3B2v8.06a/w (Dec 5 2003).51c +model
IS : 501
Prod.Type:FTP pp.1–19(col.fig.:NIL)
ED:PreethiNS PAGN:Jay SCAN:
ARTICLE IN PRESS
Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Information Systems ■ (■■■■) ■■■–■■■
www.elsevier.com/locate/infosys

Maintaining large update batches by restructuring and grouping ☆

Bin Liu*, Elke A. Rundensteiner, David Finkel

Department of Computer Science, Worcester Polytechnic Institute, Worcester, MA 01609-2280, USA

Received 13 October 2004; received in revised form 13 October 2005; accepted 2 February 2006
Recommended by Bindoit-Tollu

Abstract

Materialized views defined over distributed data sources can be utilized by many applications to ensure better access, reliable performance, and high availability. Technology for maintaining materialized views is thus critical for providing up-to-date results since a stale view extent may not help or even mislead these applications. State-of-the-art incremental view maintenance requires $O(n^2)$ or more remote *maintenance queries* with n being the number of data sources in the view definition. In this work, we propose two novel maintenance strategies, namely *adjacent grouping* and *conditional grouping*, that dramatically reduce the number of maintenance queries required to maintain the materialized views. This reduction in the number of maintenance queries brings the basic trade-off between the complexity of each query and the total number of maintenance queries that can be exploited to improve maintenance performance. The proposed maintenance strategies have been implemented in a working prototype system called TxnWrap. Experimental studies illustrate that our proposed strategies are able to achieve about 400% performance improvement in terms of total processing time compared with existing batch algorithms in a majority of cases.

Keywords: Materialized view maintenance; Batch maintenance; Shared common subexpressions; Grouping maintenance; Performance evaluation

1. Introduction

1.1. Materialized views and their maintenance

Materialized views [1–3] that integrate and store data from distributed data sources can be utilized by many applications including data integration services, data warehousing and decision support systems. Applying materialized views can achieve efficient access, reliable performance and high availability since applications can directly access materialized views instead of multiple distributed data sources [2]. Materialized views need to be maintained upon source changes since a stale view extent may not help or even mislead user applications. Incremental view maintenance, which aims at only computing the deltas of the view result instead of recomputing the view from scratch on data source

☆This work was supported in part by the NSF grant #IIS 9988776.
*Corresponding author. Tel.: +1 508 8316136; fax: +1 508 8315776.
E-mail addresses: binliu@cs.wpi.edu (B. Liu), rundenst@cs.wpi.edu (E.A. Rundensteiner), dfinkel@cs.wpi.edu (D. Finkel).

0306-4379/$ - see front matter
doi:10.1016/j.is.2006.02.002

changes, has been extensively studied in the past [1,3–11]. Among these works, the incremental maintenance of batches of updates [3,7,12,13] is of particular interest because it is attractive from both a resource and a performance perspective to most practical systems. The benefits are two-fold. One, better overall maintenance performance can be achieved due to utilizing cached results. Two, fewer conflicts of the maintenance tasks with users' read sessions on the view extent may arise due to significantly reducing the time period during which the view update process is being performed.

Modern data sources are becoming increasingly large over time. Rapid changes remain common even for such huge data sources. For instance, tens of thousands of transactions per hour may be experienced by Internet businesses such as amazon.com. Moreover, the data sources tend to be distributed over the network, i.e., over different branches of the enterprise or even over the WWW. All these pose new requirements for efficiently maintaining such materialized views. That is, practical systems utilizing such views must be equipped with strategies to efficiently maintain materialized views defined on distributed data sources even when faced with large batches of source updates.

Note that multiple data sources such as six or even more easily occur in real applications. For example, online travel assistant systems, such as priceline.com and travelocity.com, may integrate data from data sources supported by the different airlines, from sites for hotel rentals, from car rental companies and from sources with local sightseeing information. Or, a large enterprise may have to integrate data, such as daily sales information, from its branches located at different cities. Such an enterprise may have a large number of data sources depending on the organization and size of the company.

State-of-the-art view maintenance strategies require $O(n^2)$ (batch view maintenance) or more (i.e., sequential maintenance) *maintenance queries* [10] to remote data sources with n being the number of data sources in the view definition. In this work, we propose new maintenance approaches which require a smaller number of maintenance queries by effectively restructuring and grouping the batch view maintenance plans. Such reduction in the number of maintenance queries will in turn increase the complexity of each query. We find that our proposed view maintenance solution (in particular, the conditional grouping strategies) may significantly outperform existing batch view maintenance strategies (around 400% improvement) in a majority of the cases.

