1} \sum_{i=0}^{N-k-1}\left(\sigma^{i} J^{i}\right)\right)$$

which is the formula of the single tuple update case [Zhuge 98b].

Suppose the cardinality of the final query answer A^N for all interfering updates are equal. $Card(A^N) = \sigma_1 \sum_{i=2}^{N} \sigma_i J_i$ for the single tuple update case [Zhuge 98b]. For the multiple tuple update case, $Card(A^N) = \sigma_1 Card(U) \sum_{i=2}^{N} \sigma_i J_i$.

When all σs and Js are equal, $Card(A^N) = \sigma^N Card(U) J^{N-1}$. For a query with n update relations and N–n base relations, $Card(A^N) = \sigma^N Card(U)^n j^{n-1} J^{N-n}$, where $1 \le n \le N$.

The max cardinality of the COLLECT table for updates at a time is:

$$\sum_{k=1}^{N-1}\left(I^{k-1}\sigma^{k} j^{k-1} Card(U)^{k} \left(\sigma^{N} Card(U)^{k} j^{k-1} J^{N-k}\right) \sum_{i=0}^{N-k-1}\left(\sigma^{i} J^{i}\right)\right)$$

$$= \sum_{k=1}^{N-1}\left(I^{k-1}\sigma^{N+k} j^{2(k-1)} Card(U)^{2k} J^{N-k} \sum_{i=0}^{N-k-1}\left(\sigma^{i} J^{i}\right)\right).$$

The space needed is

$$Card(V)ts(V) + \sum_{k=1}^{N-1}\left(I^{k-1}\sigma^{N+k} j^{2(k-1)} Card(U)^{2k} J^{N-k} \sum_{i=0}^{N-k-1}\left(\sigma^{i} J^{i}\right)\right) ts(V) .$$

In the worst case, when $\sigma = 1$, $j = 1$, $J = j \times Card(r) = Card(r)$. The space needed at the data warehouse is

$$Card(V)ts(V) + \sum_{k=1}^{N-1}\left(I^{k-1} Card(U)^{2k} Card(r)^{N-k} \sum_{i=0}^{N-k-1}\left(Card(r)^{i}\right)\right) ts(V) .$$

3.3.5 Comparison Results

In summary, for a given view *V*, the amounts of space needed in the data warehouse for the four categories are listed in Table 3, where *AV* stands for Auxiliary View.

Category	**Space needed in the data warehouse**
Self-Maintainable Recomputation (SMR)	The best case: $Card(V)\ ts(V)$
	The average case: $Card(V)\ ts(V) + Nav\ Card(AV)\ ts(AV)$ $0 \le Nav \le N$
	The worst case: $Card(V)\ ts(V) + N\ Card(AV)\ ts(AV)$
Not Self-Maintainable Recomputation (NSMR)	$Card(V)\ ts(V)$
Self-Maintainable Incremental Maintenance (SMIM)	The best case: $Card(V)\ ts(V)$
	The average case: $Card(V)\ ts(V) + Nav\ Card(AV)\ ts(AV)$ $0 \le Nav \le N$
	The worst case: $Card(V)\ ts(V) + N\ Card(AV)\ ts(AV)$
Not Self-Maintainable Incremental Maintenance (NSIM)	The best case: $Card(V)\ ts(V)$
	The average case: $Card(V)ts(V) + \sum_{k=1}^{N-1}\left(I^{k-1}\sigma^{N+k} j^{2(k-1)} Card(U)^{2k} J^{N-k} \sum_{i=0}^{N-k-1}\left(\sigma^i J^i\right)\right) ts(V)$
	The worst case: $Card(V)ts(V) + \sum_{k=1}^{N-1}\left(I^{k-1} Card(U)^{2k} Card(r)^{N-k} \sum_{i=0}^{N-k-1}\left(Card(r)^i\right)\right) ts(V)$

Table 2. Space needed in the data warehouse

We draw the graphs for the formulas derived above using the parameters' defaults values listed in Table 1. Due to space limitation, we will only discuss the results of the average case. In this case, the not self-maintainable recomputation approach does not require extra space at the data warehouse. However, for the self-maintainable recomputation and self-maintainable incremental approaches, extra space is required at the data warehouse to store the replicated

source data as auxiliary views. This extra space is proportional to the total number of auxiliary views *Nav*, the auxiliary view cardinality *Card*(*AV*) and the auxiliary view tuple size *ts*(*AV*).

For NSIM, extra space is required to store the COLLECT table. Suppose that the tuple size of the COLLECT table is equal to the view's tuple size. In some cases, the space used to store the COLLECT table in NSIM is small compared to the space used for storing the materialized views in SMR and SMIM. In other cases, the space used for storing the COLLECT table can be grown much larger than that of materialized views. Let us examine Figures 2-4, where NSMR is not shown since it overlaps with the x-axis. At first, the space used for storing the COLLECT table is smaller and growing slower than that used for storing the materialized auxiliary views when the cardinality of auxiliary views is small (Figure 2). Then, the space used for storing the COLLECT table grows faster. At some point ($Card(AV) = 6.0 \times 10^7$ in our case), the space used for storing the COLLECT table exceeds the space used for storing the materialized auxiliary views (Figure 3). When the cardinalities of auxiliary views (or base relations) are large, the space used for storing the COLLECT table is much larger than that used to store the auxiliary views (Figure 4).

From Table 2, we can see that the space needed for the NSIM approach grows exponentially with the select factor, join factor, and cardinality of updates.

In summary, in the average case, as no additional space is required at the data warehouse for the not self-maintainable approach, it is the best approach in terms of space used in the data warehouse. The amounts of space used for both the self-maintainable approaches, SMR and SMIR, are the same. When there are very few conflicting updates and the cardinalities of the base relations are relatively small, the space used for the not self-maintainable approach is less than those of both the self-maintainable approaches. However, if there are a lot of conflicting

updates and the cardinalities of the base relations are large, the space used for the not self-maintainable approaches are much larger than those of both the self-maintainable approaches.

