International Journal of Parallel and Distributed Systems and Networks, Vol. 5, No. 4, 2002

PARALLEL MAINTENANCE OF MATERIALIZED VIEWS ON PERSONAL COMPUTER CLUSTERS

W. Liang* and J.X. Yu**

Abstract

A data warehouse is a repository of integrated information, which collects and maintains a large amount of data from multiple distributed, autonomous, and possibly heterogeneous data sources. Often the data are stored in the form of materialized views in order to provide fast access to the integrated data. How to maintain the warehouse data completely consistently with the remote source data is a challenging issue in a distributed environment. Transactions containing multiple updates at one or multiple sources further complicate this consistency issue. Due to the fact that a data warehouse usually contains a very large amount of data and its processing is time consuming, it becomes inevitable to introduce parallelism to data warehousing. The popularity and cost-effective parallelism brought by the PC cluster makes it a promising platform for this purpose. This article considers the complete consistency maintenance of select-project-join (SPJ) materialized views. Based on a PC cluster consisting of K personal computers, several parallel maintenance algorithms for the materialized views are presented. The key behind the proposed algorithms is how to trade off the work load among the PCs and how to balance the communications cost among the PCs as well as between the PC cluster and remote sources.

Key Words

Incremental maintenance, materialized view maintenance, data warehousing, partitioning, parallel algorithms, PC cluster

1. Introduction

A data warehouse mainly consists of materialized views, which can be used as an integrated and uniform basis for decision-making support, data mining, data analysis, and ad hoc querying across the source data. The maintenance problem of materialized views has received increasing attention in the past few years due to its application to data warehousing. Maintenance aims to maintain the content of a materialized view at a certain level of consistency with the remote source data, in addition to refreshing the content of the view as fast as possible when an update commits at one of the sources. It is well known that the data stored in data warehouses are usually very large amounts of historical, consolidated data. To respond to user queries quickly, it is inevitable that parallelism be introduced to the data warehousing in order to speed up data processing; hence the analysis for such a large volume of data is paintaking and time-consuming work. Thus, a parallel database engine is essential for large-scale data warehouses. With the popularity and cost-effectiveness brought by the personal computer (PC) cluster, it becomes one of the most promising platforms for data-intensive applications, such as large-scale data warehousing.

1.1 Related Work

Many incremental maintenance algorithms for materialized views have been introduced for centralized database systems [1–5]. A number of similar studies have been conducted in distributed resource environments [6–13]. These previous works 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 [8], to a full replication at the other end where the whole databases at the sources are copied to the warehouse so that the view maintenance can be handled in the warehouse locally [8–14]. The two extreme solutions are inefficient in terms of communication and query response time in the former case, and storage space in the latter case. A more efficient solution is to materialize the relevant subsets of source data in the warehouse (usually the query answer). Thus, only the relevant source updates are propagated to the warehouse, and the warehouse refreshes the materialized data incrementally against the updates [15–17]. However, in a distributed source environment, this approach may necessitate the warehouse contacting the sources many rounds for additional information to ensure the correctness of the update result [1, 10, 11, 18, 19].

To deal with the incremental maintenance of a materialized view, when and how to update the view are

* Department of Computer Science, Australian National University, Canberra, ACT 0200, Australia; e-mail: .....
** Department Systems Engineering and Engineering Management, Chinese University of Hong Kong, Hong Kong, PRC; e-mail: .....
(paper no. 204-0540)

also critical issues. In most commercial warehousing systems, updates to the source data are queued and propagated periodically to the warehouse in large batch update transactions called maintenance transactions. In order to maintain the content of view complete consistency with its source data, the approach most commonly used is to maintain the warehouse at nighttime when the warehouse is not available to users, whereas the user query can be processed in the daytime when the maintenance transactions are not running. Unfortunately, there are two major problems associated with this approach [20]. One is that there is no longer any nighttime common to all corporate sites during which it is convenient to make the warehouse unavailable to users, as corporations have become globalized; another is that 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 as all maintenance transactions must be done by the next morning. The incremental view maintenance thus is a more and more acceptable method for view maintenance. The authors of [8], they present an analytical model of view refresh time for virtual, fully materialized, and partially materialized views, and give some preliminary results. However, their experiments are based on a local network model in which the sources are located in one of the workstations connected by the local network, which is totally different from the one presented in this work. Therefore, the performance behaviour of their algorithm does not reflect the real condition of the problem in a wide area network. In [21] the authors also consider how to speed-up the refresh time of a materialized view by allocating extra space to that view, that is, each tuple in the view has two versions, to each update is attached a timestamp, and all user queries must be attached with a timestamp.

