Maintaining Materialized Views for Data Warehouses With Multiple Remote Sources

Weifa Liang† Chris Johnson† Jeffrey X. Yu‡

† Department of Computer Science,The Australian National University
Canberra, ACT 0200, Australia
‡ Department Systems Engineering and Engineering Management
Chinese University of Hong Kong, Hong Kong

Abstract. A data warehouse is a data repository which 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. However, to maintain the data in the warehouse consistent with the source data is a challenging task in a multiple remote source environment. Transactions containing multiple updates at one or more sources further complicate the consistency issue. In this paper we first consider improving the refresh time of select-project-join (SPJ) type materialized views in a data warehouse by presenting a frequency-partitioned based algorithm, which takes into account the source update frequencies and the total space for auxiliary data. We then propose a solution in the design of data warehouses which can handle a variety of materialized views with different refreshment requirements.

1 Introduction

Data warehouses usually store materialized views in order to provide fast access to the data integrated from several distributed data sources [5], while the data sources are heterogeneous and remote from the warehouse. In the study of data warehousing, one important problem is the view maintenance problem which aims to maintain the content of a materialized view at a certain level of consistency with the source data when updates commit at sources. Consequently, the view maintenance problem with multiple remote sources is totally different from its counterpart in a centralized data warehouse.

Related work: Many incremental maintenance algorithms for materialized views have been introduced for centralized database systems [2, 3, 5, 8, 6]. There are also a number of researchers investigating similar issues in distributed environments [4, 9, 10, 20–22]. A good overview of the work done in this area can be found in [5]. 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 [10], to a full replication at the other end where the whole databases at the sources are copied to the warehouse so that the view refreshments (updates) can be handled in the

warehouse locally [7, 15, 10]. The two extreme solutions are inefficient in terms of communication and query response time in the former case, and storage space in the latter. An efficient solution is to materialize some relevant subsets of source data in the warehouse (usually the query answers), the warehouse is able to refresh the materialized data incrementally against the updates when any relevant updates at sources are propagated to the warehouse. However, in a distributed source environment, this approach may necessitate a solution in which the warehouse contacts the sources for additional information to ensure the update correctness, in order to maintain the data in the warehouse at a certain level of consistency with the source data [21, 20, 4, 1, 9].

In a multiple remote source environment, how to improve the refresh time of a materialized view is very important. In [16] they considered this problem by allocating extra space to the view, i.e., they keep multiple versions of a materialized view in the warehouse. Thus, each tuple in the view has multiple versions with different timestamps. Each subsequent update or query is also with a timestamp. Although this approach allows the view refreshment (view maintenance) and the user query to be processed simultaneously, it still requires that the warehouse contacts the remote sources when performing view refreshment. A similar approach based on timestamps is also given in [12], but this latter algorithm essentially proceeds the periodic updates, not on-line incremental updates that we will consider here. In contrast to keeping multiple versions of a materialized view, there is another approach called *self-maintainable approach* [7, 15, 13], which keeps all relevant auxiliary information of a materialized view in the warehouse. Thus, the view in the data warehouse can be updated using the source update log and the auxiliary information only when there is any updates at sources, without consulting with the remote sources. This approach requires that the data warehouse run in the two modes (user query and view maintenance) exclusively. As a result, the refresh time of materialized views is reduced significantly, at the expense of lots of space spent for storing the auxiliary information. Similar to the second approach, the third approach also requires that the data warehouse run in user query and view maintenance modes. But no any auxiliary information is stored in the warehouse, the view refreshment is implemented through contacting the remote sources [1, 20] during the warehouse maintenance mode. The advantage of this approach is that it does not need any extra space for auxiliary information. On the other hand, the refresh time may take very long, depending on how many sources have to be contacted when performing the view refreshment. Recently a new approach is suggested which just stores partial auxiliary information in the warehouse. At the same time, the warehouse is allowed to have a limited number of accesses to the remote sources [18]. This approach trades-off the two important factors, the amount of available extra space in the warehouse and the refresh time of the views very well. However, they only considered the case where every source has the same update frequency and there is no space limit for auxiliary information.