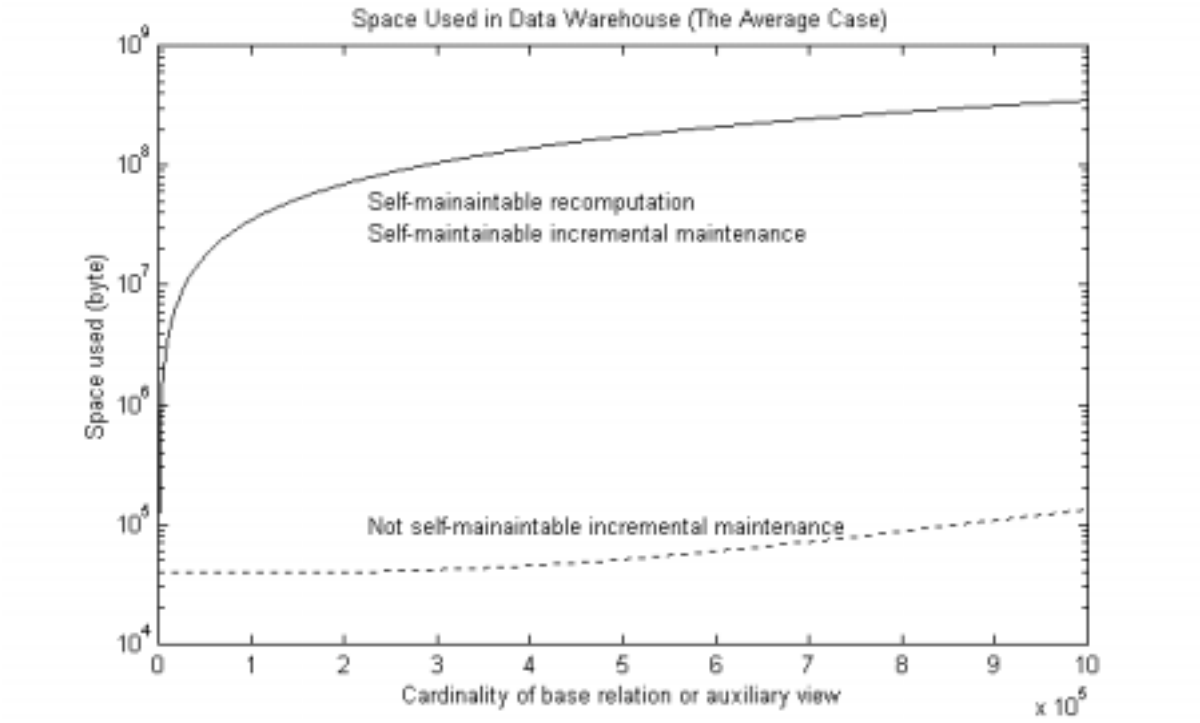

Figure 2. Space used in the data warehouse (The average case)

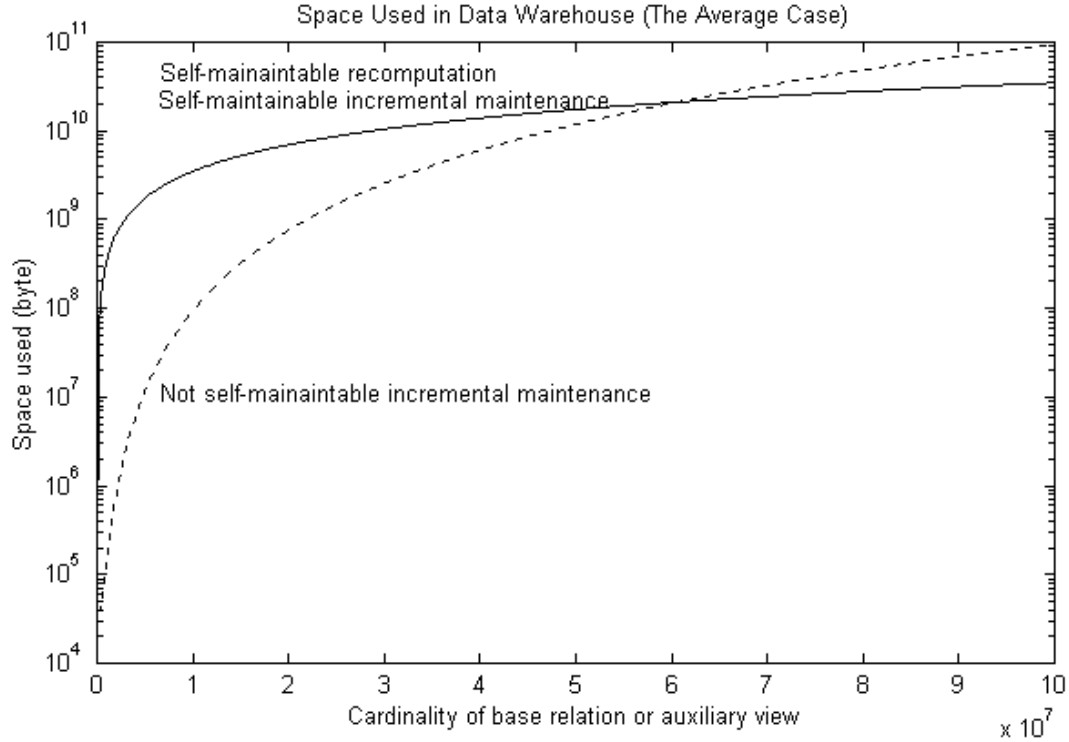

Figure 3. Space used in the data warehouse (The average case)

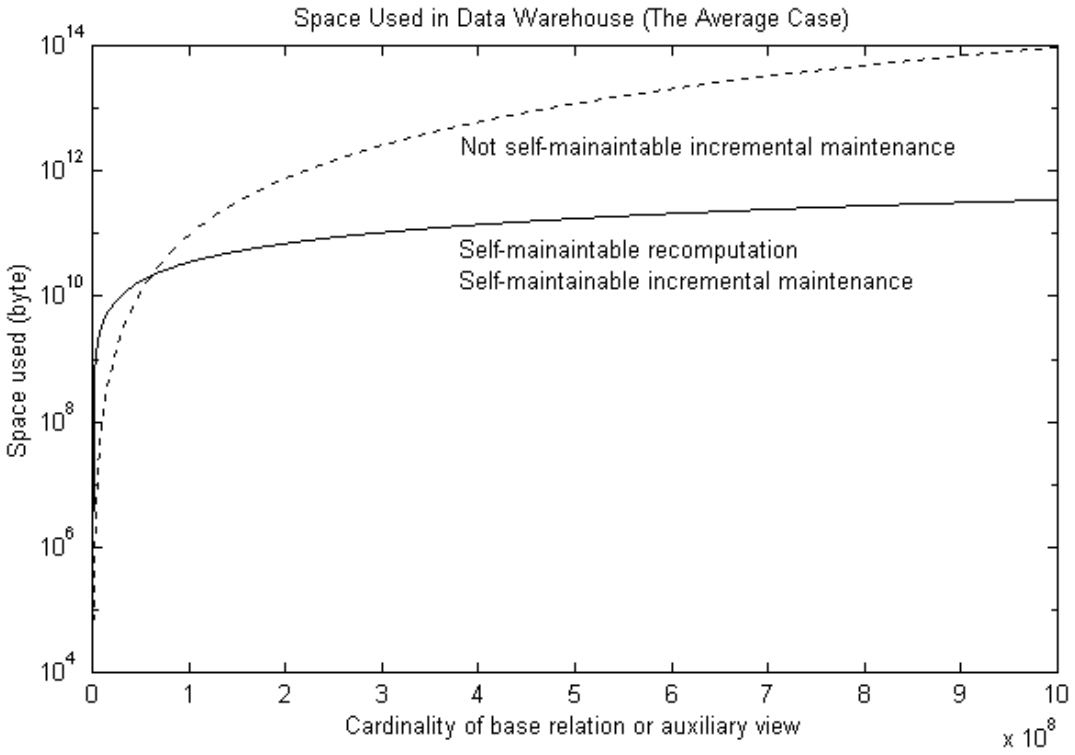

Figure 4. Space used in the data warehouse (The average case)

