Efficient View Maintenance Using Version Numbers

Eng Koon Sze and Tok Wang Ling

National University of Singapore
3 Science Drive 2, Singapore 117543
{szeek, lingtw}@comp.nus.edu.sg

Abstract. Maintaining a materialized view in an environment of multiple, distributed, autonomous data sources is a challenging issue. The results of incremental computation are effected by interfering updates and compensation is required. In this paper, we improve the incremental computation proposed in our previous work by making it more efficient through the use of data source and refreshed version numbers. This is achieved by cutting down unnecessary maintenance queries and thus their corresponding query results. The number of times of sending subqueries to a data source with multiple base relations are also reduced, as well as avoiding the execution of cartesian products. Updates that will not affect the view are detected and incremental computation is not applied on them. We also provide a compensation algorithm that resolve the anomalies caused by using the view in the incremental computation.

1 Introduction

Providing integrated access to information from different data sources has received recent interest from both the industries and research communities. Two methods to this data integration are the *on-demand* and the *in-advance* approaches.

In the on-demand approach, information is gathered and integrated from the various data sources only when requested by users. To provide fast access to the integrated information, the in-advance approach is preferred instead. Information is extracted and integrated from the data sources, and then stored in a central site as a materialized view. Users access this materialized view directly, and thus queries are answered immediately. Noting that information at the data sources do get updated as time progresses, this materialized view will have to be *refreshed* accordingly to be consistent with the data sources. This refreshing of the materialized view due to changes at the data sources is called *materialized view maintenance*. The view can be refreshed either by recomputing the integrated information from scratch, or through incrementally changing only the affected portion of the view.

It is inefficient to recompute the view from scratch. The deriving of the relevant portion of the view and then incrementally changing it is a preferred approach as a smaller set of data is involved.

In Section 2, we explain the working of materialized view maintenance, and its associated problems. In Section 3, we briefly discuss the maintenance algorithm proposed in [7]. The improvements to this algorithm is given in Section 4, and the changes to the compensation algorithm of [7] are given in Section 5. We compare related works in Section 6, and conclude our discussion in Section 7.

2 Background

In this section, we explain the view maintenance algorithm and its problems.

2.1 Incremental Computation

We consider the scenario of select-project-join view V, with n numbers of base relations $\{R_i\}_{1 \leq i \leq n}$. Each base relation is housed in one of the data sources, and there is a separate site for the view. The view definition is given as $V = \prod_{proj\_attr} \sigma_{sel\_cond} R_1 \bowtie ... \bowtie R_n$. A *count* attribute is appended to the view relation if it does not contain the key of the join relations $R_1 \bowtie ... \bowtie R_n$ to indicate the number of ways the same view tuple could be derived from the base relations. There are multiple, distributed, autonomous data sources, each with one or more base relations. There is communication between the view site and the data sources, but not between individual data sources. Thus a transaction can involve one or more base relations of the same data source, but not between different data sources. The view site does not control the transactions at the data sources. No assumption is made regarding the reliability of the network, i.e., messages sent could be lost or could arrive at the destination in a different order from what was originally sent out.

The data sources send notifications to the view site on the updates that have occurred. To incrementally refresh the view with respect to an update $\triangle R_i$ of base relation R_i, $R_1 \bowtie ... \bowtie \triangle R_i \bowtie ... \bowtie R_n$ is to be computed. The result is then applied to the view relation.

2.2 View Maintenance Anomalies and Compensation

There are three problems in using the incremental approach to maintain the view.

Interfering Updates If updates are separated from one another by a sufficient large amount of time, incremental computation will not be affected by interfering updates. However, this is often not the case.