Our contributions: Following the same spirit in [18], in this paper a complete consistency algorithm for SPJ type of views is proposed. This frequency-

partitioned based algorithm groups the sources into several groups such that the sum of source update frequencies of each group is roughly equal and the space for auxiliary views derived from the groups of sources is reasonably small. The algorithm improves the view refresh time at the expense of extra warehouse space to accommodate auxiliary views. Also, a possible solution in the design of data warehouse is suggested, which can handle a variety of materialized views with different refreshment requirements.

2 Preliminaries

2.1 The data warehouse model

A *data warehouse* is a repository of integrated data, which collects and maintains a large amount of data from multiple distributed, autonomous and possibly heterogeneous sources. A typical data warehouse architecture defined in [19] is illustrated in Figure 1. Associated with every source there is a monitor/wrapper

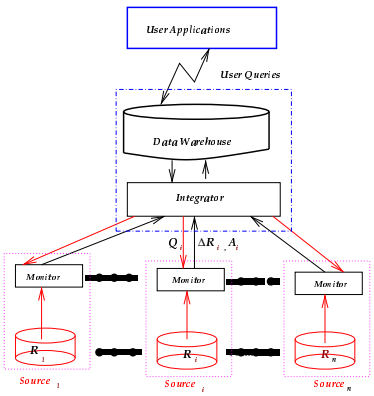


Fig. 1. A data warehouse architecture

which collects the data of interest and sends the data to the warehouse. The monitor/wrapper is 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 to compute the difference between successive snapshots of the source data. At the warehouse side, there is an integrator which 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. In effect, the warehouse caches a materialized view of the source data. The data is then readily available to user applications for querying and analysis. The communication between a source and the warehouse is assumed to be reliable FIFOs, i.e., messages are not lost on their way and are delivered

in the order that they are sent. No communication is imposed between any two sources. In fact, they may be completely independent, and may not be able to communicate with each other.

It is easy to see that the complexity of designing algorithms for view maintenance 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. (ii) Source local transactions where a sequence of updates are performed as a single transaction. However, all of the updates are directed to a single data source. (iii) Global transactions where the updates involve multiple sources. In this paper we only consider the refreshment problem for SPJ-type materialized views. We further assume that the updates being handled at the warehouse are of types (i) and (ii). The approach described in [21] can be used to extend the proposed algorithms for the updates in type (iii). The source updates can be handled at the warehouse in different ways, depending on how the updates are incorporated into the view at the warehouse. Different levels of consistency of a view have been identified as follows, by Zhuge et al [20] and Hull et al [9]. 1. **Convergence** where the updates are incorporated into the materialized view eventually. 2. **Strong consistency** where the order of the state transformations of the view at the warehouse corresponds to the order of the state transformations at the data sources. 3. **Completeness** where every state of the data sources is reflected as a distinct state at the data warehouse, and each state of views at the warehouse corresponds to a state of data sources. That is, there is a complete order preserving mapping between the states of views and the states of the sources.

As claimed in [21], complete consistency is a nice property since it guarantees that the content of a view is always consistent with its source data in every state. However, this restriction may be too strong in practice. In some cases, the convergence may be sufficient even if there are some invalid intermediate states. In most cases, the strong consistency which corresponds to the batch updates is desirable. In this paper we will focus on the design of maintenance algorithms for complete consistency, which can also be extended to strong consistency easily.

2.2 The refresh time of materialized views

Let V be a SPJ view derived from n relations $R_1, R_2, \ldots, R_n$, defined as follows.

$$V = \pi_X \sigma_P (R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n) \tag{1}$$

where X is the set of projection attributes, P is the selection condition consisting of the conjunction of disjunctive clauses, and R_i is a database located at a remote source i, $1 \leq i \leq n$. An update to the source data is assumed to be either insert or delete of tuples. A modify is treated as a delete followed by an insert. Also, all the views in the warehouse are based on set semantics.