3.4 Comparison Based on the Number of Rows Accessed

To analyze the number of rows accessed at the data warehouse by the techniques, we made the following assumptions:

- The set of a primary view and its auxiliary views (if any) is independent to other sets of primary views and their auxiliary views.
- We do not consider indices. Linear search is thus used to check if a record satisfies a select or join condition.
- All auxiliary views are self-maintainable and are replicated base relations.
- Updates to auxiliary views and primary views are for appending only.

3.4.1 Self-Maintainable Recomputation

The data warehouse will never query the remote data sources as all necessary data are available at the data warehouse. Updates from the remote data sources have to be propagated to the replicated relations at the data warehouse first, then the data warehouse recalculates the view relation and stores the result at the data warehouse as the new materialized view. In order to propagate an update to data warehouse replicated relation, the number of rows to be accessed at the data warehouse is the cardinality of the relation itself plus the cardinality of the update. That is,

$Card(r) + Card(U)$.

Then the data warehouse recomputes the materialized view using the view definition. Suppose the data warehouse materialized view is defined as:

$$Q = \Pi_{proj}(\sigma_{cond}(v_1 \bowtie v_2 \bowtie \ldots \bowtie v_N))$$

The strategy such as the one described in [Zhuge 97b] can be used to evaluate the query Q that defines the view V. Let us rename the actual relations according to the join order. The query Q that defines the view becomes

$$Q = \Pi_{proj}(\sigma_{cond}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_N))$$

Using the equivalence rules [Silberschatz 97], the select condition *cond* can be decomposed into a sequence of individual selections σ_i 's on individual actual relations R_i 's. Let T^1 be the temporary relation after σ_1 is applied to R_1, and let T^k be the temporary relation after the join and selection with relation R_k is done. T^N is the new view. To compute T^k, each tuple in R_k has to be accessed for $Card(T^{k-1})$ times. $T^k = T^{k-1} \bowtie \sigma_k(R_k)$. $Card(T^k) = \sigma_k J_k \; Card(T^{k-1})$ and $Card(T^1) =$

$\sigma_1 Card(R_1)$. Each tuple in R_1 is accessed once. R_2 is accessed $Card(T^1) = \sigma_1\ Card(R_1)$ times. The number of tuples accessed for R_1 and R_2 is

$$Card(R_1) + Card(R_2)\ \sigma_1 Card(R_1)$$

Thus,

$$Card(T^2) = \sigma_2 J_2\ Card(T^1) = \sigma_2 J_2\ \sigma_1\ Card(R_1).$$

To get the new view, the total number of tuples that must be accessed is

$$Card(R_1) + Card(R_2)\ \sigma_1\ Card(R_1) + Card(R_3)\ \sigma_1 Card(R_1)\ \sigma_2 J_2 + \ldots +$$
$$Card(R_N)\ \sigma_1 Card(R_1)\ \Pi_{i=2}^{N-1} \sigma_i J_i$$
$$= Card(R_1)\ (1 + Card(R_2)\ \sigma_1 + Card(R_3)\ \sigma_1 \sigma_2 J_2 + \ldots +\ Card(R_N)\ \sigma_1 \Pi_{i=2}^{N-1} \sigma_i J_i\ )$$

Suppose all the select and join conditions, and cardinalities of all base relations are the same, the total number of tuples to be accessed in order to recompute the view is equal to

$$Card(AV) + Card(AV)^2\ (\sigma + \sigma^2 J + \sigma^3 J^2 + \ldots + \sigma^{N-1} J^{N-2})$$
$$= Card(AV) + Card(AV)^2 \left(\sum_{i=1}^{N-1} \left(\sigma^i J^{i-1} \right) \right)$$

This is the average case. In the worst case, when $\sigma = 1$, $j = 1$, $J = j$ $Card(AV) = Card(AV)$, the total number of tuples to be accessed is equal to

$$Card(AV) + Card(AV)^2\ (1 + Card(AV) + Card(AV)^2 + \ldots + Card(AV)^{N-2})$$
$$= Card(AV)\ (1 + Card(AV) + Card(AV)^2 + \ldots + Card(AV)^{N-1})$$
$$= Card(AV) \times \left(\sum_{i=0}^{N-1} \left(Card(AV)^i \right) \right)$$

For *Nupdate* updates, the total number of tuples to be accessed in the average case is

$$Nupdate \times \left(Card(AV) + Card(AV)^2 \times \left(\sum_{i=1}^{N-1} \left(\sigma^i J^{i-1} \right) \right) \right)$$

In the worst case, the formula is

$$Nupdate \times \left(Card(AV) \times \left(\sum_{i=0}^{N-1} \left(Card(AV)^i \right) \right) \right)$$

Including the number of rows that need to be accessed to maintain the view and replicated data at the data warehouse, the total number of rows to be accessed in the average case is

$$Nupdate \times \left(Card(V) + 2 \times Card(AV) + Card(U) + Card(AV)^2 \times \left(\sum_{i=1}^{N-1} \left(\sigma^i J^{i-1} \right) \right) \right)$$

In the worst case, the number becomes

$$Nupdate \times \left(Card(V) + Card(U) + Card(AV) \times \left(1 + \sum_{i=0}^{N-1} \left(Card(AV)^i \right) \right) \right)$$

3.4.2 Not Self-Maintainable Recomputation

Only source data is required to be accessed. The reason is that the warehouse recalculates the full view using the source data each time. It does not use the data warehouse data. Suppose the system locks all base relations in order to evaluate the query expression that defines the view. If the nested-loop join method [Silberschatz 97] is used to evaluate it, the total number of rows to be accessed is $Card(r)^N$.