Example 1. Consider $R_1(\underline{A}, B)$ and $R_2(\underline{B}, C)$, and view $V = \prod_C R_1 \bowtie R_2$, with a count attribute added for the proper working of the incremental computation. R_1 contains the single tuple $(a1, b1)$ and R_2 is empty. Hence the view is also empty. Insert $R_1(a2, b1)$ occurs. The view receives this notification and the query $R_1(a2, b1) \bowtie R_2$ is sent to R_2. Let insert $R_2(b1, c1)$ occurs just before the view maintanance query for insert $R_1(a2, b1)$ reaches R_2. Thus, the tuple

$(b1, c1)$ would be returned. The overall result (without projection) is the tuple $(a2, b1, c1)$. The single projected tuple $(c1)$ is added to the view to give $(c1, 1)$. When the view receives the notification of insert $R_2(b1, c1)$, it formulates another query $R_1 \ltimes R_2(b1, c1)$ and sends it to R_1. The result $\{(a1, b1), (a2, b1)\}$ is returned to the view site. The overall result is $\{(a1, b1, c1), (a2, b1, c1)\}$ and this adds 2 more tuples of $(c1)$ to the view. The presence of interfering update (insert $R_2(b1, c1)$) in the incremental computation of insert $R_1(a2, b1)$ adds an extra tuple of $(c1)$ to the view relation, giving it a count value of 3 instead of 2.

Misordering of Messages *Compensation* is the removal of the effect of interfering updates from the query results of incremental computation. Most of the existing compensation algorithms that remove the effect of interfering updates and thus achieve complete consistency [11] are based on the first-sent-first-received delivery assumption of the messages over the network, and thus will not work correctly when messages are misordered. A study carried out by [3] has shown that one percent of the messages delivered over the network are misordered.

Loss of Messages The third problem is the loss of messages. Although the loss of network packets can be detected and resolved at the network layer, the loss of messages due to the disconnection of the network link (machine reboot, network failure, etc.) has to be resolved by the application itself after re-establishing the link. As the incremental computation and compensation method of the maintenance algorithm is driven by these messages, their lost would cause the view to be refreshed incorrectly.

3 Version Numbers and Compensation for Interfering Updates

The following types of version numbers are proposed in [7]. **Base relation version number** of a base relation. The base relation version number identifies the state of a base relation. It is incremented by one when there is an update transaction on this base relation. **Highest processing version number** of a base relation, and this number is stored at the view site. The highest processing version number is used to provide information to the maintenance algorithm on the updates of a base relation which have been processed for incremental computation. It indicates the last update transaction of a base relation which has been processed for incremental computation. **Initial version numbers** of an update. The initial version numbers of an update identify the states of the base relations where the result of the incremental computation should be based on. Whenever we pick a update transaction of a base relation for processing, the current highest processing version numbers of all base relations become the initial version numbers of this update. At the same time, the highest processing version number of the base relation of this update is incremented by one, which is also the same as the base relation version number of the update. **Queried**

version number of a tuple of the result from a base relation. The queried version numbers of the result of incremental computation of an update indicate the states of the base relations where this result is actually generated. Base relation and highest processing version numbers are associated with the data source and view site respectively, while initial and queried version numbers are used only for the purpose of incremental computation and need not be stored permanently.

The different types of version numbers allow the view site to identify the interfering updates independent of the order of arrival of messages at the view site. The numbers in between the initial and queried version numbers of the same base relation are the version numbers of the interfering updates. This result is stated in Lemma 1, which was given in [7]. Once the interfering updates are identified, compensation can be carried out to resolve the anomalies. Compensation undoes the effect on the result of incremental computation caused by interfering updates. The formal definiton for compensating the interfering updates can be found in [7].

Lemma 1. *Given that a tuple of the query result from R_j has queried version number β_j, and the initial version number of R_j for the incremental computation of $\triangle R_i$ is α_j. If $\beta_j > \alpha_j$, then this tuple requires the compensation with updates from R_j of base relation version numbers β_j down to $\alpha_j + 1$. These are the* ***interfering updates*** *on the tuple. Otherwise if $\beta_j = \alpha_j$, then compensation on the query result from R_j is not required.*

4 Improved Incremental Computation

The view maintenance approach in [7] overcomes the problems caused by the misordering and loss of messages during transmission. However, efficiency in the incremental computation is not taken into consideration. Each sub-query only accesses one base relation, but generally a data source has multiple base relations. [7] uses the same strategy to incrementally compute the change for all updates. Since the view contains partial information of the base relations, we propose in this paper to involve the view in the incremental computation.