1.2. Motivating example

We use the following example to illustrate two of the most prevailing classes of state-of-the-art incremental view maintenance strategies, namely, sequential maintenance and batch maintenance. The basic trade-off that will be exploited in our work is revealed by analyzing these two strategies. Fig. 1 describes three data sources with one relation each that will be used in the example. A view *Tour-Customer* is defined as depicted in Query 1.

CREATE VIEW *Tour − Customer* AS
SELECT *C.Name, C.Age, T.TourID, F.FlightNo, F.Dest*
FROM *Cust C, FlightRes F, Tour T*
WHERE *C.Name = F.Name AND F.Name = T.CustName*

(1)

1.2.1. Sequential maintenance

Sequential maintenance refers to maintaining one single source update at a time. As one typical example of such strategy, we illustrate the SWEEP algorithm introduced in [1]. For example, one data update "U_1 = *Insert into Cust Values* ('*Ben*', *28*, '*WPI*', *6136*)" happened at R_1. In order to determine the delta effect on the view extent, this requires us to send two maintenance queries, one to R_2 and another to R_3. In this case, one maintenance query (Query 2) is generated based on U_1 and sent to source R_2. After we get the result, say ('Ben', 28, 'AA69', 'Mia'), another maintenance query (Query 3) will be generated and sent to R_3 to get the delta change on the view extent.

SELECT *'Ben' as Name, 28 as Age F.FlightNo, F.Dest*
FROM *FlightRes F*
WHERE *F.Name = 'Ben'* (2)

R_1: Cust (Name, Age, Address, Phone)
R_2: FlightRes (Name, FlightNo, Source, Dest)
R_3: Tour (TourID, CustName, Type, Days)

Fig. 1. Description of data sources.

U_1: Insert ('Ben', 28, 'WPI', 6136) into Cust
U_2: Insert ('Tom',D L169, 'Lax', 'Bos') into FlightRes
U_3: Insert (63, 'Tom', 'Lux', 10) into Tour
U_4: Insert ('Joe', AA189, 'Bos', 'Paris') into FlightRes
U_5: Delete ('Ken', 27, 'WPI', 5857) from Cust

Fig. 2. Updates of data sources.

$$\begin{array}{rl} \text{SELECT} & 'Ben' \text{ as } Name,\ 28 \text{ as } Age,\ T.TourID \\ & 'AA69' \text{ as } FlightNo,\ 'Mid' \text{ as } Dest \\ \text{FROM} & Tour\ T \\ \text{WHERE} & T.CustName\ =\ 'Ben' \end{array} \qquad (3)$$

Thus, to maintain one source update using SWEEP, we may have to send maintenance queries to all data sources besides the one from the source update originated to compute the delta effect on the view extent. If multiple source updates need to be maintained, as illustrated in Fig. 2, we would repeat this process for each update until all updates have been processed.[1]

1.2.2. Batch maintenance

Batch maintenance refers to maintaining the view extent using source-specific *deltas* [12,13] where one *source delta* describes a set of changes made to a data source in a certain time period. For example, instead of maintaining five updates listed in Fig. 2 individually as described above, we construct a delta specific for each source. Thus, $\Delta R_1 = \{+$('Ben', 28, 'WPI', 6136), $-$('Ken', 27, 'WPI', 5857)$\}$, $\Delta R_2 = \{+$('Tom', DL169, 'Lax', 'Bos'), $+$('Joe', AA189, 'Bos', 'Paris')$\}$, and $\Delta R_3 = \{+$(63, 'Tom', 'Lux', 10)$\}$. Here for simplicity, we use '+' to represent an insert operation and '−' to denote a delete operation. Thereafter, the incremental view extent (view delta) for all five updates can be logically computed in three steps (one step per source delta). Within each step, maintenance queries are built based on the source-specific delta and submitted to the other data sources to compute the maintenance result. Here, each source delta represents the updates at a logical level, we separate the processing of insert and delete operations in the implementation.

[1] Concurrent source updates could happen during the maintenance process. Thus additional concurrency control is necessary to keep the view extent consistent [11]. We discuss this with more detail in Section 5.1.

1.2.3. Observation

Based on the above discussion, we will now describe the basic trade-off that can be observed. Batch maintenance has been shown to be more efficient in terms of the total processing time when maintaining a large set of source updates [3,7,12,13]. Sequential maintenance involves many maintenance queries in the style similar to Queries 2 and 3 in our example to be sent, with each maintenance query reflecting a single source update. Here, the total number of maintenance queries required for sequentially maintaining k source updates may in the worst-case be $k * (n-1)$ with n being the number of data sources in the view definition. Clearly, batch maintenance can improve this query workload due to the number of maintenance queries cannot exceed $n * (n-1)$ (see Section 2). Given that the number of data sources (n) usually is much smaller than the number of update tuples (k), i.e., n is usually less than 10 while k can be thousands or even millions, batch maintenance requires a much smaller number of maintenance queries. However, each maintenance query utilized in the batch maintenance process is now more complex, because it now must reflect a set of source updates.[2]

This opens the opportunity to group multiple source updates so to construct one combined maintenance query for this batch update set. The goal is to develop a batch method that may outperform the sequential process of handling each individual source update one by one. Exploitation of this trade-off between the number of maintenance queries and their complexity (expressed as query and result sizes) to improve the view maintenance performance is the main focus of this paper.