1.2 Our Contributions

In this article the maintenance of select-project-join (SPJ) materialized views, particularly the complete consistency maintenance, is considered. Three parallel maintenance algorithms for materialized views on a PC cluster are presented. The simple algorithm delivers a solution for complete consistency maintenance of a materialized view without using any auxiliary view. To improve the maintenance time of materialized views, the other two algorithms using auxiliary views are proposed. One is the equal partition-based algorithm, and the other is the frequency partition-based algorithm. They improve the view maintenance time dramatically compared with the simple algorithm, at the expense of extra warehouse space to accommodate the auxiliary data. The key to devising these algorithms is to explore the shared data, to trade off the work load among the PCs, and to balance the communications overheads among the PCs and between the PC cluster and the remote sources in a parallel computational platform.

1.3 Paper Outline

The rest of the article is organized as follows. Section 2 introduces the computational model and the four levels of consistency definition of materialized views. Section 3 presents a simple, complete consistency maintenance algorithm without the use of any auxiliary views. Section 4 devises a complete consistency algorithm based on the equal partitioning of sources in order to improve the view maintenance time. Section 5 gives another complete consistency algorithm based on the update frequency partitioning of sources, after taking into account both the source update frequency and the aggregate space needed for auxiliary views. Section 6 concludes the article.

2. Preliminaries

To keep a materialized view in a data warehouse at a certain level of consistency with its remote source data, extensive studies have been conducted. To the best of our knowledge, all previously known algorithms are sequential algorithms. In this work we focus on devising parallel algorithms for materialized view maintenance in a PC cluster.

2.1 Computational Model

A personal computer (PC) cluster consists of K ($K \geq 2$) PCs, interconnected through a high-speed network locally. Each PC in the cluster has its own main memory and disk. No shared memory among the PCs in the cluster exists. The communications among the PCs are implemented through message-passing mode. This parallel computational model is also called shared-nothing MIMD model.

In this article the defined PC cluster will serve as the platform for a data warehouse, and a data warehouse consists mainly; of the materialized views, the materialized views therefore are stored on the disks of PCs. For convenience, here we consider only relational views. It is well known that there are several ways to store a materialized view in an MIMD machine. One popular way is that the materialized view is partitioned horizontally (vertically) into K disjoint fragments, and each of the K fragments is stored into one of the PCs. However, in this work we do not intend to fragment the view and distribute its fragments to all PCs; rather, we assume that a materialized view is stored entirely in the disk of a PC. The reason behind this is that the content of a materialized view is consolidated, integrated data, which will be used for answering users' queries for decision-making purposes, and these data are totally different from the data in operational databases. Without loss of generality, let V be a materialized view located in PC_j; PC_j is called the *home* of V, $0 \leq j < K$. Note that a PC usually contains multiple materialized views. Following [10, 18], the update logs of the sources (relations) in the definition of V are sent to the data warehouse and stored at an *update message queue* (UMQ) for V, denoted by $UMQ(V)$.

2.2 View Consistency Concept

Assume that there are m materialized views in the warehouse and n remote data sources. A *warehouse state* ws represents the content of the data warehouse at that

moment, which is a vector of m components, and each component is the content of a materialized view at that moment. The warehouse state changes whenever one of the materialized views in it is updated. A source state ss represents the content of sources at a given time moment. A source state ss_j is a vector of n components, where each component represents the state of a source at that given time point. The ith component, $ss_j[i]$ of a source state represents the content of source i at that moment.

Let $ws_0, ws_1, \ldots, ws_f$ be the sequence of the warehouse states after a series of source update states $ss_0, ss_1, \ldots, ss_q$. Consider a view V derived from n sources. Let $V(ws_j)$ be the content of V at warehouse state ws_j, $V(ss_i)$ be the content of V over the source state ss_i, and ss_q be the final source state, $0 \leq i \leq q$, $0 \leq j \leq f$, and $f \leq q$. Furthermore, assume that source updates are executed in a serializable fashion across sources, and V is initially synchronized with the source data, that is, $V(ws_0) = V(ss_0)$. The following four levels of consistency between the materialized view V and its remote sources were defined in [10].