Given a materialized view V in the warehouse and its two consecutive 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. The refresh time, T_{fresh}, of V from ws_i to ws_{i+1}, is $T_{fresh} = t_1 - t_0$.

To update V incrementally from ws_i to ws_{i+1} due to an update in some source, a number of rounds of 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 answers $A_{i_1}, A_{i_2}, \ldots, A_{i_N}$ of the N queries. Then, the refresh time of V from ws_i to ws_{i+1} is $T_{fresh} = \sum_{l=1}^{N}(T_{transfer\_to\_source}(Q_{i_l}) + T_{transfer\_to\_warehouse}(A_{i_l}) + T_{warehouse\_join}(A_{i_l}) + T_{source\_join}(Q_{i_l}))$, where $T_{transfer\_to\_source}(Q_{i_l})$ is the transmission time to send query Q_{i_l} from the warehouse to source i_l; $T_{transfer\_to\_warehouse}(A_{i_l})$ is the transmission time from source i_l to the warehouse by returning the query answer; $T_{warehouse\_join}(A_{i_l})$ is the time for processing the join of A_{i_l} and the concurrent updates received so far, which is evaluated at the warehouse; $T_{source\_join}(Q_{i_l})$ is the time for processing query Q_{i_l} at source i_l, which is evaluated at source i_l. To simplify our later discussion, assume that the transmission time between the warehouse and the sources is symmetric and fixed, denoted by $\overline{t}$. We further assume that the query processing time at either the warehouse or the sources is negligible because the processing time for this part only occurs a small proportion of the entire refresh time, compare to the information transmission time in a wide area network. Also, the number of tuples involved in incremental maintenance of materialized view is relatively small, compared with the size of the materialized view. Therefore, the corresponding query processing time is relatively small. After the simplifications, T_{fresh} can be rewritten as $T_{fresh} = 2N\overline{t}$, which depends on the number of queries sent N by the warehouse.

2.3 An equal-partitioned based algorithm

In [18] an equal-partitioned based algorithm is proposed, which not only reduces a view refresh time but also maintains the complete consistency between the content of the view and the source data, providing that some extra warehouse space is available. Let V be a SPJ materialized view derived from n relations located at n sources. Assume that p is a given integer.

The equal-partitioned algorithm proceeds as follows. First, the n source relations are 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. Based on the partition of sources, an auxiliary view V_k for each group k is then derived, $1 \leq k \leq K$, as follows. $V_k = \pi_{X(k)}\sigma_{P(k)}(R_{p(k-1)+1} \bowtie R_{p(k-1)+2} \bowtie \ldots \bowtie R_{pk})$ for all k with $1 \leq k \leq K-1$, where $X(k)$ is an attribute set in which an attribute is either in X or appeared in such a disjunctive clause of P that some its attributes come from the relations which are not in $\{R_{p(k-1)+1}, \ldots, R_{pk}\}$, all the attributes in $X(k) \cup P(k)$ come from relations in $\{R_{p(k-1)+1}, \ldots, R_{pk}\}$ only, 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, $V_K = \pi_{X(K)}\sigma_{P(K)}(R_{p(K-1)+1} \bowtie R_{p(K-1)+2} \bowtie \ldots \bowtie R_n)$, where $X(K)$ and $P(K)$ in V_K can be defined similarly as we did for V_k. Thus, V is rewritten in an equivalent form semantically, using the auxiliary views defined, $V = \pi_X\sigma_P(V_1 \bowtie V_2 \bowtie \ldots \bowtie V_{K-1} \bowtie V_K)$.