1.3. Contributions

We have illustrated the idea of reducing the number of maintenance queries when maintaining batches of updates through a running example in an earlier poster paper [14]. In this journal manuscript, we now provide details of the proposed maintenance strategies, we introduce cost models and their analysis, and we also present a comprehensive experimental study. Our main contributions in this work include:

1. We propose an *adjacent grouping* strategy that

[2] The methods of composing maintenance queries for a set of source updates will be discussed and evaluated in Section 7.2.

exploits the regularity of the structure of a batch maintenance plan to share the accesses to remote data sources.
2. We also propose a *conditional grouping* strategy that groups heterogeneous deltas in a batch maintenance plan. It is able to reduce the number of maintenance queries to O(n) with n being the number of data sources in the view definition, regardless of how many source updates need to be maintained.
3. We provide a high level description of the costs of the proposed strategies in order to be able to analyze the strategies and to reveal the basic trade-off among the alternate approaches when maintaining a large batch of source updates.
4. We have implemented the proposed strategies as well as state-of-the-art algorithms from the literature in a working prototype. This enables us to conduct performance studies of the proposed techniques and to compare our solution against these existing [1,13].
5. We report on the extensive experimental study we have conducted. The experimental results show a significant performance improvement (up to 400%) gained by the *conditional grouping* approach in a majority of cases considered.

The rest of the paper is organized as follows. Section 2 describes an abstraction that we present to capture the essence of the state-of-the-art batch view maintenance process. Sections 3 and 4 describe the proposed maintenance strategies, respectively. Section 5 discusses issues related to generalizing our proposed strategies. A cost-based analysis is provided in Section 6. Section 7 discusses the experimental results, while related work and conclusions are given in Sections 8 and 9, respectively.

2. Abstract batch view maintenance

For ease of describing our proposed maintenance strategies, we first use an abstraction to capture the essence of the batch view maintenance process. Assume a materialized view V is defined as an n-way join on n distributed data sources. That is, V is denoted by $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$.[3] There are n source deltas ΔR_i, one for each source R_i with $1 \leqslant i \leqslant n$) that need to be maintained. As was mentioned in Section 1.2.2, each ΔR_i denotes the changes (the collection of insert and delete tuples) on R_i at a logical level. An actual maintenance query will be issued separately, that is, one for insert tuples and one for delete tuples.

Given the above notations, the batch view maintenance process can be represented by Eq. (4). Here R_i refers to the original data source state without any changes from ΔR_i, while $R_i' = R_i \cup \Delta R_i$ reflects the state of the data source R_i after applying the change ΔR_i. The discussion of the correctness of this batch view maintenance can be found in [12,13]. Note that concurrency control strategies, either compensation-based [1,10,15] or multiversion-based [6], need to be employed if other source updates happen concurrently. Without loss of generality, we now focus on the maintenance queries and ignore any concurrent source updates for the moment. The discussion of handling concurrent updates is deferred to Section 5.1.

$$\begin{aligned}\Delta V = {} & \Delta R_1 \bowtie R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n \\ & \cup R_1' \bowtie \Delta R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n \\ & \cup \cdots \\ & \cup R_1' \bowtie R_2' \bowtie R_3' \bowtie \cdots \bowtie \Delta R_n. \end{aligned} \tag{4}$$

We call Eq. (4) a *batch maintenance plan*. It specifies at an abstract level how to incrementally maintain the view. Each "line" in Eq. (4) is referred to as a *maintenance step*. $\Delta R_1 \bowtie R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n$ is one example of such a step. A maintenance query needs to be composed for each join ($\bowtie$) either from the source delta (ΔR_i) or the intermediate results from previous queries, i.e., the query result of $\Delta R_1 \bowtie R_2$. For ease of description, we may interchange the term 'maintenance query' and 'delta' (either ΔR_i or the result of a maintenance query) in the subsequent discussion. Two ways of composing a maintenance query from a delta will be discussed in Section 7.2. Note that the evaluation of each maintenance step is expected to start from the source delta (ΔR_i) and to visit all the other data sources. This is because each source delta is usually much smaller in size in terms of the number of tuples compared to the size of a data source. Seen from the above discussion, $n*(n-1)$ (O(n^2)) maintenance queries may be required for the batch maintenance to compute the delta change (ΔV) of the view extent.