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

2. Weak consistency: Convergence holds and, for every warehouse state 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 $R = T_1, T_2, \ldots, T_k$ of transactions such that there is a locally serializable schedule at source x that achieves that state, $0 \leq k \leq q$.

3. Strong consistency: Convergence holds and there exists a serial schedule R and a mapping ϑ from the warehouse states to the source states with the following properties:
   (a) Serial schedule R is equivalent to the actual execution of transactions at the source.
   (b) For every ws_i, $\vartheta(ws_i) = ss_j$ for some j and $V(ws_i) = V(ss_j)$.
   (c) If $ws_i \prec ws_k$, then $\vartheta(ws_i) \prec \vartheta(ws_k)$ where $\prec$ is a precedence relation.
   That is, each warehouse state reflects a set of valid source states.

4. Completeness: The view in the warehouse is strong consistency with the source data and, for every ss_j defined by the serial schedule R, there exists a ws_i such that $\vartheta(ws_i) = ss_j$. That is, there is a complete order preserving mapping between the warehouse states and the source states.

As mentioned in [11], the complete consistency is a nice property because it always guarantees that the content of a view is completely consistent with its source data in every state. This restriction, however, may be too strong to be practical. In some cases, the strong consistency suffices.

2.3 Maintenance of Materialized Views

Let V be a SPJ-type view derived from n relations $R_1, R_2, \ldots, R_n$, and R_i be located at a remote source i, which is defined as follows:

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

where X is the set of projection attributes, P is the selection condition that is the conjunction of clauses like $R_i.A\theta R_j.B$ or $R_i.A\theta c$, A and B are the attributes of R_i and R_j, respectively, $\theta = \{<, >, =, \leq, \geq, \neq\}$, and c is constant, $1 \leq i \leq n$. Updates to source data are assumed to be either tuples' inserts or deletes. A modify operation is treated as a delete followed by an insert. Furthermore, all the views in the warehouse are based on the bag semantics, which means there is a *count field* for each tuple in a table, and the value of the count may be positive and zero.

To keep V at a certain level of consistency with its remote source data, several sequential algorithms have been proposed [11, 18, 19]. In this work we develop parallel maintenance algorithms in a distributed data warehouse environment where the data warehouse platform is a PC cluster of K PCs where we focus on the complete consistency maintenance of SPJ materialized views. For the sake of completeness, here we briefly restate the `SWEEP` algorithm [18], which will be used later. The `SWEEP` algorithm is chosen because it is the best algorithm for complete consistency maintenance so far. It is also the optimal one [22].

The `SWEEP` algorithm consists of two steps for the maintenance of a SPJ materialized view V. In the first step, it evaluates the update change δV to V due to a current source update δR. Although any further source updates may occur during the current update evaluation, to remove the effects of these later updates to the current update result, $UMQ(V)$ has been used to *offset* those effects. In the second step, the update result δV is merged with the content of V and V is updated. From this we can see that the first step is a dominant step that queries remote sources and performs the evaluation. The data manipulated in this step are the content of $UMQ(V)$ and the remote source data, they are totally independent of the content of V. The second step is a minimum cost step that merges δV to V in the data warehouse locally.

3. A Simple Parallel Algorithm

In this section we introduce a simple maintenance algorithm for materialized views distributed on a PC cluster. Before we proceed, we introduce a naive algorithm as follows.

Let V be a materialized view with home at PC_j. PC_j will take care of the maintenance of V; it will keep the update message queue $UMQ(V)$ for V. The sequential maintenance algorithm `SWEEP` will be run on PC_j for the maintenance of V. The performance of the naive algorithm reaches the optimal system performance if the materialized views in the data warehouse assigned to each PC have equal aggregate update frequencies. Otherwise, if

there are materialized views at some PCs that have much higher update frequencies than the others, those PCs that host the materialized view with higher update frequencies will become very busy and the other PCs may be idle during the maintenance period. Thus, the entire system performance will deteriorate due to the work-load being heavily imbalanced among the PCs. Above the all, this algorithm is not completely consistent, which is illustrated by the following example. Consider two materialized views MV_1 and MV_2 that are located at two different PCs. Assume there is a source update U_1 to MV_1 at time t_1 and another source update U_2 to MV_2 at time t_2 with $t_1 < t_2$. To respond to the updates, the two home PCs of these two views perform the maintenance to MV_1 and MV_2 concurrently. Assume that the update to MV_2 finishes before MV_1 does. Following the complete consistency definition, MV_1 should be updated before MV_2. Thus, this maintenance algorithm does not keep the materialized views in the data warehouse completely consistent with their remote source data.