Another strategy such as the one described in [Zhuge 97b] can also be used to evaluate the query Q that defines the view V. It will reduce the total number of rows to be accessed. Let us rename the actual relations according to the join order. The query Q that defines the view becomes

$$Q = \Pi_{proj}(\sigma_{cond}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_N))$$

Using the same method presented in Section 3.4.1, suppose all select and join conditions, and cardinalities of all base relations are the same, the total number of tuples to be accessed in order to recompute the view is equal to

$$Card(r) + Card(r)^2 \, (\sigma + \sigma^2 J + \sigma^3 J^2 + \ldots + \sigma^{N-1} J^{N-2})$$

$$= Card(r) + Card(r)^2 \times \left(\sum_{i=1}^{N-1} \left(\sigma^i J^{i-1} \right) \right)$$

This is the average case. In the worst case, when $\sigma = 1$, $j = 1$, $J = j$ $Card$ (r) = $Card$ (r), the total number of tuples to be accessed is equal to

$Card(r) + Card(r)^2 (1 + Card(r) + Card(r)^2 + \ldots + Card(r)^{N-2})$

$= Card(r) (1 + Card(r) + Card(r)^2 + \ldots + Card(r)^{N-1})$

$$= Card(r) \times \left(\sum_{i=0}^{N-1} \left(Card(r)^i \right) \right)$$

For *Nupdate* updates, the total number of tuples to be accessed in the average case is

$$Nupdate \times \left(Card(r) + Card(r)^2 \times \left(\sum_{i=1}^{N-1} \left(\sigma^i J^{i-1} \right) \right) \right).$$

In the worst case, the formula is

$$Nupdate \times \left(Card(r) \times \left(\sum_{i=0}^{N-1} \left(Card(r)^i \right) \right) \right)$$

3.4.3 Self-Maintainable Incremental Maintenance

No queries are sent to the data sources for additional information. Therefore, the number of rows accessed in the data source is equal to 0. For N base relations in a view, *Nav* should be less than or equal to N. In the worst case, *Nav* is equal to N.

At first, the auxiliary view itself has to be maintained before the primary materialized view can be maintained. Let $Card(U)$ stand for the cardinality of update U. According to our assumption that auxiliary views are self-maintainable and updates are used for appending only, the number of rows needed to be accessed in order to maintain the auxiliary view is $Card(U) + Card(AV)$. Let $Card(\Delta AV)$ stand for the cardinality of the auxiliary view update, which is the same as $Card(U)$. Then the update is propagated to the primary view. We need to calculate the

primary view update. The number of rows accessed can be estimated using the same method presented in Section 3.4.1. Suppose the all select and join conditions, and cardinalities of all auxiliary views are the same, the total number of tuples to be accessed in order to calculate the primary view update is equal to

$$Card(\Delta AV) + Card(\Delta AV)\ Card(AV)\ \ (\sigma + \sigma^2 J + \sigma^3 J^2 + \ldots + \sigma^{Nav\text{-}1} J^{Nav\text{-}2})$$
$$= Card(\Delta AV) \times \left(1 + Card(AV) \times \left(\sum_{i=0}^{Nav-2} \left(\sigma^{i+1} J^i\right)\right)\right)$$

Finally, the view update is appended to the primary view. The total number of rows to be accessed in order to propagate a source update U to the data warehouse is

$$Card(V) + Card(U) + Card(AV) + Card(\Delta AV) \times \left(1 + Card(AV) \times \sum_{i=0}^{Nav-2} \left(\sigma^{i+1} J^i\right)\right)$$

Since $Card(\Delta AV) = Card(U)$, we have

$$Card(V) + Card(AV) + Card(\Delta AV) \times \left(2 + Card(AV) \times \sum_{i=0}^{Nav-2} \left(\sigma^{i+1} J^i\right)\right)$$

This is the average case. In the worst case, when $\sigma = 1$, $j = 1$, $J = j\ Card(AV) = Card(AV)$, $Nav = N$, the total number of tuples to be accessed is equal to

$$Card(V) + Card(AV) + Card(\Delta AV) \times \left(2 + Card(AV) \times \sum_{i=0}^{N-2} \left(Card(AV)^i\right)\right)$$
$$= Card(V) + Card(AV) + Card(\Delta AV) \times \left(2 + \sum_{i=1}^{N-1} \left(Card(AV)^i\right)\right)$$

In the single tuple update case, i.e., $Card(U) = Card(\Delta AV) = 1$, the total number of tuples to be accessed is equal to

$$Card(V) + Card(AV) + \left(2 + \sum_{i=1}^{N-1} \left(Card(AV)^i\right)\right)$$

For *Nupdate* updates, the average number of tuples to be accessed for the multiple tuple update case is

$$Nupdate \times \left(Card(V) + Card(AV) + Card(\Delta AV) \times \left(2 + Card(AV) \times \sum_{i=0}^{Nav-2} \left(\sigma^{i+1} J^{i} \right) \right) \right)$$

For the worst case, the total number of tuples to be accessed is

$$Nupdate \times \left(Card(V) + Card(AV) + Card(\Delta AV) \times \left(2 + \sum_{i=1}^{N-1} \left(Card(AV)^{i} \right) \right) \right)$$

3.4.4 Not Self-Maintainable Incremental Maintenance

In the ECA algorithm, all tuples in the view table have to be accessed in order to find a tuple to integrate with the view update. However, the data warehouse may have to access data from remote sites except for the best case. Parts of these queries are compensated.

In Section 3.3.4, we derive the number of wrapper queries corresponding to queries with $N - n$ relations in the multiple tuple update case as

$$Nq = \sum_{k=1}^{N-1} \left(I^{k-1} \sigma^{k} j^{k-1} Card(U)^{k} \sum_{i=0}^{N-k-1} \left(\sigma^{i} J^{i} \right) \right)$$

When there are totally *Nupdate* updates, the number of wrapper queries is

$$Nq = Nupdate \times \sum_{k=1}^{N-1} \left(I^{k-1} \sigma^{k} j^{k-1} Card(U)^{k} \sum_{i=0}^{N-k-1} \left(\sigma^{i} J^{i} \right) \right)$$

In general, for a query that is defined as $Q = \Pi_{proj}(\sigma_{cond}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_N))$, the total number of tuples to be accessed in order to process the query is

$$Card(R_1)\ (1 + Card(R_2)\ \sigma_1 + Card(R_3)\ \sigma_1\sigma_2 J_2 + \ldots + Card(R_N)\ \sigma_1 \Pi_{i=2}^{N-1} \sigma_i J_i\ )$$

For a compensating query $Q = \Pi_{proj}(\sigma_{cond}\ (U_1 \bowtie U_2 \bowtie \ldots \bowtie U_k \bowtie R_{k+1} \bowtie R_{k+2} \bowtie \ldots \bowtie R_N))$, there are k multiple tuple updates and N-k base relations. Suppose all updates are evaluated first, the number of rows to be accessed in order to evaluate a query is

$$Card(U) + Card(U)\ \sigma\ Card(U) + Card(U)\ \sigma\ Card(U)\ \sigma\ j\ Card(U) + \ldots + \sigma^{k-1}\ j^{k-2}\ Card(U)^{k} + \sigma^{k}\ j^{k-1}\ Card(U)^{k}\ Card(r)\ \Pi_{i=k}^{N-1} \sigma_i J_i$$