As an example, the source updates (deltas) described in Section 1.2.2 on the three-way join view (Query 1) can be maintained in the following three maintenance steps: $(\Delta R_1 \bowtie R_2 \bowtie R_3) \cup (R_1' \bowtie \Delta R_2 \bowtie R_3) \cup (R_1' \bowtie R_2' \bowtie \Delta R_3)$.

[3] Discussions of the handling of more general SPJ views will be deferred to Section 5.2.

However, two questions remain. First, is it possible to further reduce the number of maintenance queries, say to less than $O(n^2)$? Second, does a lower number of maintenance queries imply a reduction in total maintenance time? Put differently, this raises the underlying question what the key factors are that affect the view maintenance performance. The remaining sections of this paper explore these questions. We use the batch maintenance plan (Eq. (4)) as the baseline algorithm based on which we will propose a variety of different strategies.

Note that traditional distributed query optimization techniques [16] could be applied to improve view maintenance performance, e.g., to select an optimized join execution order for each maintenance step. Clearly, this is orthogonal to what we will explore here. Instead our focus is to find new maintenance strategies by restructuring and grouping maintenance queries. These cost-based optimization techniques can thereafter also be applied on our proposed strategies. Readers may consult [17] for more discussions on this direction. In the view maintenance context, finding the common expressions such as $R_3 \bowtie R_4$, which is investigated in traditional multiple query optimization [18], may not be beneficial. The reason is that the common parts may be too large to be evaluated if they are not first joined with the (typically much smaller) delta.

3. Adjacent grouping

One way to reduce the number of maintenance queries is to exploit the regularity in a maintenance plan to promote sharing of common accesses to data sources. Studying the batch maintenance plan (Eq. (4)), we observe that a large number of common data source accesses exists in different maintenance steps. For example, the first two maintenance steps both have $R_3 \bowtie R_4 \bowtie \cdots \bowtie R_n$ in common, while the second and the third steps both have R_1' and $R_4 \bowtie \cdots \bowtie R_n$. Thus, if we share the accesses to these common data sources, the number of maintenance queries (join operations) would be reduced.

The matrix-like abstraction of the batch maintenance plan as depicted in Fig. 3 highlights the regularity in terms of the common items between adjacent maintenance steps. The basic idea underlying the adjacent grouping strategy is illustrated in Fig. 3. Namely, we divide maintenance steps and group the deltas from different maintenance steps along the main diagonal. Then we share the accesses to common data sources.

For example, Fig. 3(a) illustrates the grouping by two. Here, the first two maintenance steps are rewritten into one expression, namely, $(\Delta R_1 \bowtie R_2 \cup R_1' \bowtie \Delta R_2) \bowtie R_3 \bowtie \cdots \bowtie R_n$. Clearly, the total number of maintenance queries for evaluating these two maintenance steps is reduced from $2 * (n-1)$ to n. While for the third and the fourth steps, we rewrite them as $R_1' \bowtie R_2' \bowtie (\Delta R_3 \bowtie R_4 \cup R_3' \bowtie \Delta R_4) \bowtie \cdots \bowtie R_n$, and so on. Thus, only $(n/2) * n$ maintenance queries are required if we group every two maintenance steps with n being an even number. Grouping maintenance steps by three can be done in a similar manner (see Fig. 3(b)), and so on.

If we divide steps equally, i.e., we group every m ($m < n$) adjacent steps along the main diagonal. Let us denote the total number of maintenance queries by N_m. Here, $\Re$ includes the leftover factors of n that cannot be divided by m. The formula N_m is derived assuming we group every group of m maintenance steps together into one query. For example, assume the view is defined on six data sources ($n = 6$), and we group every two adjacent maintenance steps together ($m = 2$). Then, the maintenance steps are divided into

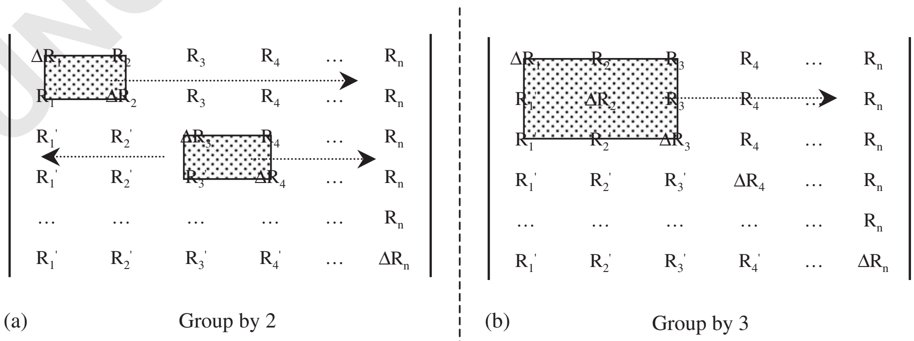


Fig. 3. Group adjacent maintenance steps.

