Efficient Refreshment of Materialized Views With Multiple Sources

Hui Wang
Dept. of CSEE
University of Queensland
St. Lucia, QLD 4072, Australia
hwang@csee.uq.edu.au

Maria Orlowska
Dept. of CSEE
University of Queensland
St. Lucia, QLD 4072, Australia
maria@csee.uq.edu.au

Weifa Liang
Dept. of Computer Science
Australian National University
Canberra, ACT 0200, Australia
wliang@cs.anu.edu.au

Abstract

A data warehouse collects and maintains a large amount of data from multiple distributed and autonomous data sources. Often the data in it is stored in the form of materialized views in order to provide fast access to the integrated data. However, maintaining a certain level consistency of warehouse data with the source data is challenging in a distributed multiple source environment. Transactions containing multiple updates at one or more sources further complicate the consistency issue.

Following the four level consistency definition of view in a warehouse, we first present a complete consistency algorithm for maintaining SPJ-type materialized views incrementally. Our algorithm speed-ups the view refreshment time, provided that some extra moderate space in the warehouse is available. We then give a variant of the proposed algorithm by taking the update frequencies of sources into account. We finally discuss the relationship between a view's certain level consistency and its refresh time. It is difficult to propose an incremental maintenance algorithm such that the view is always kept at a certain level consistency with the source data and the view's refresh time is as fast as possible. We trade-off these two factors by giving an algorithm with faster view refresh time, while the view maintained by the algorithm is strong consistency rather than complete consistency with the source data.

1 Introduction

The problem of materialized view maintenance has received increasing attention in the past few years due to its application to data warehousing. Traditionally, a view is a derived relation defined in terms of source relations. A view is said to be *materialized* when its content is stored in a specific database (a data warehouse), rather than computed whenever required from the source databases. Data warehouses usually store materialized views in order to provide fast access to the data that is integrated from several distributed and autonomous data sources [11, 21]. The data sources may be heterogeneous and remote from the warehouse. A data warehouse can be used as an integrated and uniform basis for decision support, data mining, data analysis, and ad-hoc querying across the source data. In the study of data warehousing, one important issue is the view maintenance problem which basically is to keep the contents of the materialized views at a certain level consistency with the contents of the source data when any update commits at the sources. Consequently, the problem of maintaining materialized views in a data warehouse with multiple remote sources is different from the traditionally centralized view maintenance problem.

1.1 Related work

Many incremental maintenance algorithms for views have been introduced for centralized database systems [2, 3, 5, 9, 10, 7]. There are also a number of researches investigating similar issues in distributed environments [6, 11, 17, 20, 21, 22]. These previous works have formed a spectrum of solutions ranging from a fully virtual approach at one end where no data is materialized and all user queries are answered by interrogating the source data [12], to a full replication at the other end where the whole data in the source data is copied to the warehouse so that the updates can be handled locally in the warehouse without consulting the sources [8, 14, 12, 13]. The two extreme solutions are inefficient in terms of communication and query response time in the former case, and storage space in the latter. A more efficient solution would be to materialize some relevant subsets of source data in the warehouse (usually the query answer), and relevant updates at sources are propagated to the warehouse. The warehouse then refreshes the materialized data incrementally against the updates. However, in a distributed environment, this approach may necessitate a solution in which the warehouse contacts the sources for additional information to ensure the update correctness, i.e., keep the data in the warehouse at a certain level of consistency with the source data [20, 21, 6, 1, 11]. Since we are dealing with maintaining a materialized view, then when and how to update the view (i.e., the view refreshment) is a crucial issue. In most commercial warehousing systems, updates to the source data are usually queued and propagated periodically to the warehouse in a large batch update transaction called maintenance transactions. In order to maintain the view data consistent with the source data, the approach most commonly used in commercial warehousing systems for guaranteeing consistency without blocking is to maintain the warehouse at night time when the warehouse is not available to users, while the user query can be processed in the day time when the maintenance transactions are not running. Unfortunately there are two major problems with this method [16]. As corporations become globalized there is no longer any night time common to all corporate sites during which it is convenient to make the warehouse unavailable to users. Since the maintenance transaction must be finished by the next morning, the time available for view maintenance can be a limiting factor in the number and size of views that can be materialized at the warehouse. Therefore, the incremental view maintenance is a more and more acceptable method for view refreshment. In [12], they present an analytical model of view fresh time for virtual, fully materialized, and partial materialized views, and give some preliminary results. However, their experiments are based on a local network, so, the performance behavior of their algorithm does not reflect the real scenario for this problem in a wide area network. In [16] they also consider improving the view refresh time using extra space, i.e., each tuple in a view has two versions, and each update is attached a timestamp. Recently [15] study the view refresh problem again by giving an algorithm based on timestamps, but the algorithm essentially proceeds with periodic updates, not on-line incremental updates which we will consider.