4.1 Querying Multiple Base Relations Together

Since a data source usually has more than one base relation, the sub-queries sent to it should access multiple base relations. This cuts down the total network traffic. It also reduces the time required for the incremental computation of an update, and results in smaller number of interfering updates. Using the join graph to determine the access path of querying the base relations, instead of doing a left and a right scans of the relations based on their arrangement in the relation algebra of the view definition, cuts down the size of the query results sent through the network by avoiding cartesian products.

Briefly, the incremental computation is handled as follows. Consider the join graph of the base relations of a view. The view maintenance query starts with

the base relation R_i, $1 \leq i \leq n$, where the update has occurred. A sub-query is sent to a set of relations S, where $S \subset \{R_j\}_{1 \leq j \leq n}$, the relations in S comes from the same data source, R_i and the relations in S form a connected sub-graph with R_i as the root of this sub-graph. If there are more than one such set of base relations S, multiple sub-queries can be sent out in parallel. For each sub-query sent, we marked the relations in S as "queried". R_i is also marked as "queried". Whenever a result is returned from a data source, another sub-query is generated using the similar approach. Let R_k, $1 \leq k \leq n$, be one of the relations that have been queried. A sub-query is sent to a set of relations S, where $S \subset \{R_j\}_{1 \leq j \leq n}$, the relations in S comes from the same data source, none of the relations in S are marked "queried", and R_k and the relations in S form a connected sub-graph with R_k as the root of this sub-graph. Again, if there are more than one such set of base relations S, multiple sub-queries can be sent in parallel. The incremantal computation for this update is completed when all the base relations are marked "queried".

4.2 Identifying Irrelevant Updates

If a data source enforces the referential integrity constraint that each tuple in R_j must refer to a tuple in R_i, then we know that an insertion update on R_i will not affect the view and thus can be ignored by our view maintenance process. Similarly, deletion update on R_i will not affect the view if the data source further enforces that no tuple in R_i can be dropped when there are still tuples in R_j referencing it.

4.3 Partial Self-Maintenance Using Functional Dependencies

We proposed the involvement of the view relation to improve the efficiency of the maintenance algorithm by cutting down the need to access the base relations.

Additional Version Numbers We propose the following additions to the concept of version numbers given in [7]. This enhancement would allow a data source to have more than one base relation, i.e., update transaction of a data source can involve any number of the base relations within it, and enable the maintenance algorithm to utilize the view in its incremental computation.

The two new types of version numbers are as follow. **Data source version number** of a data source. Since a data source usually has more than one base relation, it is not sufficient to determine the exact sequence of two update transactions involving non-overlapping base relations using the base relation version number alone. The data source version number is used to identify the state of a data source. It is incremented by one when there is an update transaction on the data source. **Refreshed version number** of a base relation, and this number is stored at the view site. If we want to use the view relation for incremental computation, we provide the refreshed version numbers to identify the states. The refreshed version number indicates the state of a base relation the view relation is currently showing.

The data source version number is used to order the update transactions from the same data source for incremental computation, and subsequent refreshing of the view. The base relation version number continues to be used for the identification of interfering updates. The refreshed version number is assigned to the queried version number of the tuples of the result when the view is used for incremental computation.

Accessing View Data for Incremental Computation Lemma 2 uses functional dependencies to involve the view in the incremental computation. In the case where the view is not used, all tuples in μR_i (the query result of the incremental computation from R_i) need to be used to query the next base relation R_j. When conditions (1) and (2) of Lemma 2 are satisfied, only those tuples in μR_i that cannot be matched with any of the tuples in $V[R'_j, S']$ need to be used in accessing the base relations.