3.1 A Simple Maintenance Algorithm

To overcome the work-load imbalance and to keep all materialized views completely consistent with their remote source data, a timestamp is assigned to the source update when the PC cluster receives a source update at the data warehouse side, and the source update is sent to the UMQs of those materialized views in which the source has been used in their definitions. The materialized views in the data warehouse are then updated sequentially by the order of timestamps assigned to them. For several materialized views sharing an update from a common source, the update sequence of the materialized views is determined by their topological order. Assume that the dependence relationships among the materialized views form a DAG. Now we are ready to give the detailed algorithm.

Given a materialized view V with PC_j as its home, by assumption there is an update message queue $UMQ(V)$ associated with V at PC_j. Let δR_i (δR_i may be either a set of insert updates ΔR_i or a set of delete updates ∇R_i) be a source update log in $UMQ(V)$. Denote by $UMQ(V)_i$ a partial queue of $UMQ(V)$ with the head δR_i, that is, $UMQ(V)_i$ is a queue such that all front of updates before δR_i have been removed from $UMQ(V)$ and δR_i becomes the head of the resulting queue. The proposed parallel algorithm proceeds as follows.

For each source update, δR_i, of the first K updates in the queue $UMQ(V)$, it is assigned to one of the K PCs in parallel (if the total number of updates in $UMQ(V)$ is less than K, then each update is assigned to one of the K PCs randomly; in the end some PCs are idle), so is $UMQ(V)_i$. $UMQ(V)_i$ will be used to offset the effect of later updates to the current update result derived from δR_i. Each PC then evaluates the view update to respond to the source update assigned to it. During the view update evaluation, once a source update related to V is received by the data warehouse, the source update will be sent to $UMQ(V)$ and $UMQ(V)_i$ for all i, $0 \leq i \leq K-1$.

Let δR_j be a source update in $UMQ(V)$ assigned to PC_i. PC_i is responsible for evaluating the view update $\delta V^{(j)}$ to V, using the sequential algorithm `SWEEP`. After the evaluation is finished, PC_i sends the result $\delta V^{(j)}$ to the home of V. When the home PC of V receives an update result, it first checks whether the update result is derived from the source update at the head of $UMQ(V)$. If it is, it merges the result with the content of V and removes the source update from the head of $UMQ(V)$. Otherwise, it waits until all the update results derived from the source updates in front of the current update in $UMQ(V)$ have been received and merged, and then merges the current result with the content of V. In the end, $V = V \cup \bigcup_{i=1}^{K} \delta V^{(i)}$.

Lemma 1. The simple maintenance algorithm is completely consistent.

Proof. Consider an update δR_i in $UMQ(V)$, which can be further distinguished by the following two cases: (1) δR_i is the head of $UQM(V)$; (2) δR_i is one of the first K updates in $UMQ(V)$.

Let us consider case (1). Assume that the source update δR_i is assigned to PC_j; then $UMQ(V)_i$, which is $UMQ(V)$ in this case, is also assigned to PC_j by the initial assumption. PC_j will evaluate the view update δV to V due to the update δR_i, using the `SWEEP` algorithm. Note that to evaluate δV, the data needed are only related to the source data, $UMQ(V)_i$, and the partial result of δV so far. Initially, the partial result of δV is empty. In other words, the evaluation of δV is independent of the content of V. Once the evaluation is done, the result is sent back to the home of the materialized view V. In this case the result will be merged to the content of V immediately due to δR_i being the head of $UMQ(V)$. Thus, the content $V \cup \delta V$ of V after the merge is completely consistent with the source data, because its behaviour is exactly as the same as the `SWEEP` algorithm.