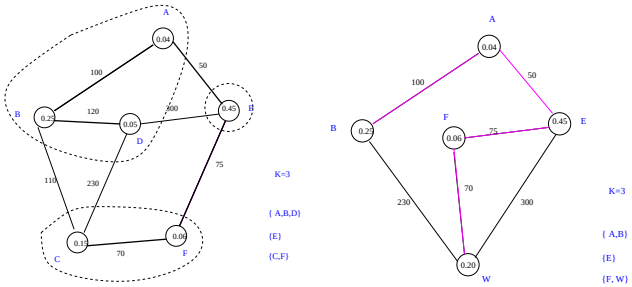
To refresh V, two assumptions are imposed: (1) V and an auxiliary view V_k are materialized in the warehouse for any k, $1 \leq k \leq K$; (2) the warehouse is capable to decide to which auxiliary view an update belongs and to which auxiliary view a query answer (from sources) belongs. We consider the incremental maintenance of V due to an update ΔR_i at source i, assuming that R_i is in group k. The proposed refreshment algorithm proceeds as follows. First, we perform the incremental maintenance for V_k, using the `SWEEP` algorithm [1]. As a result, an updated view $V_k = V_k \cup \Delta V_k$ is obtained, and so is ΔV_k. We then update V locally without consulting the remote sources. Let V^{new} be the updated result of V, $V^{new} = V \cup \Delta V$ where $\Delta V = \pi_X \sigma_P (V_1 \bowtie V_2 \bowtie \ldots \bowtie V_{i-1} \bowtie \Delta V_i \bowtie \ldots \bowtie V_K)$. Thus, ΔV can be obtained by local evaluation because each V_j is materialized in the warehouse, $1 \leq j \leq K$ with $j \neq k$. To maintain V in complete consistency with the source data, all the other auxiliary views are locked during the incremental maintenance of V_k. Otherwise, the data in the other auxiliary views used later for the maintenance of V may be contaminated during the update. The proposed algorithm has been shown to have complete consistency with the source data. The refresh time of V is $T_{view\_update\_W} = 2(p-1)\overline{t} + t_{WH}$, where t_{WH} is the update time used to update V in the warehouse by joining ΔV_k and all the other V_j with $j \neq k$, which depends on the number of auxiliary views $\lceil n/p \rceil$ participating in the joining and the size of each table involved. From the formula for $T_{view\_update\_W}$, we can see that the choice of the parameter p is very important. If p is chosen too small, although 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 is required. Also, the overhead of updating V in the warehouse t_{WH} will increase because more relations need to be joined. Otherwise, there is no any substantial improvement in terms of the view refresh time, despite extra warehouse space spent for auxiliary views.

3 A Frequency Partitioned Based Algorithm

Let V be a SPJ materialized view derived from n sources and f_i be the update frequency of source R_i, $1 \leq i \leq n$ and $\sum_{i=1}^{n} f_i = 1$. To improve the refresh time of V, the relations in the definition of V first are partitioned into K groups (assume that K is given at the beginning) such that the sum of source update frequencies in each group is roughly equal and the total space used for the auxiliary views is limited, where an auxiliary view is derived for each of the groups using the relations in the group and the definition of V. Formally speaking, given a SPJ view V and an integer K, the problem is to find K auxiliary views such that (i) the total space for the 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 C_i and C_j with $i \neq j$, i.e., the sum of source update frequencies in each group is roughly equal, where C_i is the set of source relations. Clearly, this 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 loads. It is not hard to show that this optimization problem is NP-hard. Instead, we will focus on finding feasible solutions for it by proceeding as follows.

We first construct an undirected weighted graph $G = (N, E, w_1, w_2)$ where each relation in the definition of V is a vertex in N. For every $v \in N$, the associated weight $w_1(v)$ is the update frequency of the corresponding relation. There is an edge $(u, v) \in E$ between $u \in N$ and $v \in N$ if and only if there is a conditional clause in P which contains the attributes only from u and v. The weight $w_2(u, v)$ associated with (u, v) is the size of the resulting table after joining the two tables with the conditional clause, where P is the selection condition in the definition of V. In addition, assume that G is connected. Otherwise, our algorithm will apply to each connected component of G. For example, consider a SPJ view V that consists of six relations. Figure 2(a) is such a graph, where vertex A has weight 0.04 which is the update frequency of the corresponding source. The weight of edge (A, B) is 100, which is the size of the resulting table after joining tables A and B using the conditional clause between them. To



(a) A graph for frequency-based partition

(b) A resulting graph G'.