$= Card(U) + \sigma\, Card(U)^2 + \sigma^2\, j\, Card(U)^3 + \ldots + \sigma^{k-1}\, j^{k-2}\, Card(U)^k +$
$+ \ldots + \sigma^k\, j^{k-1}\, Card(U)^k\, Card(r) + \ldots + \sigma^k\, j^{k-1}\, Card(U)^k\, Card(r)\, \Pi_{i=k}^{N-1} \sigma_i J_i$

$$= Card(U) + \sum_{i=0}^{k-2} \left(\sigma^{i+1}\, j^i\, Card(U)^{i+2} \right) + \sigma^k\, j^{k-1} Card(U)^k\, Card(r) \sum_{i=0}^{N-k-1} \left(\sigma^i J^i \right)$$

For the multiple tuple update case, the total number of tuples to be accessed in order to propagate *Nupdate* updates from the remote source to the target data warehouse view is:

$$Nupdate \times \sum_{k=1}^{N-1} \left(I^{k-1} \times number\_of\_queries \times tuples\_accessed\_per\_query \right)$$

$$= Nupdate \times \sum_{k=1}^{N-1} (I^{k-1} \sigma^k\, j^{k-1} Card(U)^k \sum_{i=0}^{N-k-1} \left(\sigma^i J^i \right) \times (Card(U) + \sum_{i=0}^{k-2} \left(\sigma^{i+1}\, j^i\, Card(U)^{i+2} \right) + \sigma^k\, j^{k-1} Card(U)^k\, Card(r) \sum_{i=0}^{N-k-1} \left(\sigma^i J^i \right)))$$

In the worst case, the total number is

$$Nupdate \times \sum_{k=1}^{N-1} (I^{k-1} Card(U)^k \sum_{i=0}^{N-k-1} \left(Card(r)^i \right) \times (Card(U) + \sum_{i=0}^{k-2} \left(Card(U)^{i+2} \right) + Card(U)^k\, Card(r) \sum_{i=0}^{N-k-1} \left(Card(r)^i \right)))$$

3.4.5 Comparison Results

The formulas to calculate the number of rows that need to be accessed in order to maintain a materialize view for all four categories are listed in Tables 3 and 4. We use the parameters' default values listed in Table 2 to draw the graphs showing the number of rows that need to be accessed to maintain a materialized view. Due to space limitation, here we will discuss the results for the average case only.

Category	Number of rows accessed in data warehouse	Number of rows accessed in data source
SMR	The average case: $Nupdate \times (Card(V) + 2Card(AV) + Card(U) + Card(AV)^2 \left(\sum_{i=1}^{N-1} \left(\sigma^i J^{i-1} \right) \right))$	0
	The worst case: $Nupdate \times (Card(V) + Card(U) + Card(AV) \times \left(1 + \sum_{i=0}^{N-1} \left(Card(AV)^i \right) \right))$	
NSMR	$Nupdate \times Card(V)$	The best case: 0
		The average case: $Nupdate \times \left(Card(r) + Card(r)^2 \times \left(\sum_{i=1}^{N-1} \left(\sigma^i J^{i-1} \right) \right) \right)$
		The worst case: $Nupdate \times \left(Card(r) \times \left(\sum_{i=0}^{N-1} \left(Card(r)^i \right) \right) \right)$
SMIM	The average case: $Nupdate \times (Card(V) + Card(AV) + Card(\Delta AV) \times \left(2 + Card(AV) \times \sum_{i=0}^{Nav-2} \left(\sigma^{i+1} J^i \right) \right))$ $0 \le Nav \le N$	0
	The worst case: $Nupdate \times (Card(V) + Card(AV) + Card(\Delta AV) \times \left(2 + \sum_{i=1}^{N-1} \left(Card(AV)^i \right) \right))$	

Table 3. Number of Rows Accessed (Part 1 of 2)

Category	Number of rows accessed in data warehouse	Number of rows accessed in data source
NSMIM	$Nupdate / I \times Card(V)$	The best case: 0
		The average case: $Nupdate \times \sum_{k=1}^{N-1}(I^{k-1}\sigma^{k} j^{k-1} Card(U)^{k} \sum_{i=0}^{N-k-1}(\sigma^{i} J^{i}) \times (Card(U) + \sum_{i=0}^{k-2}(\sigma^{i+1} j^{i} Card(U)^{i+2}) + \sigma^{k} j^{k-1} Card(U)^{k} Card(r) \sum_{i=0}^{N-k-1}(\sigma^{i} J^{i})))$
		The worst case: $Nupdate \times \sum_{k=1}^{N-1}(I^{k-1} Card(U)^{k} \sum_{i=0}^{N-k-1}(Card(r)^{i}) \times (Card(U) + \sum_{i=0}^{k-2}(Card(U)^{i+2}) + Card(U)^{k} Card(r) \sum_{i=0}^{N-k-1}(Card(r)^{i})))$

Table 4. Number of Rows Accessed (Part 2 of 2)

In the average case, the total numbers of rows accessed for the not self-maintainable recomputation (NSMR) approach and the self-maintainable recomputation (SMR) approach are very similar. As the same base relations are replicated at the data warehouse in the self-maintainable recomputation approach, actually these data will be accessed twice. The first time is for maintaining the auxiliary views and the second time is for maintaining the target materialized views at the data warehouse. When the cardinality of the base relations/auxiliary views, *Card*(*r*), is small, the number of rows accessed for the self-maintainable recomputation approach is somewhat larger than that of the not self-maintainable recomputation approach (Figure 5). When *Card*(*r*) becomes larger, the difference is small enough to be neglected (Figure 6).