Now we deal with case (2). Assume that δR_i is assigned to PC_j, so is the partial update message queue $UMQ(V)_i$. Following the argument in case (1), PC_j now is responsible to the evaluation of δV due to the update δR_i, and this evaluation can be done using the source data, $UMQ(V)_i$ and the partial update result of δV so far. Once the evaluation is done, the result is sent back to the home PC of V. If δR_i now becomes the head of $UMQ(V)$, it can be merged with the current content of V, and the merged result is completely consistent with the source data, which follows the `SWEEP` algorithm. Otherwise, if the view update results due to the source updates in front of δR_i in $UMQ(V)$ have not been merged with the content of V, then V is still in some old state; to maintain complete consistency of V, δV derived by δR_i cannot be merged to V until it becomes the head of $UMQ(V)$. Therefore, the lemma follows. □

The advantage of the proposed algorithm is that it keeps the work load of all PCs evenly, because at a given time interval each PC deals with a source update of a given materialized view V. However, a partial copy $UMQ(V)_i$ of

$UMQ(V)$ is needed to be distributed to all PCs; therefore, extra space is needed to accommodate these queues.

Compared with its sequential counterpart, the speedup obtained by this simple parallel algorithm is almost K in an ideal case where every PC is busy evaluating a source update, and the communications cost among the PCs is ineligible because only the incremental update results are sent back to the home PC of the materialized view V, but the data transfer from remote sites and the query evaluation at remote sites take a much longer time.

4. Equal Partition-Based Maintenance Using Auxiliary Views

Given a materialized view V, assume that the time used for the update evaluation δV is t, in response to a single source update δR_i. For each update, there is no difference in terms of its update evaluation time between running on the PC cluster and a single CPU machine, that is, the sequential and parallel algorithms will visit the other $n-1$ sources except the update source one by one in order to get the final update result. The time spent for the view maintenance is thus linear to the number of accesses to the remote sources. In the following, an approach aiming to reduce the number of such accesses is proposed.

4.1 Equal Partition-Based Algorithm

This approach is introduced to improve the view maintenance time using auxiliary views. The basic idea of the approach is first to derive several auxiliary views from the definition of the view, and each auxiliary view is derived from a subset of sources. The auxiliary views are materialized at the warehouse too. Then the view is redefined equivalently, using the auxiliary views instead of the base relations. Thus, the view update evaluation is implemented through evaluating its auxiliary views, which takes less time. The detailed explanation of the approach is as follows [19].

Let V be a materialized view derived from n relations. The n source relations is partitioned into $K = \lceil n/p \rceil$ disjoint groups, and each group consists of p relations except the last group, which contains $n - p \times \lfloor n/p \rfloor$ relations. Without loss of generality, assume that the first p relations form group one, the second p relations form group two, and the last $n - p\lfloor n/p \rfloor$ relations form group K. Following the definition of $V = \pi_X \sigma_P (R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)$, an auxiliary view AV_k is defined for each group k as follows, $0 \le k \le K-2$:

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

where $X(k)$ is an attribute set in which an attribute is either in X or in a clause of P such that the attribute comes from the relations in $\{R_{pk+1}, \ldots, R_{p(k+1)}\}$ and $P(k)$ is a maximal subset of clauses of P in which the attributes of each clause come only from R_{pk+1} to $R_{p(k+1)}$. Note that the attributes in $X(k) \cup P(k)$ come only from relations in $\{R_{pk+1}, \ldots, R_{p(k+1)}\}$. The last group $K-1$, V_{K-1} is defined as follows:

$$AV_{K-1} = \pi_{X(K-1)} \sigma_{P(K-1)} (R_{p(K-1)+1} \bowtie R_{p(K-1)+2} \bowtie \ldots \bowtie R_n) \quad (3)$$

where $X(K-1)$ and $P(K-1)$ in AV_{K-1} can be defined similarly as those defined in AV_{K-2}. Thus, V can then be rewritten equivalently in terms of the auxiliary views defined:

$$V = \pi_X \sigma_P (AV_0 \bowtie AV_1 \bowtie \ldots \bowtie AV_{K-2} \bowtie AV_{K-1}) \quad (4)$$

4.2 Parallel Algorithm

In this subsetion we show how to implement the `equal partition-based algorithm` in a cluster of K PCs by proposing a parallel maintenance algorithm.

Given an SPJ-type view V, assume that its K auxiliary views AV_i defined have been derived, $0 \le i \le K-1$. The maintenance of V is implemented through the maintenance of its auxiliary views. Let PC_j be the home of V. Initially, the K auxiliary views are assigned to the K PCs in the cluster. Assume that auxiliary view AV_i is assigned to $PC_{(i+r) \ mod \ K}$. Then AV_i is materialized at that PC too, where r is a given random number before the assignment. Let $k = (i + r) \ mod \ K$. Following the initial assumption, there is an update message queue $UMQ(AV_i)$ for AV_i at PC_k in addition to $UMQ(V)$ for V at PC_j. During the update evaluation, once a new source update is added to $UMQ(V)$, the home PC of V sends the update to PC_k immediately if the update comes from a source that has been used in the definition of AV_i.