Fig. 2. An example

avoid the *direct product* joining in generating the auxiliary views, the vertices in a group must be connected in G. Thus, the above optimization problem then becomes as follows. Let $\mathcal{P} = \{C_1, C_2, \ldots, C_K\}$ be a vertex partition of G, i.e., $\cup_{i=1}^{K} C_i = N$ and $C_i \cap C_j = \emptyset$ if $i \neq j$. The problem is to partition the vertices in G into K groups such that (1) the vertices in each group are connected; (2) the weighted sum of vertices in each group is roughly equal, i.e., minimize $|\sum_{v \in C_i} w_1(v) - \sum_{u \in C_j} w_1(u)|$ for any two groups C_i and C_j with $i \neq j$; and (3) the weighted sum of the edges in the forest consisting of K subtrees is minimized. In other words, condition (3) is to minimize the space for the auxiliary views.

3.1 The K-subtree partition problem

Consider the K-subtree partition problem, defined as follows. Given a tree $T(N, E_1, w_1)$, where w_1 is a weighted function of vertices in T, each vertex $v \in N$ has a weight $w_1(v)$ and $|N| = n$, E_1 is the edge set of T, the problem is to partition T into K subtrees such that the weighted sum of vertices in each subtree is roughly equal. In other words, the problem is to find a K-partition of the tree which maximizes the weight of the lightest weighted subtrees.

There is a naive approach for the problem, described as follows. Each time we first remove $K-1$ tree edges from the tree. As a result, the tree becomes a forest of K subtrees. We then compute the absolute difference of the weighted sums of vertices between any two subtrees and find the maximum one. There are $\binom{n-1}{K-1}$ ways to remove $K-1$ edges from the tree which lead to $\binom{n-1}{K-1}$ different ways of K-subtree partitioning. We finally choose a partition which minimizes the maximum absolute difference in the partition. Obviously, this approach requires $O(n^K)$ time, which grows exponentially when K is not fixed. Another efficient approach to solving the K-subtree partition problem is given in [14], which takes $O(K^2 rd(T) + Kn)$ time, where $rd(T)$ is the number of edges in the radius of T. Note that the K-subtree partition problem was given a different name called *max-min K partition* [14].

3.2 The MST-based approximation algorithm

Given V and its source update frequencies, assume that $G(N, E, w_1, w_2)$ defined has been constructed. The MST-based approximation algorithm consists of two stages. In the first stage it optimizes the space for auxiliary views. In the second stage it balances the load of source updates among the auxiliary views. The algorithm for the optimization problem is presented as follows.

`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 an MST $T(N, E', w_1)$ from G, using the edge weights;
2. Find a max-min K partition of T, using the vertex weights.
3. The vertices in a subtree form a group and a partition $\mathcal{P}$ of N is obtained.

Having the K-vertex partition $\mathcal{P}$, an auxiliary view for each group is then derived using the definition of V. We claim that the space for the auxiliary views generated by the algorithm is reasonably small, which is demonstrated through an example. Consider a view consisting of the joining of three relations R_1, R_2 and R_3. Let s_{12} and s_{23} be the sizes of $R_1 \bowtie R_2$ and $R_2 \bowtie R_3$, respectively. To estimate the size s of $R_1 \bowtie R_2 \bowtie R_3$, a linear cost model is employed, which has also been used in [11] for the similar purpose. Thus, $s = s_{12} + s_{23}$ under this space cost model, the weighted sum of the edges in a minimum spanning tree of a subgraph of G induced by the vertices in a group is equal to the amount of space used for the auxiliary views approximately. Since this space cost model is just an estimate of the real cost. The space for auxiliary views, therefore, is not necessarily optimal, for a given K.

Consider the example given in Figure 2(a) with $K = 3$. A minimum spanning tree in G is first constructed, by applying `Appro_Partition`, which consists of edges $\{(A,B),(A,E),\ (B,D),(C,F),(E,F)\}$. A 3-vertex partition $\mathcal{P} = \{\{A,B,D\},\{E\},\{C,F\}\}$ of the tree is then obtained. In this example, the maximum value of the absolute difference between two groups is 0.24. The space for auxiliary views is $(w_2(A,B)+w_2(B,D))+S_E+w_2(C,F) = 100+70+130+70 = 370$, assuming that the size S_E of the tuples in table E satisfying the condition of V is 130.