In this paper we study a specific view – SPJ (select-project-join) type view maintenance and its refresh time. This is the first time that we show how to speed-up a view's refresh time without compromising its consistency with the source data, provided that some extra moderate warehouse storage space is available. Due to limited space, all proof details are omitted in this paper.

1.2 Our contributions

In this paper we first consider the incremental maintenance of SPJ-type materialized views by presenting a complete consistency algorithm for it. The algorithm speed-ups the view refresh time, provided that some extra space in the warehouse is available. We then give a variant of the proposed algorithm, which takes the sources' update frequencies into account. As a result, the proposed algorithm has a better view refresh time on average, compared to that given by our first algorithm. We also discuss the relationship between a certain level consistency of a view and its refresh time. It is usually difficult, in a multiple source environment, to propose an incremental maintenance algorithm such that the view is always kept at a certain level consistency (e.g. complete consistency) and the view refresh time is as small as possible. Through incorporating the parallelism, we finally present an algorithm which, in most cases, provides a better view refresh time than our first complete consistency algorithm, but the resulting view is only kept in strong consistency, rather than complete consistency.

1.3 Paper outline

The rest of the paper is organized as follows. Section 2 introduces the level consistency concepts and the data warehouse model. Section 3 presents the view consistency issue which has arisen in multiple distributed sources and briefly reproduces two previously known complete consistency algorithms [21, 1] for our later use. Section 4 devises a complete consistency algorithm which is used to improve the refresh time. A variant of the algorithm, which takes the source update frequency into account, is also presented in this section. Section 5 shows how to improve the view refresh time further, by lowering the view certain level consistency with the source data. A strong consistency algorithm is given for this purpose.

2 Preliminaries

2.1 The Data Warehouse Model

A *data warehouse* is a repository of integrated information from distributed, autonomous and possibly heterogeneous sources. Here we adopt a typical data warehouse architecture defined in [18]. Figure 1 illustrates the basic components of a data warehouse. At each source a monitor/wrapper collects the data of interest and sends it to the warehouse. The monitor/wrapper are responsible for identifying changes, and notifying the warehouse. If a source provides relational-style triggers, the monitor may simply pass on information. On the other hand, a monitor for a legacy source may need

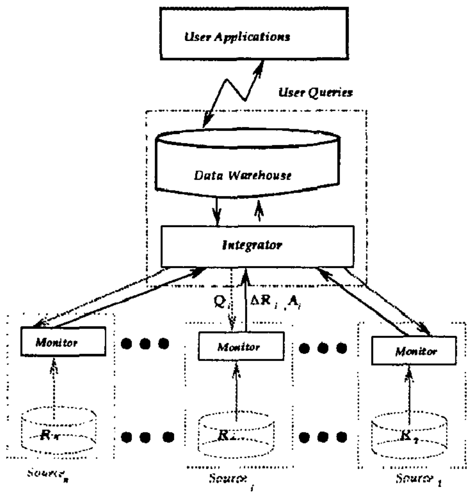

Figure 1: Data warehouse architecture

to compute the difference between successive snapshots of the source data. At the warehouse, the integrator receives the source data, performs necessary data integration and translation, adds any extra desired information such as the timestamps for historical analysis, and requests the warehouse to store the data.

The communication between a source and the data warehouse is assumed to be reliable FIFO, i.e., messages are not lost and are delivered in the order in which they originally are sent. Additionally, there is no any assumption about the communication among the different sources. In fact, they may be completely independent, and may not be able to communicate with each other. Further, each source action, and the resulting message sent to the warehouse, is considered as one *event*, and events are atomic.

It is easy to observe that the design complexity of consistent warehouse algorithms is closely related to the scope of transactions at the sources. In this model we classify the update transactions into the following three categories: (i) Single update transactions where each update is executed at a single source. Thus, single update transaction comprises its own transaction only and is reported to the warehouse separately. (ii) Source local transactions where a sequence of updates are performed as a single transaction. Therefore the goal is to reflect all of these actions atomically at the warehouse. Note that all of these updates happen in a single source. (iii) Global transactions where the updates involve multiple sources. We assume that there is a global serialization order to the global transactions. The goal here is to reflect the global transactions atomically at the warehouse. In this paper we assume that the updates being handled at the warehouse are of types 1 and 2. The approach described in [21] can be used to extend the proposed algorithms for type 3 updates.