Consider a source update δR_i that is the head of $UMQ(V)$. Assume that R_i has been used in the definition of AV_j, which is located in PC_k. Then to respond to the update δR_i, the view update evaluation δAV_j to AV_j will be carried out at PC_k by applying the sequential algorithm `SWEEP`. Once the evaluation is finished, the result δAV_j is not merged to the content of AV_j at PC_k immediately, in order to keep V completely consistent with the remote source data. But the result can be passed to $PC_{(k+1) \ mod \ K}$, which then performs the join with another auxiliary view AV_j' of V in it, and passes the joined result to its next neighbouring $PC_{(k+2) \ mod \ K}$, and so on. This procedure continues until the initial sender PC_k receives the joined result, which is the final result δV. PC_k sends the result to the home PC of V and merges the result with the content of V. At the same time, PC_k merges the partial result δAV_j with the content of AV_j. By (5), the correctness of the algorithm follows:

$$\delta V = \pi_X \sigma_P (AV_0 \bowtie AV_2 \bowtie \ldots \bowtie \delta AV_j \bowtie \ldots \bowtie AV_{K-1}) \quad (5)$$

Lemma 2. The equal partition-based maintenance algorithm is completely consistent.

Proof. Consider a source update δR_i. Assume that δR_i is used in the definition of AV_j that assigned to PC_k. The view update evaluation δV proceeds as follows.

The view update δAV_j is first evaluated by PC_k. To maintain the view completely consistent with the source data, the result δAV_j is not merged with the content of AV_j immediately because the view update evaluations from the other source updates after δR_i in $UQM(V)$ may use the content of AV_j for their evaluations. Note that $\delta AV_j \cup AV_j$ is completely consistent with the source data, which is guaranteed by the `SWEEP` algorithm.

We now proceed to the view update evaluation δV due to δR_i. Having obtained δAV_j, suppose that PC_k also holds a token for δAV_j. Following (5), to evaluate δV, PC_k sends its result δAV_k, which is a partial result of δV with the token to $PC_{(k+1) \ mod \ K}$ containing $AV_{j'}$. When an auxiliary view receives the token and the partial result, it performs a merge operation to produce a new partial update result $\delta AV_j \bowtie AV_{j'}$ of δV. Once the merge is done, it passes the partial result and the token to $PC_{(k+2) \ mod \ K}$, and so on. Finally, PC_k receives the partial update result, which is δV, and the token it is initially sent from. PC_k sends the result back to the home PC of V. The home PC of V now proceeds with the merge with the content of V, and removes the update δR_i from the head of $UMQ(V)$. At the same time, it informs PC_k to merge δAV_j with the content of AV_j. Obviously, the current content of V is completely consistent with the source data because all data in AV_j, $0 \leq j < K$ are at the state where the warehouse starts to deal with the view update evaluation due to δR_i and δR_i is the head of V. □

Compared with the simple maintenance algorithm, the equal partition-based parallel algorithm has dramatically reduced the size of the partial update message queue of $UMQ(V)$ at the other PCs except the home of the view. In this case the home PC of an auxiliary view AV_i holds only the update message queue $UMQ(AV_i)$ of AV_i, and $UMQ(AV_i)$ contains only the source update logs to the relations used in the definition of AV_i, rather than the relations used in the definition of V. Meanwhile, to obtain the view update evaluation result, the number of accesses to the remote sites is reduced to $\lceil n/K \rceil$ instead of n; therefore, it reduces the view maintenance time, improving the system performance. It must be mentioned this is obtained at the expense of more space for accommodating auxiliary views and extra time used for maintaining auxiliary views.

5. Frequency Partition-Based Maintenance Using Auxiliary Views

The performance of the equal partition-based algorithm deteriorates when the aggregate update frequencies of some auxiliary views are extremely high. As a result, the home PCs of these auxiliary views will have heavier work loads and the other PCs will have lighter workloads during the view maintenance period, because the home PC of a materialized (auxiliary) view is also responsible for handling the update result merging with its content in addition to handling the update evaluation for the auxiliary view on it, like any other PC. In this section we assume that not every source has identical update frequency. To balance the workload among the PCs in the cluster, each of the K auxiliary views of V must have equal update frequencies aggregately; finding such K auxiliary views derived from the definition of V has generally been proven to be NP-hard. Instead, two approximate solutions have been given, which are based on the minimum spanning tree and edge-contraction approaches [23]. Here we will use one of the algorithms for the K auxiliary views.