3.3 The edge contraction approximation algorithm

Here we give another heuristic algorithm for the optimization problem which delivers a solution that better balances the source update load among the auxiliary views.

Let $c > 1$ be constant and $M = \sum_{v \in N} w_1(v)$. The proposed approximation algorithm also consists of two stages. Assume that the current graph is $G(N, E, w_1, w_2)$ initially. In the first stage it proceeds the following iterations until either K vertices are left in the resulting graph or there does exist two neighboring vertices satisfying the *edge-contraction* condition. Each time the algorithm chooses two neighboring vertices u and v such that u and v are the two minimum weighted vertices in the current graph and the sum of $w_1(u)+w_1(v) \leq M/c$, which is referred to the edge-contraction condition. If u and v satisfy the edge contraction condition, they are merged by contracting the edge between them. After the merge, a new vertex w is created. The weight assigned to w is the sum $w_1(u) + w_1(v)$ of the update frequencies of u and v. If either (u,x) or (v,x) but not both of them exists in the current graph, an edge between w and x is then created and assigned weight $w_2(w,x) = w_2(u,x)$, assuming that (u,x) exists. Otherwise, an edge (w,x) is created and is assigned weight $w_2(u,x) + w_2(v,x)$. A resulting graph G' is obtained after the iteration terminated. In the second stage, if G' contains K vertices exactly, then the K vertices form a K-vertex partition of G. The vertices in G merged to a vertex in G' form a group. Otherwise, G' contains more than K vertices, the MST-based approximation algorithm is applied. That is, an MST T' is found for G' first, followed by applying the algorithm [14] for finding a max-min K partition of T'. Clearly, the vertices in each group are connected in G.

Consider the given example in Figure 2(a) with 3-vertex partition ($K = 3$) by applying the edge contraction approximation algorithm. Let $c = 5$. $M = \sum_{v \in N} w_1(v) = 1$. First, two neighboring vertices C and D are chosen to be merged because $w_1(C) + w_1(D) = 0.2 \leq M/c$. The resulting graph G' is generated in Figure 2(b). After that, there is no any neighboring vertices in G' satisfying the edge contraction condition. The algorithm then shifts to the second stage in which an MST T' of G' is obtained, and T' consists of the edges $\{(A,B),(A,E),(E,F),(F,W)\}$. A 3-vertex partition of T' $\mathcal{P}' = \{\{A,B\},\{E\},\{F,W\}\}$ therefore is obtained. As a result, a corresponding 3-vertex partition of G finally is achieved, which is $\mathcal{P} = \{\{A,B\},\{E\},\{C,D,F\}\}$. In this example, the maximum value of the absolute difference between any two groups is

0.20. The space for auxiliary views is $w_2(A,B) + S_E + (w_2(C,D)) + w_2(C,F)) = 100 + 130 + (230 + 70) = 530$. Compared with `Appro_Partition`, this latter algorithm gives a better solution in terms of load of source updates for each auxiliary view but a worse solution in terms of the space for auxiliary views.

3.4 Optimizing space for auxiliary views

The algorithms above are based on an assumption that the number of auxiliary views K is already given at the very beginning. For a given materialized view, sometimes a smaller or a larger K may lead to small space for auxiliary views. However, when K is too large (almost the linear size of the size of the sources), it is possible to make a duplicate in the warehouse for each source, thus all materialized views in the warehouse are self maintainable. When K is too small, there is no significant improvement in the refresh time despite spending space for auxiliary views. Without loss of generality, we here assume that the number of auxiliary views is between $\lceil \sqrt[3]{n} \rceil$ and $\lfloor \sqrt{n} \rfloor$. Then, we can find a K such that the total space for auxiliary views is minimized by running the following algorithm.