three groups ($n/m = 3$). In each group, the number of maintenance queries corresponds to $m(m-1)$ (the $m \times m$ matrix) and the rest $(n-m)$. The total number of maintenance queries corresponds to the sum of the counts for the three groups.

$$N_m = \left\lfloor \frac{n}{m} \right\rfloor (m(m-1) + (n-m)) + \Re,$$

$$\Re = \left(n - \left\lfloor \frac{n}{m} \right\rfloor m\right)(n-1).$$

We can solve the equation $\partial N_m / \partial m = 0$ to find the number m that minimizes N_m. If we assume n is perfectly divided by m, then $\partial N_m / \partial m$ equals $n^2/m^2 - n$. As can be seen, the total number of queries N_m reaches its minimum when m is around $\sqrt{n}$. Note that other grouping heuristics may also be possible. For example, we could group maintenance steps unevenly based on the estimated respective delta sizes.

By replacing m with $\sqrt{n}$, the total number of maintenance queries now becomes $O(n^{3/2})$. However, this approach only combines temporary results that have the same schema. For example, the combination of the result from $\Delta R_1 \bowtie R_2$ and $R_1' \bowtie \Delta R_2$. This limits the type of query shrinking that can be considered. Below, we propose a new solution strategy that relaxes the constraint of only combining (unioning) deltas with the same schema. This solution dramatically reduces the number of maintenance queries.

4. Group heterogeneous deltas

4.1. Basic notations

We use $\oplus$ to represent the operation that takes a list of deltas as input, possibly with different schemas, and combines (union) them together. For example, $\oplus([\Delta R_1], \Delta R_2, \Delta R_3)$ equals a combined delta containing both ΔR_2 and ΔR_3. The tuples contained in the brackets are not included in the union. At this point, we focus on the logical expressions only. The engineering problem of how to implement the union of deltas with different schemata will be discussed in Section 4.4. A join operator applied to an expression containing the $\oplus$ operator corresponds to the computation of each delta in the result set produced by the $\oplus$ expression. For example, $\oplus([\Delta R_1], \Delta R_2, \Delta R_3) \bowtie R_i$ equals the collection of result deltas $\Delta R_2 \bowtie R_i$ and $\Delta R_3 \bowtie R_i$, henceforth represented by $\{\Delta R_2 \bowtie R_i,\ \Delta R_3 \bowtie R_i\}$. To further simplify the notation, we may omit the $\bowtie$ symbol in the result set whenever the context is clear, i.e., $\{\Delta R_2 \bowtie R_i, \Delta R_3 \bowtie R_i\}$ will be simplified to $\{\Delta R_2 R_i, \Delta R_3 R_i\}$.

We assume that each ΔR_i has been processed at R_i before it is reported to the view manager for maintenance. That is, insert tuples in ΔR_i have already been inserted into R_i, while delete tuples in R_i have already been deleted from R_i. Thus, each maintenance query will be evaluated on R_i' instead of on R_i. Compensations are needed to get the maintenance query results based on the original state R_i. We introduce θ_i to represent the compensation process using ΔR_i. For example, assuming $\mathscr{D}$ is a delta (either ΔR_i or a previous maintenance query result), then $\theta_i(\mathscr{D} \bowtie R_i') = \mathscr{D} \bowtie R_i' - \mathscr{D} \bowtie \Delta R_i = \mathscr{D} \bowtie R_i$. The rationale behind this compensation process can be illustrated by : $\mathscr{D} \bowtie R_i' = \mathscr{D} \bowtie (R_i \cup \Delta R_i) = \mathscr{D} \bowtie R_i \cup \mathscr{D} \bowtie \Delta R_i$. Note that both $\mathscr{D}$ and ΔR_i are available at the view manager. Thus such compensation can be computed locally at the view manager when we get the result of $\mathscr{D} \bowtie R_i'$.