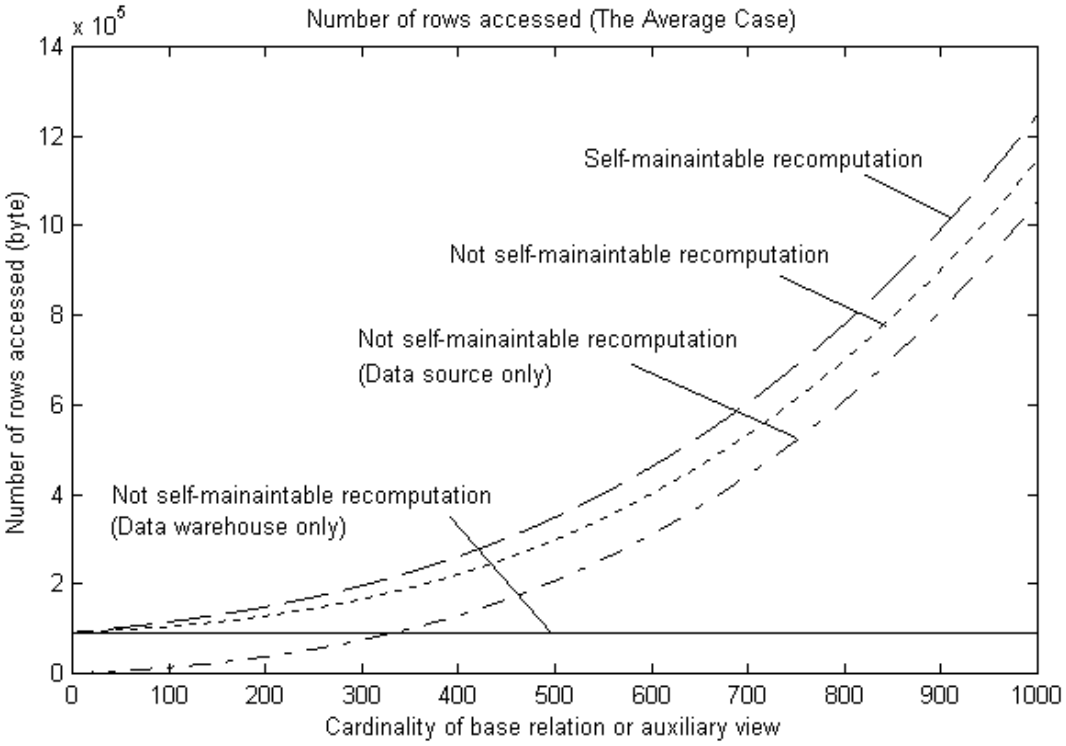

Figure 5. Number of rows accessed in the data warehouse (The average case)

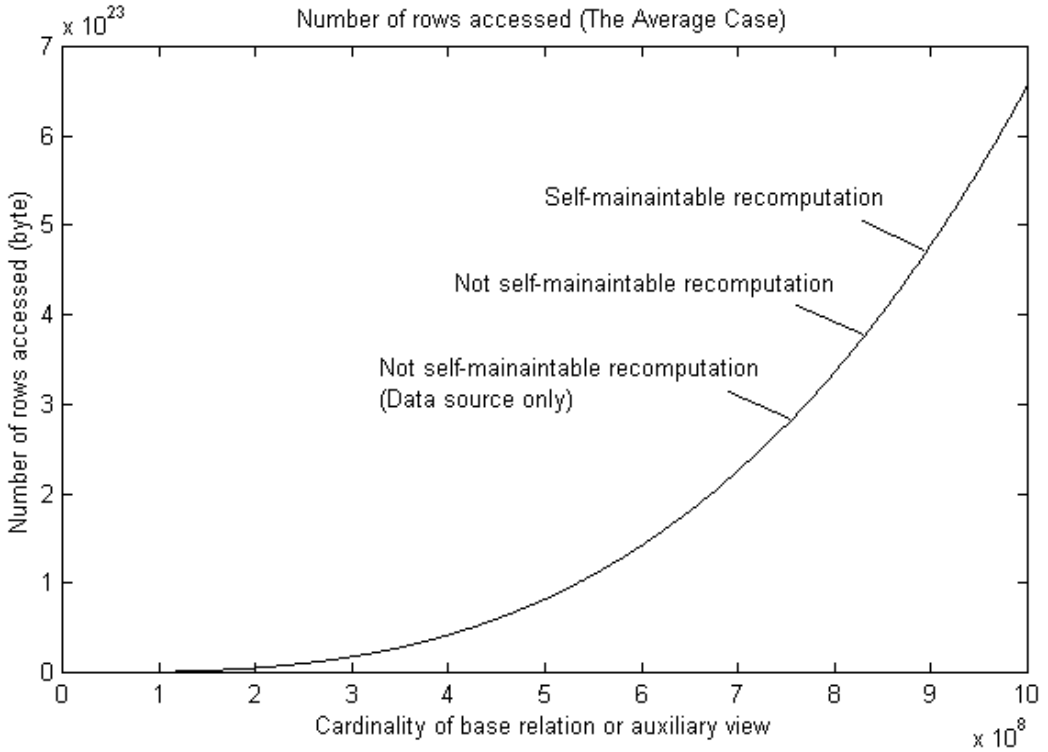

Figure 6. Number of rows accessed in the data warehouse (The average case)

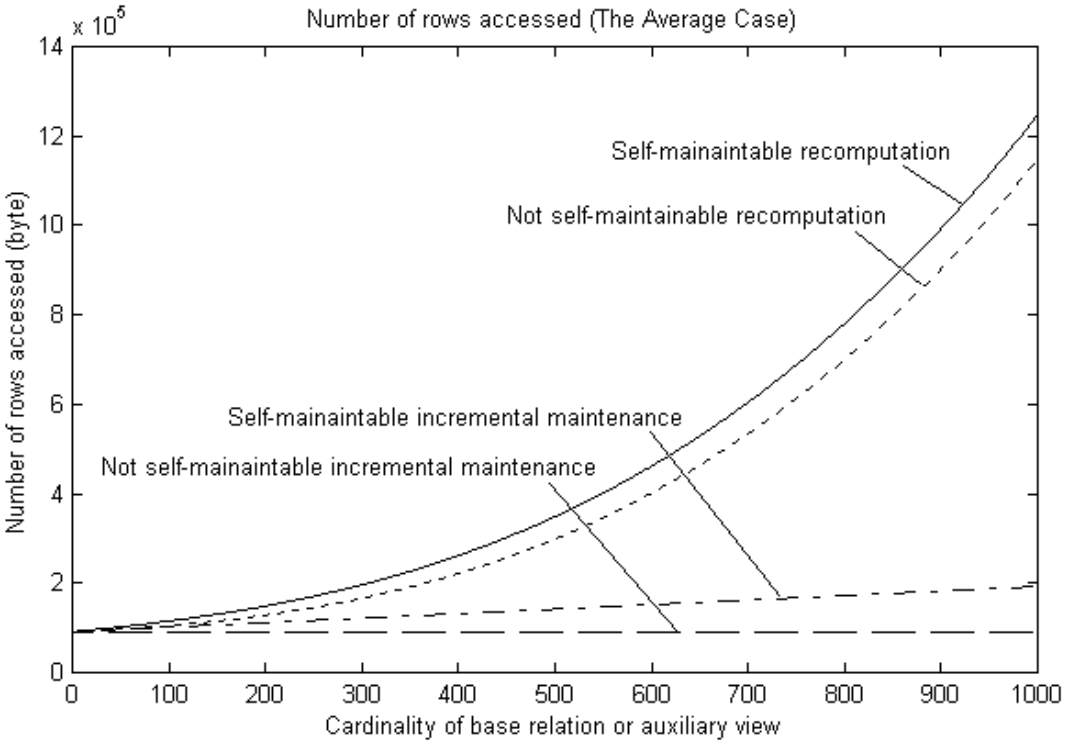

Figure 7. Number of rows accessed in the data warehouse (The average case)