Lemma 2. *When the incremental computation of an update (insertion, deletion, and modification) needs to query base relation R_j, (1) if the key of R_j is found in the view, and (2) if R_j functionally determines the rest of the relations S that are to be queried based on the query result of R_j (denoted as μR_j), then the view can be accessed for this incremental computation and the refreshed version numbers are taken as the queried version numbers for the result. The view is first used for the incremental computation using $\mu R_j, S = \mu R_i \ltimes V[R'_j, S']$, where R'_j is the set of attributes of R_j in V, S' is the set of attributes of relations S in V, and $\mu R_j, S$ is the query result for R_j and the set of relations S. For the remaining required tuples are not found in the view, the base relations are next accessed.*

Example 2. Consider $R_1(\underline{A}, B, C)$, $R_2(\underline{C}, D, E)$ and $R_3(\underline{E}, F, G)$, with the view defined as $V = \prod_{B,C,F} R_1 \bowtie R_2 \bowtie R_3$. The following shows the initial states of the base and view relations.

$R_1(\underline{A}, B, C)$	$R_2(\underline{C}, D, E)$	$R_3(\underline{E}, F, G)$	$V(B, C, F, count)$
a1,b1,c1	c1,d1,e1	e1,f1,g1	b1,c1,f1,1
	c2,d2,e2	e2,f2,g2	

Let R_1 reside in data source 1, and R_2 and R_3 reside in data source 2. The base relation version numbers of R_1, R_2 and R_3 are each 1, and the data source version numbers of data sources 1 and 2 are also 1. The refreshed version numbers at the view site are $\langle 1, 1, 1 \rangle$ for R_1, R_2 and R_3 respectively.

Update transaction with data source version number 2 occurs at data source 1, and the updates involved are insert $R_1(a2, b2, c2)$ and insert $R_1(a3, b3, c1)$, which now has its base relation version number changed to 2. The view site receives this notification and proceed to handle the incremental computation. The highest processing version number of R_1 is updated to 2. Thus, the initial version numbers for the incremental computation of this update are $\langle 2, 1, 1 \rangle$. R_2 and R_3 are to be queried. Since the key of R_2, which is C, is in the view, and R_2 functionally determines the other base relations to be queried (only R_3 in this case), the view is first accessed for this incremental computation using the

query $\{(a2, b2, c2), (a3, b3, c1)\} \rtimes V[R_2', R_3']$ (R_2' contains attribute C, and R_3' contains attribute F). It is found that the tuple (a3,b3,c1) can join with the tuple (c1,-,-) from R_2 and (-,f1,-) from R_3 ($C \rightarrow F$). Note the use of "-" for the unknown attributes values. The queried version numbers for both tuples are 1, taken from the refreshed version numbers for R_2 and R_3. Projecting the overall result over the view attributes adds one tuple of (b3,c1,f1) to the view. Thus the tuple (b3,c1,f1,1) is inserted into the view relation. The tuple (a2,b2,c2) that cannot retrieve any results from the view relation will have to do so by sending the view maintenance query $[\prod_C \{(a2, b2, c2)\}] \rtimes (R_2 \bowtie R_3)$ to data source 2. The result returned consists of the tuple (c2,d2,e2) from R_2 and (e2,f2,g2) from R_3, each with queried version number 1. Since the queried version numbers here correspond with the initial version numbers of both R_2 and R_3, there is no interfering update and compensation is not required. The tuple (b2,c2,f2,1) is inserted into the view.

Maintaining the View Without Querying All Base Relations It is not necessary to know the complete view tuples before the view can be refreshed in some cases of modification or deletion updates. In this paper, modification update that involves the change of any of the view's join attributes will be handled as a deletion and an insertion updates because these update will join with different tuples of the other relations after the modification. Otherwise, the modification update will be handled as one type of update by our view maintenance algorithm. Applying Lemma 3 would also serve to reduce the overall size of queries and results transmitted.

Lemma 3. *For a modification or deletion update $\triangle R_i$, if the key of R_i is in the view, then maintenance can be carried out by modifying or deleting the corresponding tuples of $\triangle R_i$ in the view through using the key value of $\triangle R_i$, without the need to compute the complete view tuple.*

5 Compensation for Missing Updates

Lemma 1, which was proposed in [7], is used to identify interfering updates in the case where the base relations is queried for incremental computation. Using the view relation for incremental computation also creates the similar kind of problem, in that the view relation might not be refreshed to the required state when it is accessed. We called them *missing* updates to differentiate from the interfering updates.