4.2. A greedy grouping approach

To maintain n source deltas $\Delta R_1, \Delta R_2, \Delta R_3, \ldots, \Delta R_n$ on an n-way join view, one extreme solution is to group all the intermediate results (deltas) computed in the maintenance steps (ΔR_i or any previous maintenance query result) to construct a combined query. We thus need to access each data source (R_i, $1 \leqslant i \leqslant n$ only) once to evaluate the maintenance process (see Eq. (4)). This way, we only require n combined maintenance queries (the theoretically minimal number). These n combined maintenance queries will be evaluated in a sequential manner by sending them to the data sources $R_1, R_2, \ldots, R_n$, respectively. These queries are represented by $Q_1, Q_2, \ldots, Q_n$, as further described below. The overall approach is sketched in Algorithm 1, while each of its steps is further elaborated upon below.

Algorithm 1. GreedyGrouping(*s_Deltas*)

/****s_Deltas***: *An array list of source deltas, with* *s_Deltas*[*i*] $= \Delta R_i$ *initially* */

for ($i = 1;\ i \leqslant n;\ i++$) **do**
  Compose maintenance query Q_i from *s_Deltas*,
  except for *s_Deltas*[*i*];
  Send Q_i to R_i;
  Compensate query result of Q_i;

5: Update *s_Deltas* based on the compensated query result;
6: **end for**
7: Compose ΔV by unioning deltas in *s_Deltas*;

The composition of each query Q_i (step 2 in Algorithm 1) and the corresponding compensation processes of each query result (step 4 in Algorithm 1) are described below.

- Q_1: We combine all source deltas (except ΔR_1) and send them to the data source R_1. We evaluate the query result. This process can be expressed by $\oplus([\Delta R_1], \Delta R_2, \Delta R_3, \ldots, \Delta R_n) \bowtie R'_1 = \{\Delta R_1, R'_1\Delta R_2, R'_1\Delta R_3, \ldots, R'_1\Delta R_n\}$.
- Q_2: We combine all result deltas from Q_1 except the one containing ΔR_2 and send it to R_2 (referred as evaluation).
  - *Evaluation*: $\oplus(\Delta R_1, [R'_1\Delta R_2], R'_1\Delta R_3, \ldots, R'_1\Delta R_n) \bowtie R'_2 = \{\Delta R_1 R'_2, R'_1\Delta R_2, R'_1 R'_2\Delta R_3, \ldots, R'_1 R'_2\Delta R_n\}$.

  After we get the query result, we compensate it using ΔR_2 for those result deltas containing ΔR_1 (referred as compensation).
  - *Compensation*: $\{\theta_2(\Delta R_1 R'_2), R'_1\Delta R_2, R'_1 R'_2\Delta R_3, \ldots, R'_1 R'_2\Delta R_n\} = \{\Delta R_1 R_2, R'_1\Delta R_2, R'_1 R'_2\Delta R_3, \ldots, R'_1 R'_2\Delta R_n\}$.

We now describe this process in general for any query Q_i $(1 < i \leqslant n)$:

- for any query Q_i $(1 < i \leqslant n)$, we combine the results from query Q_{i-1}—except the one containing ΔR_i—and then ship them to the data source R_i for evaluation.
  - *Evaluation*: $\oplus(R'_1 R'_2 \ldots \Delta R_k R_{k+1} \ldots R_{i-1}$ $(1 \leqslant k < i)$, $[R'_1 R'_2 \ldots R'_{i-1}\Delta R_i]$, $R'_1 R'_2 \ldots R'_{i-1}\Delta R_k$ $(i < k \leqslant n)) \bowtie R'_i = \{R'_1 R'_2 \ldots \Delta R_k R_{k+1} \ldots R_{i-1} R'_i$ $(1 \leqslant k < i)$, $R'_1 R'_2 \ldots R'_{i-1}\Delta R_i$, $R'_1 R'_2 \ldots R'_{i-1} R'_1 \Delta R_k$ $(i < k \leqslant n)\}$.

  The result deltas that contain delta ΔR_j $(j < i)$, which correspond to any data source that has already been visited before, will be compensated using ΔR_i, as described next:
  - *Compensation*: apply θ_i to $R'_1 R'_2 \ldots \Delta R_k R_{k+1} \ldots R_{i-1} R'_i$ $(1 \leqslant k < i)$, we get the result of Q_i as $\{R'_1 R'_2 \ldots \Delta R_k R_{k+1} \ldots R_i$ $(1 \leqslant k < i)$, $R'_1 R'_2 \ldots R'_{i-1}\Delta R_i$, $R'_1 R'_2 \ldots R'_i \Delta R_k$ $(i < k \leqslant n)\}$.