Let V be a SPJ-type view derived from n relations $R_1, R_2, \ldots, R_n$ at corresponding data sources, defined as follows.

$$V = \pi_X \sigma_P (R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n) \qquad (1)$$

where X is the set of projection attributes and P is the selection conditions. The updates to the source data are assumed to be either inserts or deletes of tuples. A modify operation will be treated as a delete followed by an insert. We assume that all views in the warehouse are based on bag semantics, i.e., the multiplicity of a tuple in a view is maintained in terms of a control field that maintains the occurrence of that tuple [10]. We finally assume that each tuple in the materialized view has an additional *count* value which indicates in how many ways the same tuple can be derived from the source data by the view definition.

3 Consistency Concept

Let each warehouse state ws represent the content of the data warehouse. The warehouse state changes whenever one of its views is updated. Let $ws_0, ws_1, \ldots, ws_f$ be a sequence of the warehouse states after a series of updates. Consider a view V at the warehouse which has been defined by expression (1). Let $V(ws_j)$ be the content of V at state ws_j. A source state ss_j is a vector that contains n components, where each component represents the state of a source at a given time. The ith component, $ss_j[i]$ is the state of source i which represents the content of source i. We further assume that source updates are executed in serializable fashion across sources, and ss_q is the final state of the sources. $V(ss_j)$ is the result of computing the view V over the source state ss_j. That is, for each relation R at source i which contributes to V, $V(ss_j)$ is evaluated over R at the state $ss_j[i]$.

In [20], they define the following four-level consistency of a warehouse view with its source data. Assume that V at the warehouse is initially synchronized with the source data, i.e., $V(ws_0) = V(ss_0)$.

1. **Convergence** For all finite executions, $V(ws_f) = V(ss_q)$ where ws_f is the final state of the warehouse. That is, the view is finally consistent with the source data after the last update and all activities have been ceased.

2. **Weak consistency** Convergence holds and, for all ws_i, there exists a source state ss_j such that $V(ws_i) = V(ss_j)$. Furthermore, for each source x, there exists a serial schedule $R = T_1, T_2, \ldots, T_k$ transactions such that there is a locally serializable schedule at source x that achieves that state.

3. **Strong consistence** Convergence holds and there exists a serial schedule R and a mapping m from the warehouse states to the sources states with the following properties: (i) Serial schedule R is equivalent to the

actual execution of transactions at the source. (ii) For all ws_i, $m(ss_i) = ss_j$ for some j and $V(ws_i) = V(ss_j)$ (iii) if $ws_i \prec ws_k$, them $m(ws_i) \prec m(ws_k)$ where $\prec$ is a precedence symbol. That is, each warehouse state reflects a set of valid source states.

4. **Completeness** The view in the warehouse is strong consistent with the source data, and for every ss_j, there exists a ws_i such that $m(ws_i) = ss_j$. That is, there is a complete order preserving mapping between the states of the view and the states of the sources.

As said in [21], complete consistency is a nice property since it claims that the view is always consistent with the source data in every state. However, we believe that this restriction may be too strong in practice. In some cases, the convergence may be sufficient even if there is some invalid intermediate states. In most cases, the strong consistency which corresponds to the batch updates is desirable.

3.1 Consistency maintenance in incremental view computation

It is well understood that updating a view incrementally in response to the source updates is a common practice. In the centralized data warehouse, the view, as well the source data, is stored in the same machine, and each update transaction is carried out atomically. Thus, the view is always kept completely consistent with the source data. There are many efficient algorithms for this purpose [2, 7, 10, 9]. However, in the distributed environment, the situation is totally different; the source update is sent to the warehouse through messages, thus a communication overhead is involved in each of these operations. Also, the updates in the sources are normally independent even there are concurrent updates among the sources. As a result, the inconsistency occurs quite often. Below is an example adopted from [1] to illustrate the inconsistency problem.

Assume that a view V is generated by joining two relations R_1 and R_2, and R_1 and R_2 are located in different databases, i.e., $V = R_1 \bowtie R_2$. Assume that an insertion update ΔR_1 committed in source 1, then the updated view should be as follows.

$$(R_1 \cup \Delta R_1) \bowtie R_2 = (R_1 \bowtie R_2) \cup (\Delta R_1 \bowtie R_2)$$