5.1 Frequency Partition-Based Algorithm

Let f_i be the update frequency of source R_i, $1 \leq i \leq n$ and $\sum_{i=1}^{n} f_i = 1$. The rest focuses on finding a K-partition of the sources to minimize the space for the auxiliary views and to balance the sum of source update frequencies among different groups. Formally speaking, given an SPJ view V and an integer K, the problem is to find K auxiliary views such that (1) the total space for the k auxiliary views is minimized, and (2) the absolute difference $|\sum_{v \in C_i} f(v) - \sum_{u \in C_j} f(u)|$ is minimized for any two groups of relations C_i and C_j with $i \neq j$, that is, the sum of the source update frequencies in each group is roughly equal, $\cup_{i=0}^{K-1} C_i = \{R_1, R_2, \ldots, R_n\}$, $C_i \cap C_j = \emptyset$, $i \neq j$ and $0 \leq i < j < K$. Clearly, the problem is an optimization problem with two objectives to be met simultaneously. The first objective is to minimize the extra warehouse space to accommodate the auxiliary views. The second objective is to balance the sources' update load. This optimization problem is NP-hard; instead, a feasible solution for it is given below.

An undirected weighted graph $G = (N, E, w_1, w_2)$ is constructed, where each relation used in the definition of V is a vertex in N. Associated with each vertex $v \in N$, the weight $w_1(v)$ is the update frequency of the corresponding relation. There is an edge between $u \in N$ and $v \in N$ if and only if there is a conditional clause in P containing the attributes from the two relational tables u and v only, and a weight $w_2(u, v)$ associated with the edge is the size of the resulting table after joining the two tables, where P is the selection condition in the definition of V. Having $G(N, E, w_1, w_2)$, an MST-based approximation algorithm for the problem is presented as follows [23].

`Appro_Partition`(G, N, E, w_1, w_2, K)

/* w_1 is the weight function of vertices and w_2 is the weight function of edges */

1. Find a minimum spanning tree $T(N, E', w_1)$ from G.
2. Apply the approach in [24] to find a max-min K partition of T.
3. The vertices in each subtree form a group, and a vertex partition $\mathcal{P}$ of G is obtained.

The K-vertex partition $\mathcal{P}$ in G is obtained by running algorithm `Appro_Partition`. K auxiliary views can then be derived by the definition of V, and each is derived from a group of relations C_i, $0 \leq i < K$. Note that each auxiliary view obtained has equal update frequency aggregately.

5.2 Parallel Algorithm

For a given SPJ-type view V, assume that the K auxiliary views above defined have been found by applying the `Appro_Partition` algorithm. We then assign each of the auxiliary views to one of the K PCs in the cluster. The remaining processing is exactly the same as that in the equal partition-based maintenance algorithm, omitted. Therefore, we have the following lemma.

Lemma 3. The frequency partition-based maintenance algorithm is completely consistent.

Proof. The proof is similar to Lemma **?**. □

In contrast to the `equal partition-based algorithm`, the above algorithm can improve the performance of the entire system even when the update frequency at each site is imbalanced heavily.

6. Conclusions

In this article several parallel algorithms for materialized view maintenance have been proposed, based on a PC cluster. The proposed algorithms guarantee the content of a materialized view completely consistent with its remote source data. The key to devising these algorithms is to explore the shared data and trade off the work load among the PCs and to balance the communication overhead between the PC cluster and the remote sources and among the PCs in a parallel computational environment.

Acknowledgements

The work described in this article was partially supported by a grant (F00025) from the Australian Research Council, and the Research Grants of Council of the Hong Kong Special Administrative Region (Project No. CUHK4198/00E).

References

[1] J.A. Blakeley, P. A. Larson, & F.W. Tompa, Efficiently updating materialized views, *Proc. 1986 ACM-SIGMOD Conf.*, 1986, 61–71.

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

[3] A. Gupta & I. Mumick, Maintenance of materialized views: problems, techniques, and applications, *IEEE Data Engineering Bulletin, 18*(2), 1995, 3–18.