Lemma 4. *Extending Lemma 1, the following is added. If $\beta_j < \alpha_j$, then this tuple (taken from the view relation) requires the compensation with updates from R_j of base relation version numbers β_j+1 to α_j. These are the **missing updates** on the tuple.*

5.1 Resolving Missing Updates

The compensation of a missing insertion update is to add the tuples of this insertion into the query result. The compensation with a missing modification update is simply to update the tuples from the unmodified state to the modified state. These are given in Lemmas 5, 6 and 7 respectively.

Lemma 5. *Let μR_j be the query result from R_j (retrieved from the view) for the incremental computation of $\triangle R_i$. To compensate the effect of* ***missing deletion update*** *$\triangle R_j$ for the result of incremental computation of $\triangle R_i$, all tuples of $\triangle R_j$ that are found in μR_j are dropped, together with those tuples from the other base relations that are retrieved due to the original presence of $\triangle R_j$ in μR_j.*

Lemma 6. *Let μR_j be the query result from R_j (retrieved from the view) for the incremental computation of $\triangle R_i$, and μR_k be the query result from R_k. To compensate the effect of* ***missing insertion update*** *$\triangle R_j$ on the result of incremental computation of $\triangle R_i$, and assuming that μR_j is queried using the result from μR_k, all tuples of $\triangle R_j$ that can join with μR_k are added to μR_j, together with those tuples from the other base relations that should be retrieved due to the inclusion of $\triangle R_j$ in μR_j.*

Lemma 7. *Let μR_j be the query result from R_j (retrieved from the view) for the incremental computation of $\triangle R_i$. To compensate the effect of* ***missing modification update*** *$\triangle R_j$ (which does not involve any change to the view's join attributes), for the result of incremental computation of $\triangle R_i$, each old tuple (before the modification) of $\triangle R_j$ that occurs in μR_j has its values changed to the corresponding new tuple (after the modification).*

Note that for both missing deletion or modification update $\triangle R_j$, if the key of R_j is not in the view, then the relation R_k with its key that functionally determines the attributes of R_j will be used in applying Lemmas 5 and 7.

Theorem 1 gives the overall compensation process that is applied to resolve the maintenance anomalies.

Theorem 1. *Given α_1,...,α_n as the initial version numbers of incremental computation of update $\triangle R_i$, the compensation starts with those relations that are linked to R_i in the join graph, and proceed recursively to the rest of the relations in the same sequence as they are been queried. The compensation on the query result from R_j proceeds by first compensating with the missing updates of base relation version number $\beta_j^{min}+1$ to α_j, where β_j^{min} is the minimum queried version number of the tuples in the result from R_j, using Lemmas 5, 6 and 7. This is followed by the compensation with the interfering updates of base relation version number β_j^{max} down to α_j+1, where β_j^{max} is the maximum queried version number of the tuples in the query result from R_j, using the method discussed in [7].*

Theorem 2. *To achieve complete consistency, the view will be refreshed with the results of incremental computation in the same order as they have been queried.*

6 Comparison

Related works in this area are the Eager Compensation Algorithm (ECA and ECA^K) [10], the Strobe and C-Strobe Algorithms [11], the work of [2], the SWEEP and Nested SWEEP Algorithms [1], the work of [3], and the work of [7]. We compare these using a set of criteria, which are grouped into four categories.

The first criterion under the environment category is the number of data sources. All the approaches, except ECA, ECA^K and [2], cater for multiple data sources. The second criterion is the handling of compensation. ECA and C-Strobe send compensating queries to the data sources, while the other algorithms handle compensation locally at the view site. The latter method is preferred as compensating queries add to the overall traffic.

The first criterion under the correctness category is the correct detection of interfering updates. The compensation methods of ECA, ECA^K and Strobe are not through the detection of interfering updates, and hence they only achieve strong consistency [11]. C-Strobe does detect some interfering deletion updates which turn out to be non-interfering. The rest of the algorithms work by correctly detecting for the presence of interfering updates when messages are not misordered or lost. The next criterion is the network communication assumption. All the approaches, except [7] and the work of this paper, assume that messages are never lost and misordered. [3] also does not assume that messages are never misordered.