Thus, after the nth query Q_n (replacing i with n), we get $\{R'_1 R'_2 \ldots \Delta R_k R_{k+1} \ldots R_n$ $(1 \leqslant k < n)$, $R'_1 R'_2 \ldots R'_{i-1}\Delta R_n\}$. By listing individual result deltas, we get $\{\Delta R_1 \bowtie R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n,\ R'_1 \bowtie \Delta R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n,\ R'_1 \bowtie R'_2 \bowtie \Delta R_3 \bowtie \cdots \bowtie R_n, \ldots, R'_1 \bowtie R'_2 \bowtie R'_3 \bowtie \cdots \bowtie \Delta R_n\}$. This is same as the equation we have shown for the batch maintenance plan (Eq. (4)) if we union these deltas together. The correctness of the approach is shown by the fact that the nth query result is the same as Eq. (4). Thus, by issuing only n combined queries to the underlying data sources, we can indeed compute theincremental view extent ΔV.

One potential weakness of this approach is the possibility of a *large intermediate result set*, in the case that no join condition exists between some of the intermediate results and the data source. For example, assume we send $\oplus([\Delta R_1], \Delta R_2, \Delta R_3, \ldots, \Delta R_n)$ to data source R_1 in Q_1. Assume that only R_2 has a join condition with R_1 given the view is defined by $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$. Thus, to evaluate the result $\Delta R_k \bowtie R'_1$ $(3 \leqslant k \leqslant n)$, we may have to compute the Cartesian product instead between R_1 and those other relations. Given that the size of each data source may be huge, this approach may not always be very beneficial in practice.

4.3. Conditional grouping approach

To address the above mentioned problem of potentially large intermediate results arising in the greedy grouping approach, we now propose the *conditional grouping* strategy. The basic idea is to make use of join conditions in the view definition. This is because a maintenance query based on join conditions is generally much cheaper to process than one based on Cartesian products.

The main steps of the conditional grouping approach are outlined in Algorithm 2, while each subroutine is thereafter described in more detail. The overall maintenance process is divided into two phases, called the *scroll up* phase and the *scroll down* phase. In each phase, we issue $n-1$ queries by grouping the deltas with common join conditions for a data source together.

Algorithm 2. ConditionalGrouping(*s_Deltas*)

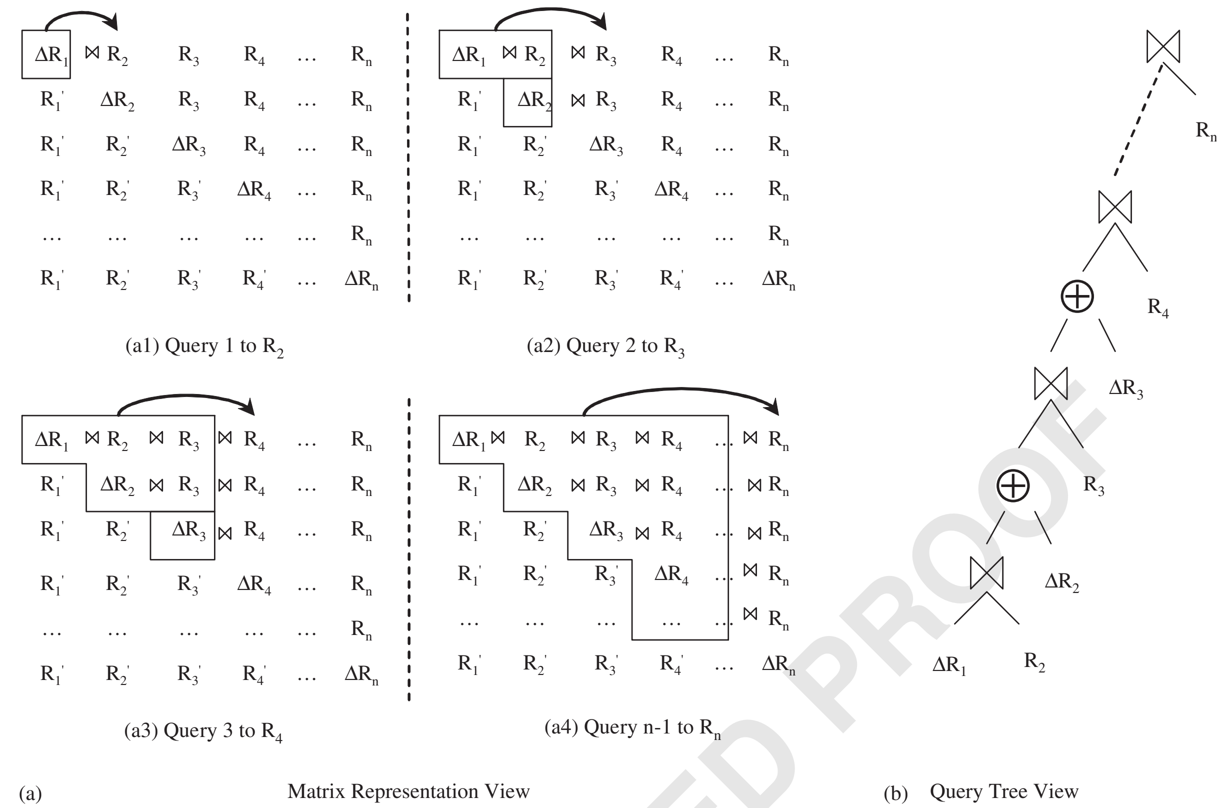


Fig. 4. Scroll up phase: (a) matrix representation view; (b) query tree view.

