Parallel Maintenance of Materialized Views on Personal Computer Clusters

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

Jeffrey X. Yu
Dept. Systems Eng. and Eng. Management
Chinese University of Hong Kong
Shatin, N.T., Hong Kong
email: yu@se.cuhk.edu.hk

ABSTRACT
A data warehouse is a repository of integrated information that collects and maintains a large amount of data from multiple distributed, autonomous and possibly heterogeneous data sources. Often the data is stored in the form of materialized views in order to provide fast access to the integrated data. How to maintain the warehouse data completely consistent with the remote source data is a challenging issue, and 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 become a promising platform for such purpose.

In this paper the complete consistency maintenance of select-project-join (SPJ) materialized views is considered. 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 tradeoff the work load among the PCs and how to balance the communications cost among the PCs as well between the PC cluster and remote sources.

KEY WORDS
Materialized view incremental 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 been received increasing attention in the past few years due to its application to data warehousing. The view 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 is usually very large amount of historical, consolidated data. To respond to user queries quickly, it is inevitable to introduce parallelism to speed up the data processing in data warehousing, due to that the analysis of such large volume of data is painstaking and time consuming. Thus, parallel database engines 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 for large scale data warehousing.

Many incremental maintenance algorithms for materialized views have been introduced for centralized database systems [2, 6, 7, 4]. A number of similar studies have also been conducted in distributed resource environments [3, 8, 15]. 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 [5, 8]. 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. 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 [9, 10]. 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 [15, 3, 1, 14].

To keep a materialized view in a data warehouse at a certain level of consistency with its remote source data, extensively studies have been conducted in the past. To the best of our knowledge, all those previously known algorithms are sequential algorithms. In this paper we focus on devising parallel algorithms for materialized view maintenance in a PC cluster. Specifically, the complete consistency maintenance of select-project-join (SPJ) materialized views 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 another 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 of devising these algorithms is to explore the shared data, to tradeoff 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.

The rest of the paper is organized as follows. Section 2 introduces the computational model and 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 presents 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 paper.

2 Preliminaries

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 paper the defined PC cluster will serve as the platform for a data warehouse, while a data warehouse consists of the materialized views mainly, the materialized views therefore are stored on the disks of PCs. For convenience, we here only consider 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 paper we do not intend to fragment the view and distribute its fragments to all PCs, rather, we assume that a materialized view is stored in the disk of a PC entirely. The reason behind this is that the content of a materialized view is consolidated, integrated data, which will be used for answering users' query for decision making purpose, and this data is 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 [15, 1], 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 $UMQ(V)$.

View consistency. 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 warehouse state sequence 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 the sources, and V is initially synchronized with the source data, i.e., $V(ws_0) = V(ss_0)$. The following four levels of consistency between the materialized view V and its remote sources has been defined in [15].

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 are 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 achieving that state, $0 \leq k \leq q$.

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

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 is a ws_i such that $\vartheta(ws_i) = ss_j$. That is, there is a complete order preserving mapping between the warehouse and source states.

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 is located at a remote source i, which is defined as $V = \pi_X \sigma_P(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)$, where X is the set of projection attributes, P is the selection condition which 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, $\theta \in \{<, >, =, \leq, \geq, \neq\}$, 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. All 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 [15, 1, 14]. In this paper we dedicate ourselves to develop parallel maintenance algorithms in a distributed data warehouse environment where the data warehouse platform is a PC cluster of K PCs on which we focus on the complete consistency maintenance of SPJ materialized views. For the sake of completeness, here we briefly restate the SWEEP algorithm [1] 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 [12].

The SWEEP algorithm consists of two steps for the maintenance of a SPJ materialized view V. In step one, it evaluates the update change δV to V due to a current source update δR. While any further source updates may occur during the current update evaluation, to remove the effects of these later updates on the current update result, $UMQ(V)$ has been used to *offset* those effects. In step two, the update result δV is merged with the content of V and V is updated. It is easy to see that step one is the dominant step which queries remote sources and performs the evaluation. While the data manipulated in this step are the content of $UMQ(V)$ and the remote source data, it is totally independent of the content of V. Step two is a minimum cost step which 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. First of all we introduce the following naive algorithm.

Let V be a materialized view with home at PC_j. PC_j will take care of the maintenance of V and 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 this 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 which have much higher update frequencies than the others, then, the PCs hosting these materialized views will become very busy while the other PCs may be idle during the whole maintenance period. Thus, the entire system performance will be deteriorated due to the work load heavily imbalance among the PCs. Above the all, this algorithm is not completely consistent, illustrated by the following example. Consider two materialized views MV_1 and MV_2 which are located at two different PCs. If 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 the 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.

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, 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. If several materialized views sharing an update from a common source, then the update sequence of these materialized views is determined by their topological order in a DAG, assuming that the dependence relationships among the materialized views forms a DAG. Now we are ready to give the detailed algorithm.

Given a materialized view V with PC_j as its home, by the 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, i.e., $UMQ(V)_i$ is such a queue 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 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 to evaluate 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 yes, 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 re-

ceived and merged, and then merges the current result with the content of V. As results we have $V = V \cup \bigcup_{i=1}^{K} \delta V^{(i)}$.

Lemma 1 *The simple maintenance algorithm is completely consistent.*

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

Let us consider case (i). 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 is 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 that δR_i is 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 it's behavior is exactly as the same as the SWEEP algorithm.