Although the data in $R_1 \bowtie R_2$ are already in the data warehouse, we need to compute $\Delta R_1 \bowtie R_2$ in order to update V. In doing so, the data warehouse issues a query $Q_1 = \Delta R_1 \bowtie R_2$ to source 2. Source 2 sends the answer A_1 of Q_1 back to the warehouse after evaluating Q_1. Then, the warehouse incorporates A_1 to the materialized view V, and V is updated. This approach works well as long as two consecutive source updates are far from each other. Otherwise, the problem becomes more complicated. Now we assume that a concurrent update ΔR_2 issued and committed at source 2, and source 2 sends the update to the warehouse. Thus, there are two warehouse states ws_1 and ws_2, corresponding to the two updates. Assume that ΔR_1 is prior to ΔR_2. If we want V to be in complete consistency with the source data, then $V(ws_1) = (R_1 \cup \Delta R_1) \bowtie R_2$ and $V(ws_2) = (R_1 \cup \Delta R_1) \bowtie (R_2 \cup \Delta R_2)$ after the two updates. Now let us see what may happen in the distributed environment. Assume that the update ΔR_2 is committed at source 2 just before Q_1 is committed. In other words, R_2 now becomes $R_2 \cup \Delta R_2$. Then, the evaluating result of Q_1 at source 2 becomes $A_1' = \Delta R_1 \bowtie (R_2 \cup \Delta R_2) = (\Delta R_1 \bowtie R_2) \cup (\Delta R_1 \bowtie \Delta R_2)$, i.e., after the evaluation of Q_1, $V(ws_1) = A_1'$. Obviously A_1' is an incorrect answer for the update ΔR_1. The error occurs because $\Delta R_1 \bowtie \Delta R_2$ is included, which is called the contaminated data. In order to remove the effect of the update ΔR_2 to the update ΔR_1, different approaches to deal with the contaminated data will lead to different incremental maintenance algorithms.

3.2 Previous complete consistency algorithms

[20, 21] present a nice solution for the contaminated data problem. In [20] they first mention the problem by proposing the ECA algorithm (ECA stands for Eager Compensation Algorithm). However, their algorithm is only applicable to the case where there is only a single remote source. [21] extend their work by proposing a Strobe algorithm for the multiple source case by adding more constraints on the view and the source relations. The basic principle behind both ECA and Strobe is an eager compensation method which is illustrated below. Following the previous example, assume that there are two concurrent updates ΔR_1 and ΔR_2 and ΔR_1 is prior to ΔR_2. Source 2 evaluates Q_1 as usual. However, ΔR_2 must be received by the warehouse prior to A_1 (due to the FIFO assumption at each source), then, the warehouse sends its second query $Q_1^1 = \Delta R_2 \bowtie \Delta R_1$ to source 2 for compensation to remove the effect of ΔR_2. Thus, after the answer A_1^1 of Q_1^1 returns to the warehouse and there is no further updates, V is updated and $V = V \cup (A_1 - A_1^1)$ which is exactly $V(ws_1)$. Note that Strobe only ensures strong consistency but not complete consistency since it incorporates the effects of several updates collectively. Also, the view can be updated only at the moment when there is no further updates from the sources. They call this moment a *quiescence* state. If there is no such quiescence state, the view V would never be updated. To remove quiescence requirement and make the algorithm for complete consistency, [21] present a variant of Strobe, C-Strobe which handles each update completely before handling subsequent updates, i.e., an answer to a given update is evaluated by consulting all the sources. Due to concurrent updates, the answer may contain contaminated

data. Further queries are generated to compensate for correction, and further contamination may arise due to concurrent updates during the compensation queries. They show that, if there are at most μ concurrent updates that can arrive between the time a query committed and its answer to be incorporated to the view, at most μ^{n-2} queries need to be sent for a single update.

[11, 12] consider the incremental maintenance of hybrid views by mentioning that the above compensation can actually be carried out locally without consulting the sources because the warehouse contains all necessary information. Later [1] apply this idea by presenting a linear, complete consistency algorithm SWEEP. In their method, the warehouse does not send compensation queries to the sources (e.g. in our previous example, instead sending Q_1^1 to source 2, Q_1^1 will be evaluated at the warehouse), and the evaluation for the queries can be done locally. Compared with C-Strobe, SWEEP is a considerable improvement because it removes some requirements which are needed in C-Strobe, such as, all the keys of the relations are stored in V. Most important, SWEEP takes $O(n)$ time to respond to each update, and the number of messages sent to sources is $O(n)$, in the worst case, is in contrast to the exponential number of messages needed by C-Strobe in terms of the number of updates. Here we briefly sketch the SWEEP algorithm for later use. Assume that the warehouse is evaluating an incremental query resulting from ΔR_i, it may receive a concurrent update ΔR_j, $j < i$. As previously discussed, when the query answer arrives from source R_j, it is $(R_j \cup \Delta R_j) \bowtie R_{j+1} \bowtie \ldots \bowtie R_{i-1} \bowtie \Delta R_i$ instead of $R_j \bowtie R_{j+1} \bowtie \ldots \bowtie R_{i-1} \bowtie \Delta R_i$ that we expected. The contaminated data is $\Delta R_j \bowtie R_{j+1} \bowtie \ldots \bowtie R_{i-1} \bowtie \Delta R_i$, which can be evaluated at the warehouse locally since both ΔR_j and $R_{j+1} \bowtie \ldots \bowtie R_{i-1} \bowtie \Delta R_i$ are available at the warehouse. Note that the term $R_{j+1} \bowtie \ldots \bowtie R_{i-1} \bowtie \Delta R_i$ is the partially evaluated answer from R_{j+1} for the query initiated on account ΔR_i. The case of a concurrent update ΔR_j, $j > i$, is symmetric. After all sources answered, the final answer is included to the view. The detail about SWEEP can be seen from [1].