[4] A. Gupta, I. Mumick, & V.S. Subrahmanian, Maintaining views incrementally, *Proc. 1993 ACM-SIGMOD Conf.*, 1993, 157–166.

[5] T. Griffin & L. Libkin, Incremental maintenance of views with duplicates, *Proc. 1995 ACM-SIGMOD Conf.*, 1995, 328–339.

[6] L. Colby, T. Griffin, L. Libkin, I. Mumick, & H. Trickey, Algorithms for deferred view maintenance, *Proc. 1996 ACM-SIGMOD Conf.*, 1996, 469–480.

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

[8] R. Hull & 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, Canada, 1996, 91–102.

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

[10] Y. Zhuge, H. Garcia-Molina, J. Hammer, & J. Widom, View maintenance in a warehousing environment, *Proc. 1995 ACM-SIGMOD Conf.*, 1995, 316–327.

[11] Y. Zhuge, H. Garcia-Molina, & J.L. Wiener, The strobe algorithms for multi-source warehouse consistency, *Proc. Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, 1996, 146–157.

[12] Y. Zhuge, H. Garcia-Molina, & J.L. Wiener, Multiple view consistency for data warehousing, *IEEE ICDE'97*, Birmingham, UK, 1997, 289–300.

[13] Y. Zhuge, H. Garcia-Molina, & J.L. Wiener, Consistency algorithms for multi-source warehouse view maintenance, *Journal of Distributed and Parallel Database. 6*, 1998, 7–40.

[14] A. Gupta, H. Jagadish, & I. Mumick, Data integration using self-maintainable views, *Proc. 4th Int. Conf. on Extending Database Technology*, 1996, 140–146.

[15] D. Quass, A. Gupta, I.S. Mumick, & J. Widom, Making views self-maintainable for data warehousing, *Proc. Int. Conf. on Parallel and Distributed Information Systems*, Miami Beach, FL, 1996, 158–169.

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

[17] W. Liang, H. Li, H. Wang, & M. Orlowska, Making multiple views self-maintainable in a data warehouse, *Data and Knowledge Engineering, 30*(2), 1999, 121–134.

[18] D. Agrawal, A. El Abbadi, A. Singh, & T. Yurek, Efficient view maintenance at data warehouses, *Proc. 1997 ACM-SIGMOD Conf.*, 1997, 417–427.

[19] H. Wang, M. Orlowska, & W. Liang, Efficient refreshment of materialized views with multiple sources, *Proc. 8th ACM Conf. on Information and Knowledge Managements*, 1999, 375–382.

[20] D. Theodoratos, S. Ligoudistianos, & T. Sellis, Designing the global data warehouse with SPJ views, *Proc. 11th Int. Conf. on Advanced Information Systems Engineering*, 1999, 180–194.

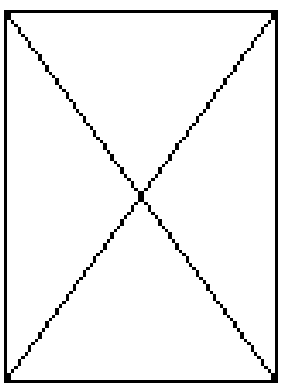
[21] D. Quass & J. Widom, On-line warehouse view maintenance, *Proc. 1997 ACM-SIGMOD Conf.*, Tucson, Arizona, 1997, 393–404.

[22] W. Liang & J.X. Yu, Revisit on view maintenance in data warehouses, *Proc of 2nd Int. Conf. on Web-Age Information Management*, Lecture Notes on Computer Science, Vol. 2118, 2001, 203–211.

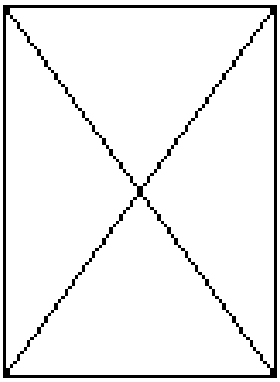
[23] W. Liang, C. Johnson, & J.X. Yu, Maintaining materialized views for data warehouses with the multiple remote source environments, *Proc of 1st Int. Conf. on Web-Age Information Management*, Lecture Notes on Computer Science, Vol. 1846, 2000, 299–310.

[24] Y. Perl & S.R. Schach, Max-min tree partitioning, *Journal of the ACM, 28*(1), 1981, 5–15.

Biographies



Weifa Liang received his .....



Jeffrey X. Yu received his .....