We now deal with case (ii). Assume that δR_i is assigned to PC_j, so is the partial update message queue $UMQ(V)_i$. Following the argument in case (i), PC_j now is responsible to the evaluation of δV due to the update δR_i, while 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 keeps the work load of all PCs evenly because at a given time interval, each PC deals with a source update of a given a materialized view V. However, a partial copy $UMQ(V)_i$ of $UMQ(V)$ is needed to be distributed to all the PCs, therefore, the extra space is needed to accommodate these queues. Compared with its sequential counterpart, the speed-up obtained by this simple parallel algorithm is almost K in an ideal case where every PC is busy for 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, while the data transfer from remote sites and the query evaluation at remote sites take much longer time.

4 Equal Partition-Based Maintenance

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, i.e., 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 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 behind it is first to derive several auxiliary views from the definition of a view, and each auxiliary view is derived from a subset of sources. The auxiliary views are materialized at the warehouse too. The view then is re-defined 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 [14].

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 containing $n - p \times \lfloor n/p \rfloor$ 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 for each group k is defined as follows, $0 \leq k \leq K-2$,

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

where $X(k)$ is an attribute set in which an attribute is either in X or in such a clause of P 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 $X(k) \cup P(k)$ only come from relations in $\{R_{pk+1}, \ldots, R_{p(k+1)}\}$ only. The last group $K-1$, V_{K-1} can be defined similarly. Thus, V can then be rewritten equivalently in terms of the auxiliary views defined, $V = \pi_X \sigma_P (AV_0 \bowtie AV_1 \bowtie \ldots \bowtie AV_{K-2} \bowtie AV_{K-1})$.

4.2 Parallel algorithm

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

Given a SPJ-type view V, assume that its K auxiliary views AV_i have been derived, $0 \leq i \leq K-1$. 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 which has been used in the definition of AV_i.

Consider a source update δR_i which is the head element in $UMQ(V)$. 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 then it passes the joined result to its next neighboring $PC_{(k+2) \ mod \ K}$, and so on. This procedure continues until the initial sender PC_k receives the joined result which is the final result δV actually, 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 Eq. (2), the correctness of the proposed algorithm follows.

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

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 which is 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 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 Eq. (2), 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 actually, and the token from which it is initially sent. PC_k sends the result back to the home PC of V. The home PC of V now proceeds the merge with the content of V, and removes 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, $0 \leq j < K$ is at the state where the warehouse starts to deal with the view update evaluation due to δR_i and δR_i is the head in $UMQ(V)$. □

Compared with the simple maintenance algorithm, the equal partition-based parallel algorithm has reduced the size of the partial update message queue of $UMQ(V)$ at other PCs except the home of the view dramatically. In this case the home PC of an auxiliary view AV_i only holds the update message queue $UMQ(AV_i)$ of AV_i, while $UMQ(AV_i)$ contains only the source update logs of 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, thereby improving the system performance ultimately. It must be mentioned this is obtained at the expenses of more space for accommodating auxiliary views and extra time used for maintaining auxiliary views.

5 Frequency Partition-Based Maintenance

The performance of the equal partition-based algorithm is deteriorated when the aggregate update frequencies of some auxiliary views are extremely high. As a result, work loads of the home PCs of these auxiliary views will be heavier while the work loads of other PCs will be lighter during the view maintenance period, because the home PC of a materialized (auxiliary) view is also responsible to handle the update result merging with its content in addition to handling the update evaluation for the auxiliary view on it, like any other PCs. In this section we assume that not every source has identical update frequency. To balance the work load among the PCs in the cluster, it requires that each of the K auxiliary views of V have equal update frequencies aggregately, while finding such K auxiliary views derived from the definition of V has generally been shown to be NP-hard. Instead, two approximate solutions have been given, which are based on the minimum spanning tree and edge-contraction approaches [11]. Here we will use one of the algorithms for finding 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 $\sum_{i=1}^{n} f_i = 1$. Given a SPJ view V and an integer K, the problem is to find K auxiliary views such that (i)

the total space of the k auxiliary views is minimized; and (ii) 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 $i \neq j$, i.e., 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 [11].

`Appro_Partition`(G, N, E, w_1, w_2, K)
/* w_1 and w_2 are the weight functions of vertices and edges */
1. Find a minimum spanning tree $T(N, E', w_1)$ from G;
2. Find a max-min K partition of T by an algorithm in [13].
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 an 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 2, omitted. □

6 Conclusions

In this paper 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 devise these algorithms is to explore the shared data and tradeoff 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.

Acknowledgment: The work was partially supported by both a small grant (F00025) from Australian Research Council and the Research Grants of Council of the Hong Kong Special Administrative Region (Project No. CUHK4198/00E).

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

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

[5] A. Gupta et al. Data integration using self-maintainable views. *Proc. 4th Int'l Conf. on Extending Database Technology*, 1996, 140–146.

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

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

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

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

[10] W. Liang et al. Making multiple views self-maintainable in a data warehouse. *Data and Knowledge Engineering*, 30(2), 1999, 121–134.

[11] W. Liang et al. Maintaining materialized views for data warehouses with the multiple remote source environments. *Proc of 1st Int'l Conf. on WAIM*, LNCS, Vol. 1846, 299–310, 2000.

[12] W. Liang and J. X. Yu. Revisit on view maintenance in data warehouses. *Proc of 2nd Int'l Conf. on WAIM*, LNCS, Vol. 2118, 203–211, 2001.

[13] Y. Perl and S. R. Schach. Max-min tree partitioning. *J. ACM*, 28(1), 1981, 5–15.

[14] H. Wang et al. Efficient refreshment of materialized views with multiple sources. *Proc. of 8th ACM-CIKM*, 1999, 375–382.

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