4 A New Complete Consistency Algorithm

4.1 The view refresh time

Let V be a materialized view in a warehouse, and for any two consecutive warehouse states ws_i and ws_{i+1}, assume that ws_i is obtained at time t_0 and ws_{i+1} is obtained at time t_1. Then, the refresh time of V from ws_i to ws_{i+1} is $t_1 - t_0$. During this time interval, V undergoes a state change from ws_i to ws_{i+1}, while the source states may undergo many source state changes. If V is complete consistent with the source data, then by the consistency definition, each warehouse state has a corresponding source state. Let ws_i be the current warehouse state corresponding to a source state ss_j, $i \leq j$. By our previous discussion, in order to implement incremental updating V to the state ws_{i+1} due to an update in some sources, a number of round communications between the warehouse and the sources is needed. Assume that the warehouse needs to send N queries $Q_{i_1}, Q_{i_2}, \ldots, Q_{i_N}$ to the various sources, and to receive the N answers $A_{i_1}, A_{i_2}, \ldots, A_{i_N}$ from the sources. To keep V completely consistent with the source data, the refresh time of V from ws_i to ws_{i+1} is

$$\begin{aligned} T_{refresh} &= \sum_{l=1}^{N}(T_{to\_source}(Q_{i_l}) + T_{to\_house}(A_{i_l}) \quad (2) \\ &\quad + T_{h\_join}(A_{i_l}) + T_{s\_join}(Q_{i_l})) \end{aligned}$$

where the terms in expression 2 are defined as follows. $T_{to\_source}(Q_{i_l})$ is the transmission time to send query Q_{i_l} from the warehouse to source i_l through the network; $T_{to\_house}(A_{i_l})$ is the transmission time from source i_l to the warehouse by returning the query answer; $T_{h\_join}(A_{i_l})$ is the time for processing the join of A_{i_l} and the concurrent updates received so far, which is carried out at the warehouse. $T_{s\_join}(Q_{i_l})$ is the time for processing query Q_{i_l} at source i_l, which is carried out at source i_l.

To simplify our discussion, we assume that the transmission time between the warehouse and the sources is symmetric and fixed, denoted by $\bar{t}$. We also assume that the query processing time at either the warehouse or the sources is negligible. We argue that there are at least three reasons to make this second assumption. Firstly, the processing time only takes a very small proportion of the entire refresh time, compare to the transmission time in a wide area network. Secondly, the processing time varies, depending on the sizes of sources and the view. Finally, we assume that the amount of incremental maintenance of a materialized data is relatively small, so the corresponding query processing time is relatively small too. After the above simplifications, expression 2 can be rewritten as

$$T_{refresh} = 2N\bar{t},$$

which depends on N - the number of queries sent by the warehouse.

4.2 The refresh time of previous algorithms

We now analyze the view refresh time of the two previously known complete consistency algorithms, based on the above model.

First, let us look at the SWEEP algorithm due to [1]. In order to proceed with the incremental maintenance to V, the warehouse needs to communicate with all other sources except the update source to get further information. The compensation process to remove the effect of subsequent concurrent source updates to the current update is carried out in the warehouse. Thus, by the SWEEP algorithm, the view refresh time is $(2n-2)\bar{t}$, because the warehouse needs to send $N = n-1$ queries and to receive the answers of the N queries from the sources, which takes $(2n-2)\bar{t}$ time in total.

Then, let us analyze the view refresh time of the C-Strobe algorithm due to [21]. Based on the similar reasoning as above, the view refresh time of C-Strobe is $\max\{2\mu^{n-2}\bar{t}, (2n-2)\bar{t}\}$ where μ is the number of concurrent updates. Clearly C-Strobe requires sending $N = \mu^{n-2}\bar{t}$ queries to the sources in the worst case.

4.3 A complete consistency algorithm based on equal partitioning

In this section we present an algorithm which not only reduces a view refresh time but also maintain the complete consistency of the view, provided that some extra warehouse space is available. Our approach can be described as follows. Let V be a materialized view derived from n relations corresponding to n sources. Assume that p is a given integer. We first partition the n source relations into $K = \lceil n/p \rceil$ disjoint groups, and each group consists of p relations except the last group which contains $n - p\lfloor n/p \rfloor$ relations. We call such a partition as *equal partition.* Without loss of generality, assume that the first p relations form the first group, the second p relations form the second group, and the last $n - p\lfloor n/p \rfloor$ relations form the Kth group.