`Opt_Appro_Partition`$(G, N, E, w_1, w_2, K, S)$
/* w_1 is the weight function of vertices and w_2 is the weight function of edges,*/
$K' := \lfloor \sqrt{n} \rfloor$; $Total\_space := S$ /* S is the total space limit. */
repeat
1. call `Appro_Partition`$(G, N, E, w_1, w_2, K', S)$;
2. Let $S_{K'}$ be the space required for auxiliary views with the K'-partition;
3. if $S_{K'} < Total\_space$
4. then $Total\_space := S_{K'}$; $K := K'$;
5. $K' = K' - 1$;

until $K' \leq \lceil \sqrt[3]{n} \rceil$.

4 Improving Refresh Time of Multiple Views

In this section we deal with how to improve the refresh time for a group of SPJ materialized views instead of a single view. One trivial solution for this problem is to build a set of auxiliary views for each individual materialized view by applying the approaches in the previous section. However, the solution obviously is not the best one because it does not take into account the shared information among the materialized and auxiliary views. In the following we propose an approach which exploits the shared information among the materialized views as well the auxiliary views. Assume that there is a set MV of materialized views with $|MV| = m$. The query frequency of a materialized view $mv_i \in MV$ is q_i and $\sum_{i=1}^{m} q_i = 1$, $1 \leq i \leq m$. Further assume that the materialized views are derived from n remote data sources. Let f_j be the update frequency of source R_j and $\sum_{j=1}^{n} f_j = 1$, $1 \leq j \leq n$. The algorithm is proposed as follows.

1. Sort the materialized views by their query frequencies in decreasing order. Let $mv_1, mv_2, \ldots, mv_m$ be the sequence of the materialized views after the

sorting, i.e., $q_1 \geq q_2 \geq \ldots \geq q_m$. Let $AV := \emptyset$ /* the auxiliary view set */

2. for $i := 1$ to m do
   - 2.1 Apply algorithm `Appro_Partition` to find a K-partition of sources for mv_i. Let S_{mv_i} be the space used by the auxiliary views AV_{mv_i} for mv_i.
   - 2.2 Use the views in $U = \{mv_1, mv_2, \ldots mv_{i-1}\} \cup AV$ to rewrite mv_i. Let $mv_i = ma \cup mb$ where ma is the rewritten part by the existing views in U and mb is the original part. Find a set of auxiliary views AVB_{mv_i} for mb, let S_b be the space requirement for this part.
   - 2.3. If $S_{mv_i} < S_b$ then rewrite mv_i using the auxiliary views in AV_{mv_i}; $AV = AV \cup \{AV_{mv_i}\}$. else rewrite mv_i using the views in $AV = AV \cup \{AVB_{mv_i}\}$.

Thus, an auxiliary view set AV for the views in MV has been obtained, which can be used to improve the refreshment of the materialized views in MV.

5 View Maintenance with Different Refreshments

In this section we consider the design of a data warehouse which can handle a variety of materialized views with different refreshment requirements by giving a possible solution. As we know, the refresh time of a materialized view mainly depends on how many remote sources need to be communicated with and how long the communication delay takes. Therefore, to improve the refreshment of a materialized view is to reduce its contact with the remote sources. To this end, we classify the materialized views in a warehouse into three categories, depending on their refreshment requirements. (i) Type one materialized views: the views are the most accessed views and need to be refreshed as quickly as possible. (ii) Type two materialized views: the views are refreshed periodically and must be refreshed within the given refresh time window. (iii) Type three materialized views: the views are only refreshed if no other "urgent" views to be refreshed. The views are refreshed by the user request. The following policy is proposed to handle the three type views.