There are five criteria under the efficiency category. The first criterion is the number of base relations accessed per sub-query. Most of the approaches can only work by querying one base relation at a time. ECA and ECA^K can query all base relations of their single data source together. The method proposed in this paper is able to access multiple base relations within the same data source via the same query. The second criterion is the parallelism in the incremental computation of an update. Existing methods base their view maintenance querying on a left and right scan approach, and thus limit their parallelism to the two scans. In this paper, we use the join graph to guide the accessing of the base relations, and thus provide more parallelism. The third criterion is the parallelism in the incremental computation between different updates. ECA, ECA^K, Strobe and Nested SWEEP are able to process the incremental computation of different updates concurrently, but achieving only strong consistency. The rest of the methods have to process the incremental computation of different updates sequentially. [7] and the method in this paper can process the incremental computation concurrently and also achieve complete consistency. The fourth criterion is the use partial self-maintenance. ECA^K, Strobe and C-Strobe have a limited form of partial self-maintenance in that deletion update need not be processed for incremental computation. [2] can detect updates that will not affect the view. In this paper, we provide for more opportunity of partial self-maintenance. The fifth criterion is the handling of modification as one type of update. Only [7] and the method in this paper consider modification as one type of update.

The criteria under the application requirements category are the flexibility of the view definition, quiescence requirement and level of consistency achieved.

ECA^K, Strobe and C-Strobe require that the key of each base relation be retained in the view. The number of base relations in [2] is limited to two. The others have no such requirement. C-Strobe, [2], SWEEP, [3], [7] and our approach achieve complete consistency, and also do not require a quiescent state before the view can be refreshed. This does not apply to ECA, ECA^K, Strobe and the Nested SWEEP Algorithm.

7 Conclusion

The use of data source and refreshed version numbers in the maintenance algorithm allow for partial self-maintenance, as well as the accessing of multiple base relations residing at the same data source within a single query. Also, the accessing of the base relations for incremental computation are based on the join graph to avoid cartesian products. Using the join graph to determine the query path also results in more parallelism. Knowledge of referential integrity constraint imposed by the data source is used to eliminate irrelevant updates. Overall performance of the maintenance algorithm is improved by reducing the amount and size of messages sent over the network.

References

1. Agrawal, D., El Abbadi, A., Singh, A., Yurek, T.: Efficient View Maintenance at Data Warehouses. International Conference on Management of Data (1997) 417–427
2. Chen, R., Meng, W.: Efficient View Maintenance in a Multidatabase Environment. Database Systems for Advanced Applications (1997) 391–400
3. Chen, R., Meng, W.: Precise Detection and Proper Handling of View Maintenance Anomalies in a Multidatabase Environment. Conference on Cooperative Information Systems (1997)
4. Colby, L.S., Griffin, T., Libkin, L., Mumick, I.S., Trickey, H.: Algorithms for Deferred View Maintenance. International Conference on Management of Data (1996) 469–480
5. Griffin, T., Libkin, L.: Incremental Maintenance of Views with Duplicates. International Conference on Management of Data (1995) 328–339
6. Griffin, T., Libkin, L., Trickey, H.: An Improved Algorithm for the Incremental Recomputation of Active Relational Expressions. Knowledge and Data Engineering, Vol. 9 No. 3 (1997) 508–511
7. Ling, T.W., Sze, E.K.: Materialized View Maintenance Using Version Numbers. Database Systems for Advanced Applications (1999) 263–270
8. Qian, X., Wiederhold, G.: Incremental Recomputation of Active Relational Expressions. Knowledge and Data Engineering, Vol. 3 No. 3 (1991) 337–341
9. Quass, D.: Maintenance Expressions for Views with Aggregation. Workshop on Materialized Views: Techniques and Applications (1996)
10. Zhuge, Y., Garcia-Molina, H., Hammer, J., Widom, J.: View Maintenance in a Warehousing Environment. International Conference on Management of Data (1995) 316–327
11. Zhuge, Y., Garcia-Molina, H., Wiener, J.L.: The Strobe Algorithms for Multi-Source Warehouse Consistency. Conference on Parallel and Distributed Information Systems (1996)