By the definition of V, $V = \pi_X \sigma_P (R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)$, where X is the projection attribute set, and P is the selection condition which is the conjunction of disjunctive clauses. Now, we define an auxiliary view V_k for each group k, $1 \le k \le K$, as follows. When k is in $\{1, 2, \ldots, K-1\}$, V_k is defined as

$$V_k = \pi_{X(k)} \sigma_{P(k)} (R_{p(k-1)+1} \bowtie R_{p(k-1)+2} \bowtie \ldots \bowtie R_{pk}),$$

where $X(k)$ is an attribute set in which an attribute is in either X or P and all these attributes only come from relations $R_{p(k-1)+1}$ to R_{pk}, and $P(k)$ is a maximal subset of clauses of P in which the attributes of each clause are only from $R_{p(k-1)+1}$ to R_{pk}. The last group K, V_K is defined as follows.

$$V_K = \pi_{X(K)} \sigma_{P(K)} (R_{p(K-1)+1} \bowtie R_{p(K-1)+2} \bowtie \ldots \bowtie R_n)$$

Thus, V can be rewritten in semantically equivalent form, in terms of the auxiliary views defined as above, $V = \pi_X \sigma_P (V_1 \bowtie V_2 \bowtie \ldots \bowtie V_{K-1} \bowtie V_K)$.

Our main idea is based on the following two points.

- view V and auxiliary views V_k are materialized for any k, $1 \le k \le K$.
- The warehouse is capable to decide to which auxiliary view an update belongs and to which auxiliary view a query answer (from sources) belongs.

Having the above setting, let us consider the incremental maintenance of V due to an update ΔR_i in source i, assuming that R_i is in group k. We proceed with the view maintenance of V_k, using the SWEEP algorithm. As a result, an updated view $V_k = V_k \cup \Delta V_k$ is obtained. Having ΔV_k, we can proceed the update of V locally without consulting the sources. Let V^{new} be the V after the update, then $V^{new} = V \cup \Delta V$, where

$$\Delta V = \pi_X \sigma_P (V_1 \bowtie V_2 \bowtie \ldots \bowtie V_{k-1} \bowtie \Delta V_k \bowtie \ldots \bowtie V_K).$$

Thus, ΔV can be obtained by local evaluation because each V_j is materialized in the warehouse, $1 \le j \le K$ and $j \ne k$. Note that, during evaluating V_k, all the other auxiliary views are locked, in order to keep V in complete consistency with the source data, otherwise the data in other auxiliary views that will be used later for the maintenance of V may be contaminated. It is not hard to show that the above proposed algorithm guarantees the complete consistency.

Theorem 1 *There is an algorithm which maintains the complete consistency of the view in the warehouse and makes the view's refresh time shorter, provided that some additional warehouse storage space is available, which is used for storing auxiliary views.*

Now we analyze the view refresh time of the presented algorithm. It is easy to see that the refresh time of V is $T_{view\_update\_W} = 2(p-1)\bar{t} + t_{WH}$ where t_{WH} is the local update time used to update V in the warehouse by joining ΔV_k and all the other V_j with $j \ne k$, which is a function of joining $\lceil n/p \rceil$ auxiliary views. From $T_{view\_update\_W}$, we can see that the choice of the parameter p is very important. If p is chosen too small, although in this case the communication time between the warehouse and the sources can be reduced dramatically, the number of auxiliary views stored in the warehouse $\lceil n/p \rceil$ will increase, which means more space for the auxiliary views required. Also, the overhead of updating V in the warehouse t_{WH} will increase because more joining relations are involved. Otherwise, there is no any substantial improvement of the view refresh time despite extra warehouse space spent. Compared with the SWEEP algorithm, our algorithm reduces the view refresh time by a factor of $\frac{n}{p}$. The expense for this improvement is obviously the consumption of extra warehouse storage space.