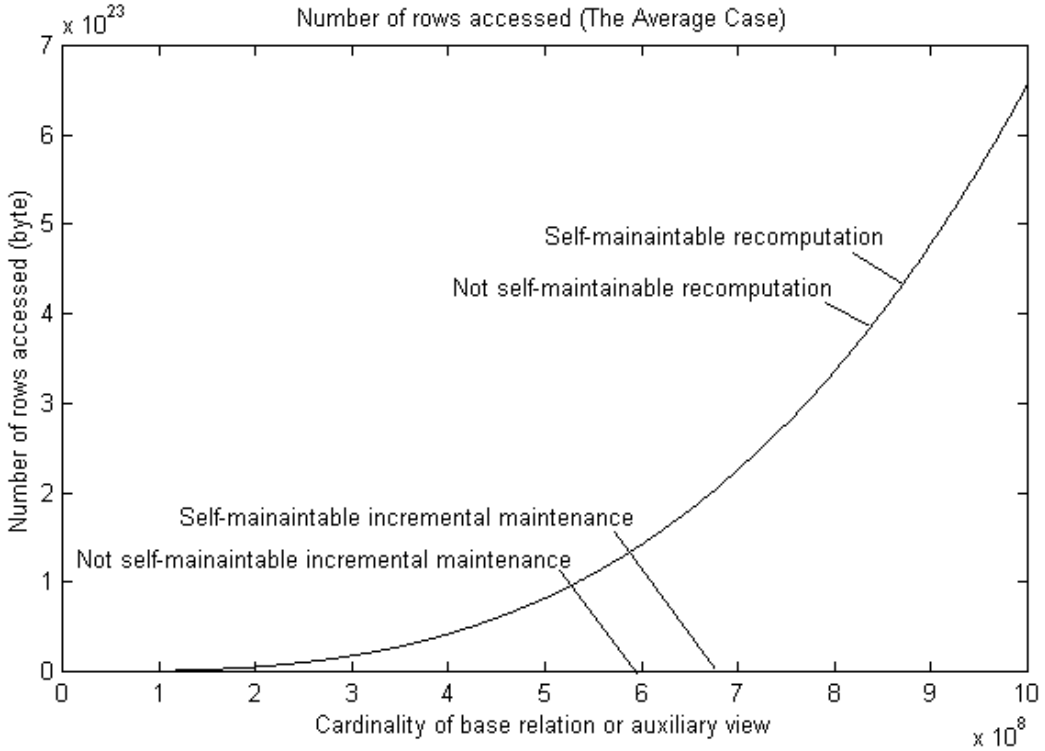

Figure 8. Number of rows accessed in the data warehouse (The average case)

When *Card*(*r*) is small, the numbers of rows accessed for self-maintainable recomputation and not self-maintainable recomputation grow faster than those for self-maintainable incremental maintenance and not self-maintainable incremental maintenance (Figure 7). When *Card*(*r*) becomes very large, the number of rows accessed for both the recomputation approaches are much larger that those of both the incremental approaches (Figure 8). In order to see the differences, we redrew Figure 8 using a logarithmic scale in the y-axis; the results are shown in Figure 9. From Figure 9, we can see that the number of rows accessed in SMR is the same as that in NSMR, but is higher than that of NSIM, which is higher than that of SMIM. We also observe the similar results when varying the values of the select factor σ and join factor j .

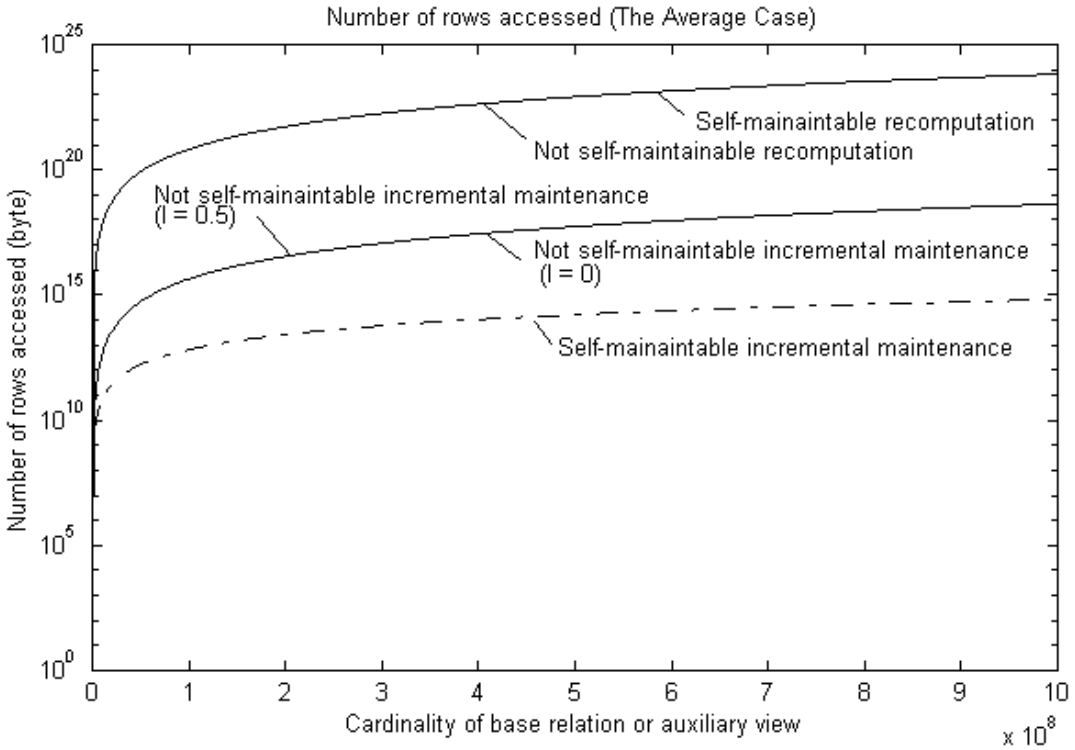

Figure 9. Number of rows accessed in the data warehouse (The average case)