Let MA_{urgent} be the set of type one materialized views. Apply algorithms in [15, 13, 17] to find a set of auxiliary views $\mathcal{A}$ such that all views in $\mathcal{A} \cup MA_{urgent}$ are self-maintainable. Thus, once there are any changes in the sources, the views in MA_{urgent} will be refreshed using the source update log without consulting the remote sources. Let MA_{need} be the set of type two materialized views. For each of type two materialized views, rewrite it using the views in $\mathcal{A} \cup MA_{urgent}$ if possible. If it can be rewritten completely, rewrite it using the views. Otherwise, apply the algorithm in Section 4 to find a set of auxiliary views AV for them and rewrite it using the auxiliary views. The refreshment of this kind of view may have a limited number of contacts to the remote sources. For each of type three materialized views, if it can be rewritten using the views in $\mathcal{A} \cup MA_{urgent}$, then rewrite it. Otherwise, rewrite it using the views in $\mathcal{A} \cup MA_{urgent} \cup MA_{need} \cup AV$ if it can be rewritten using the views in $MA_{urgent} \cup \mathcal{A} \cup MA_{need} \cup AV$. In most cases, to implement its refreshment, the view needs to communicate with the remote sources, using the refreshment algorithm like `SWEEP` [1].

Acknowledgement. The work by Weifa Liang and Jeffrey X. Yu was partially supported by a small grant (F00025) from Australian Research Council.

References

1. D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient view maintenance at data warehouses. *Proc. of ACM-SIGMOD Conf.*, 1997, 417–427.
2. J.A. Blakeley, P. A. Larson, and F. W. Tompa. Efficiently updating materialized views. *Proc. of ACM-SIGMOD Conf.*, 1986, 61–71.
3. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. *Proc. of the 17th VLDB Conf.*, 1991, 577–589.
4. L. Colby, T. Griffin, L. Libkin, I. Mumick, and H. Trickey. Algorithms for deferred view maintenance. *Proc. of the 1996 ACM-SIGMOD Conf.*, 1996, 469–480.
5. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing.* 18(2), June, 1995.
6. T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. *Proc. of ACM-SIGMOD Conf.*, 1995, 328–339.
7. A. Gupta, H. Jagadish, and I. Mumick. Data integration using self-maintainable views. *Proc. 4th Int'l Conf. on Extending Database Technology*, 1996, 140–146.
8. A. Gupta, I. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *Proc. of ACM-SIGMOD Conf.*, 1993, 157–166.
9. R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. *Proc. of ACM-SIGMOD Conf.*, 1996.
10. 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, Canada, 1996, 91–102.
11. W. J. Labio, R. Yerneni, H. Garcia-Molina. Shrinking the warehouse update window. *Proc. of ACM-SIGMOD Conf.*, 1999, 383–394.
12. A. Labrinidis and N. Roussopoulos. Reduction of materialized view staleness using online updates. TR3878, DCS, Univ. of Maryland at College Park, 1998.
13. W. Liang, H. Li, H. Wang and M. Orlowska. Making multiple views self-maintainable in a data warehouse. *DKE*, 30(2), 1999, 121–134.
14. Y. Perl and S. R. Schach. Max-min tree partitioning. *J. ACM*, 28(1), 1981, 5–15.
15. D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. *Proc. of PDIS'96*, FL, 1996, 158–169.
16. D. Quass and J. Widom. On-line warehouse view maintenance. *Proc. of ACM-SIGMOD Conf.*, Tucson, Arizona, 1997, 393–404.
17. D. Theodoratos, S. Ligoudistianos, and T. Sellis. Designing the global data warehouse with SPJ views. *Proc. of the 11th Int'l Conf. on Advanced Information Systems Engineering*, 1999, 180–194.
18. H. Wang, M. Orlowska, and W. Liang. Efficient refreshment of materialized views with multiple sources. *Proc. of the 8th ACM-CIKM*, 1999, 375–382.
19. J. Wiener, H. Gupta, W. Labio, Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. A system prototype for warehouse view maintenance. *Proc. of Workshop on Materialized Views*, Montreal, Canada, 1996, 26–33.
20. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. *Proc. of ACM-SIGMOD Conf.*, 1995, 316–327.
21. Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The strobe algorithms for multi-source warehouse consistency. *Proc. of PDIS'96*, FL, 1996, 146–157.
22. Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. Multiple view consistency for data warehousing. *IEEE ICDE'97*, Birmingham, UK, 1997, 289–300.