4.4 A partitioning algorithm based on the source update frequencies

We now consider how to speed-up view maintenance with the assumption that not every source has the same update frequency. Let f_i be the update frequency of source data R_i, $1 \leq i \leq n$. Without loss of generality, we assume $f_i \geq f_{i+1}$, i.e., the frequency sequence $f_1, f_2, \ldots, f_n$ is a descending sequence. Based on the update frequencies, we partition the n relations dynamically into K groups such that the sum of update frequencies of each group are roughly equal. This is described as follows.

```
Procedure F_Parti($n, K$);
$k := 1$; /* The current group. */
$i := 0$; /* the index of a relation */
$F_k =: 0$; /* the sum of update frequencies in group $k$ */
$G_k := \emptyset$; /* the $k$th group */
$M := \lfloor \sum_{j=1}^{n} f_j / K \rfloor$;
$test := true$; /* a Boolean variable */
while $test$ do
    $i' := i$;
    repeat
            $i := i + 1$; $G_k := G_k \cup \{R_i\}$;
            $F_k := F_k + f_i$;
    until $(F_k > M)$ or $(i = n)$;
    $G_k$ consists of tables indexed from $i' + 1$ to $i$;
    if      $i < n$ then
            $k := k + 1$; $F_k := 0$; $G_k := \emptyset$;
            else    $test := false$;
    endif;
endwhile.
```

For each group k obtained by algorithm F_Parti, an auxiliary view V_k for V is derived and materialized in the warehouse. We apply the algorithm developed in the previous section for incremental updating V_k if there is a source update coming from one member in group k. Using the update result ΔV_k of V_k and the contents of the other auxiliary views, the update to V is then carried out in the warehouse.

Assume that there are K groups generated by algorithm F_Parti, $G_1, G_2, \ldots, G_K$. Denote by $|G_k|$, the number of relations in G_k, $1 \leq k \leq K$. Then,

Lemma 1 *Assume that G_k is generated by* F_Parti, *then $|G_k| \leq |G_{k+1}|$ for any k with $1 \leq k \leq K-2$.*

Lemma 1 implies that if a source has a higher update frequency, the number of relations in the group of that source participated is small. By our previous discussion, we know that the refresh time of a view depends on its auxiliary view refresh time, while the latter is determined by the number of relations in the group that forms the auxiliary view. Therefore, the refresh time of an auxiliary view in a higher update frequency group is smaller, compared with the refresh time of an auxiliary view in a lower update frequency group. This will finally lead to a better refresh time of a view derived from those auxiliary views. We state this by the following lemma.

Lemma 2 *For any two updates U_i and U_j, let U_i and U_j come from groups k and k', and let t_i and t_j be the view refresh times respectively due to these two updates. If $k < k' < K$, then, $t_i \leq t_j$, which means, an update in a higher update frequency group will have shorter view refresh time.*

By Lemma 2, we derive that the view refresh time is various rather than fixed, depending on from which group the updates come. Therefore, the average view refresh time $\overline{T}_{refresh}$ in terms of n source updates is as follows.

$$\overline{T}_{refresh} = \sum_{i=1}^{n} f_i t_i / n \tag{3}$$

where $0 \leq f_i < 1$ and $\sum_{i=1}^{n} f_i = 1$ and t_i is the view refresh time to response an update from source i.

5 A Strong Consistency Algorithm

In this section we discuss the relationship between the view's freshness and the view's consistency with the sources. We consider improving the view refresh time by sacrificing the view's complete consistency through several approaches such as: increasing the parallelism, and adding the key attributes of source relations to the view and the auxiliary views.

5.1 Speed-up view refresh time through parallelism

Recall that the source relations from which the materialized view V is derived are grouped by some measure, e.g., based on either the equal partition or based on the source update frequency partition. Also, a corresponding auxiliary view for each group is materialized in the warehouse.

We now consider the incremental maintenance of V by showing how to improve the refresh time of V with the expense of lowering its consistency level. We show that the parallelism can be used to speed-up V's refresh time.

The algorithm proceeds in two phases. In the first phase, each auxiliary view proceeds its incremental maintenance independently, using the SWEEP algorithm. In the second phase, the update to V is carried out in the data warehouse, based on the update information supplied by the auxiliary views. We explain why this approach can improve the refresh time of V further.

Assume that there are r consecutive updates in different sources (corresponding r consecutive source states). If the r updates come from the relations in the same group, then there is no any difference from our previous algorithm, that is, V is always completely consistent with the source data at each warehouse state, the view is refreshed to the up-to-date state after $2(p-1)\bar{t}r$

time. Otherwise, the r source updates come from the relations in r different groups. By the above approach, in response to these r updates, each auxiliary view proceeds its update independently. Then, updating V takes $2(p-1)\bar{t}$ time only (assuming that each group contains p relations). Compared with the $2(p-1)\bar{t}r$ time algorithm in the previous section, this new approach improves the view refresh time by a factor of between 1 to r.

According to the level consistency definition, the proposed algorithm does not provide complete consistency, because for some source states (from the source state of the first source update to the source state of the $(r-1)$-th source update), there do not exist the corresponding warehouse states. But we can see that the warehouse state will be consistent with the source state after the warehouse finished the processing of the r source updates. Therefore, the proposed algorithm provides a strong consistency. We have proven this by the following theorem.