4.0 Conclusions and Future Research

Category	Advantage	Disadvantage
Self-Maintainable Recomputation	-Data warehouse view maintenance operations are totally separated from OLTP operations; -Unavailable source will not block the data warehouse view maintenance process;	-Data are replicated at data warehouse; -Need extra data storage for replicate data; -Have to implement and maintain data transfer processes to transfer data from sources to data warehouse;
Not Self-Maintainable Recomputation	-Very simple to implement; -No replicate data at the data warehouse; -No extra data storage for replicate data; -Do not have to implement and maintain data transfer processes to transfer data from sources to data warehouse;	-Unavailable source will block the data warehouse view maintenance process; -Evaluating queries at the data sources consumes local resources; -Data warehouse view maintenance operations are not separated from OLTP operations;
Self-Maintainable Incremental Maintenance	-Data warehouse view maintenance operations are totally separated from OLTP operations; -Unavailable source will not block the data warehouse view maintenance process; -In the worst case, the number of rows accessed to maintain a view is the lowest;	-Data are replicated at data warehouse; -Need extra data storage for replicate data; -Have to implement and maintain data transfer processes to transfer data from sources to data warehouse;
Not Self-Maintainable Incremental Maintenance	-No replicate data at the data warehouse; -No extra data storage for replicate data; -Do not have to implement and maintain data transfer processes to transfer data from sources to data warehouse;	-Unavailable source will block the data warehouse view maintenance process; -Evaluating queries at the data sources consume local resources; -Data warehouse view maintenance operations are not separated from OLTP operations; -Have to design the view maintenance process carefully to avoid the anomaly problem; -In the worst case the number of rows accessed is the highest; -Performance is down-graded rapidly; -Need extra storage for intermediate data (COLLECT tables);

Table 5. Advantages and disadvantages of the view maintenance techniques

All data warehouse view maintenance techniques can be classified into four major categories. They are self-maintainable recomputation, not self-maintainable recomputation, self-

maintainable incremental maintenance, and not self-maintainable incremental maintenance. Their advantages and disadvantages are listed in Table 5.

Both self-maintainable recomputation and self-maintainable incremental maintenance approaches totally separate the data warehouse view maintenance operations from the OLTP operations. Therefore, the view maintenance operations will not consume data sources' local resources. These operations only consume the data warehouse's resources. Even if the remote data sources are not available, the data warehouse view maintenance process can continue running. However, a part or all source data are replicated at the data warehouse to make the data warehouse view maintenance process self-maintainable. These replicated data take space. Data transfer processes are implemented to transfer data from the remote data sources to the data warehouse. Design, implement and maintain these processes are time-consuming. A lot of unnecessary data may be duplicated at the data warehouse. However, these are the approaches that probably many large companies have to take if they want to separate their data warehouse view maintenance operations from their OLTP operations.

Both the not self-maintainable recomputation and not self-maintainable incremental maintenance approaches suffer from some common disadvantages. As the remote data sources have to process queries from the data warehouse that consume their limited local resources, the OLTP system will be slow. Once a data source is unavailable, the data source will not be able to answer queries sent from the data warehouse in time. It will block the data warehouse view maintenance process. The not self-maintainable incremental maintenance approach has some additional disadvantages. To avoid the anomaly problem, the view maintenance process must be designed carefully. If a lot of updates happen at the data sources, the data warehouse may issue many compensating queries. It is very possible that the data warehouse may never get the final

query results. Both approaches also have some common advantages. As there is no replicate data stored at the data warehouse, no data transfer process has to be implemented and maintained. There is no extra space for storing replicate data. Both approaches are good for small to mid-sized companies whose OLTP database systems are not too busy.

Among all the four categories, self-maintainable incremental maintenance is the best in terms of space used in the data warehouse and number of rows accessed in order to propagate an update to the target materialized view in the data warehouse. As the cost of data storage becomes increasingly low, this is the best approach to implement a data warehouse.

For future work, we plan to consider another important performance measurement called the view refresh time, which is defined as the elapse time from the time the system receives a source update to the time the update is reflected in the view. We will also address the problem of multiple view maintenance in a multiple source environment.

References

[Cui 99] Y. Cui and J. Widom. "Storing Auxiliary Data for Efficient View Maintenance and Lineage Tracing." http://www-db.stanford.edu/pub/papers/auxview.ps Techincal Report, Stanford University, 1999.

[Hammer 95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge, "The Stanford Data Warehousing Project." *IEEE Data Engineering Bulletin*, June 1995.

[Huyn 96a] N. Huyn, "Efficient View Self-Maintenance." *Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7, 1996.

[Huyn 96b] N. Huyn, "Efficient Self-Maintenance of Materialized Views." http://www-db.stanford.edu/pub/papers/vsm-2-tr.ps. Technical Note, 1996.

[Huyn 97a] N. Huyn, "Maintaining Data Warehouse Under Limited Source Access." Ph.D. Thesis, Stanford University, August 1997.

[Huyn 97b] N. Huyn, "Exploiting Dependencies to Enhance View Self-Maintainability." http://www-db.stanford.edu/pub/papers/fdvsm.ps. Technical Note, 1997.

[Huyn 97c] N. Huyn, "Multiple-View Self-Maintenance in Data Warehousing Environments." *Proceedings of the 23rd VLDB Conference*, Athens, Greece, 1997.

[Hull 96] R. Hull and G. Zhou, "A framework for supporting data integration using the materialized and virtual approaches," In *SIGMOD* 1996.

[Labio 97b] W. Labio, D. Quass, and B. Adelberg, "Physical Database Design for Data Warehousing." *Proceedings of the International Conference on Data Engineering*, Binghamton, UK, April, 1997.

[Quass 96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom, "Making Views Self-Maintainable for Data Warehousing." *Proceedings of the Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, December 1996.
[Silberschatz 97] A. Silberschatz, H. F. Korth and S. Sudarshan, Database System Concepts, 3rd. Edition, McGraw-Hill, 1997.
[TPC 99] Transaction Processing Performance Council (TPC), "TPC Benchmark™ R (Decision Support) Standard Specification, Revision 1.0.1." http://www.tpc.org, 1999
[Widom 95] J. Widom, "Research Problems in Data Warehousing." *Proceedings of the 4th International Conference on Information and Knowledge Management* (*CIKM*), November 1995.
[Wiener 96] J. L. Wiener, H. Gupta, W. J. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom, "A System Prototype for Warehouse View Maintenance." *Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 7, 1996, pp. 26-33.
[Zhuge 95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom, "View Maintenance in a Warehousing Environment." *Proceedings of the ACM SIGMOD Conference*, San Jose, California, June 1995, pp 316-327.
[Zhuge 96] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener, "The Strobe Algorithms for Multi-Source Warehouse Consistency." *Proceedings of the Conference on Parallel and Distributed Information Systems*, Miami Beach, FL, December 1996.
[Zhuge 97a] Y. Zhuge, J. L. Wiener, and H. Garcia-Molina, "Multiple View Consistency for Data Warehousing." *Proceedings of the International Conference on Data Engineering*, Binghamton, UK, April, 1997.
[Zhuge 97b] Y. Zhuge, "Whips Performance: Model and Experiments." http://www-db.stanford.edu/pub/papers/perf-tech.ps. Technical Note, December, 1997.
[Zhuge 98a] Y. Zhuge and H. Garcia-Molina. "Performance Analysis of WHIPS Incremental Maintenance." Submitted for publication. September 1998.
[Zhuge 98b] Y. Zhuge and H. Garcia-Molina. "Performance Analysis of WHIPS Incremental Maintenance." Full version of [Zhuge 98a]. September 1998.