Theorem 2 *The proposed algorithm provides a strong consistency.*

5.2 Improving view refresh time by adding sources' keys

We briefly outline another possible approach to improve view refresh time. [19, 4] demonstrate that the incremental maintenance time of a view can be reduced dramatically if the key attributes of the source relations are incorporated into the view definition. Note that the cost for this benefit is that more space is needed in the warehouse. Now we make use of this concept for our case.

Assume that the relations are grouped, based on their update frequencies. Each group then forms an auxiliary view, which is used for the incremental maintenance of the view locally. We treat those groups with higher update frequencies as *special groups*. For those auxiliary views which correspond to special groups, we index the tuples and/or incorporating the identifiers and/or the keys of the tuples of source relations from which they are derived to the views. Thus, the refresh times for those auxiliary views will be reduced, and this will finally lead to the view maintenance time reduction.

In practice, in order to maintain different level consistency and satisfy different refresh time requirements, we may mix some or all the techniques introduced so far in the design of efficient algorithms for incremental view maintenance.

6 Conclusions

In this paper we have considered the view refresh time and the view consistency issue. We have shown that it is possible to reduce the view refresh time and maintain the view completely consistent with the source data, using additional warehouse space. Also, if consistency level of a view is allowed to be lowered, then its refresh time can be further improved. This reflects a trade-off between the view refresh time and the view consistency.

References

[1] D. Agrawal et al. Efficient view maintenance at data warehouses. *Proc. of ACM-SIGMOD Conf.*, 1997, 417–427.

[2] J.A. Blakeley et al. Efficiently updating materialized views. *Proc. of ACM-SIGMOD Conf.*, 1986, 61–71.

[3] J. A. Blakeley et al. Updating derived relations: detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems*, 14(3),1989, 369–400.

[4] L. Bakgaard and N. Roussopoulos. Efficient refreshment of data warehouse views. TR3642, DCS, Univ. of Maryland at College Park, Dec., 1996.

[5] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. *Proc. of the 17th VLDB Conf.*, 1991, 577–589.

[6] L. Colby et al. Algorithms for deferred view maintenance. *Proc. of ACM-SIGMOD Conf.*, 1996, 469–480.

[7] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. *Proc. of ACM-SIGMOD Conf.*, 1995.

[8] A. Gupta et al. Data integration using self-maintainable views. *Proc. 4th Int'l. Conf. on Extending Database Technology*, 1996.

[9] A. Gupta and I. Mumick. Maintenance of materialized views: problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 1995, 3–18.

[10] A. Gupta et al. Maintaining views incrementally. *Proc. of ACM-SIGMOD Conf.*, 1993, 157–166.

[11] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. *Proc. of ACM-SIGMOD Conf.*, 1996, 481–492.

[12] R. Hull and G. Zhou. Towards the study of performance trade-offs between materialized and virtual integrated views. *Proc. of Workshop on Materialized Views: Techniques and Applications*, Montreal, 1996, 91–102.

[13] N. Huyn. Efficient view self-maintenance. *Proc. of the 23rd VLDB Conf.*, Athens, 1997, 26–35.

[14] D. Quass et al. Making views self-maintainable for data warehousing. *Proc. of Int'l. Conf. on Parallel and Distributed Information Systems*, FL, 1996.

[15] A. Labrinidis and N. Roussopoulos. Reduction of materialized view staleness using online updates. TR3878, DCS, Univ. of Maryland at College Park, Feb., 1998.

[16] D. Quass and J. Widom. On-line warehouse view maintenance. *Proc. of ACM-SIGMOD Conf.*, Tucson, Arizona, 1997, 393–404.

[17] A. Segev and J. Park. Updating distributed materialized views. *IEEE Trans. Knowledge and Data Engineering*, 1(2), 1989, 173–184.

[18] J. Wiener et al. A system prototype for warehouse view maintenance. *Proc. of Workshop on Materialized Views*, Montreal, Canada, 1996, 26–33.

[19] Y. Zhuge and H. Garcia-Molina. View maintenance in a warehousing environment Performance analysis of WHIPS incremental maintenance. *http://www-db.stanford.edu/ zhuge/publications.html*, 1998.

[20] Y. Zhuge et al. View maintenance in a warehousing environment. *Proc. of ACM-SIGMOD Conf.*, 1995, 316–327.

[21] Y. Zhuge et al. The strobe algorithms for multi-source warehouse consistency. *Proc. of Int'l. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, 1996.

[22] Y. Zhuge et al. Multiple view consistency for data warehousing. *IEEE ICDE'97*, Birmingham, UK, 1997.