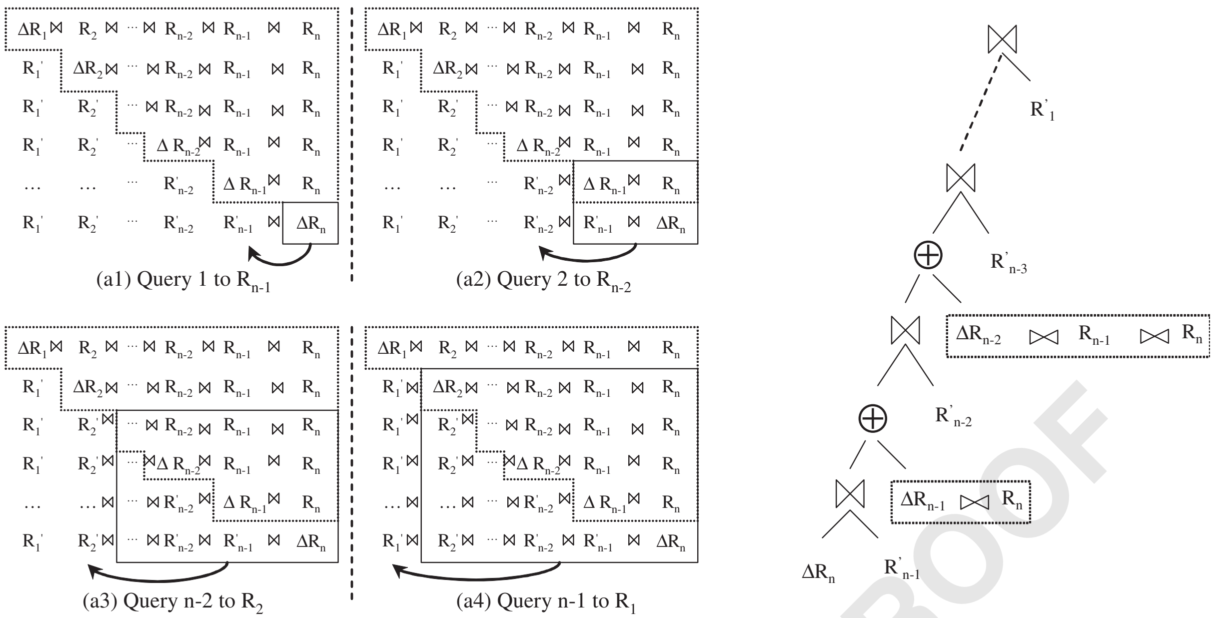
/* ***s_Deltas***: *An array list of source deltas, with* $s\_Deltas[i] = \Delta R_i$ *initially.* */
1: $s\_Deltas = scroll\_up\ (s\_Deltas)$;
2: $s\_Deltas = scroll\_down\ (s\_Deltas)$;
3: Compose ΔV by unioning deltas in *s_Deltas*;

4.3.1. Scroll up phase

The $n-1$ queries in this phase are represented by $Q_1^u, Q_2^u, \ldots, Q_{n-1}^u$, respectively. They are evaluated sequentially. We describe each query below.

- Q_1^u: We send ΔR_1 to R_2, evaluate $\Delta R_1 \bowtie R_2'$ and then compensate the result using ΔR_2. These two steps can be expressed by $\oplus(\Delta R_1) \bowtie R_2' = \Delta R_1 R_2'$ and $\theta_2(\Delta R_1 R_2') = \Delta R_1 R_2$ (see Fig. 4(a1)).
- Q_2^u: We union the first query ($\Delta R_1 R_2$) with ΔR_2 and send this collection to R_3. We then compensate this query result using ΔR_3. The following steps capture this process: (1) $\oplus(\Delta R_1 R_2, \Delta R_2) \bowtie R_3' = \{\Delta R_1 R_2 R_3', \Delta R_2 R_3'\}$, and (2) $\{\theta_3(\Delta R_1 R_2 R_3'), \theta_3(\Delta R_2 R_3')\} = \{\Delta R_1 R_2 R_3, \ \Delta R_2 R_3\}$ (see Fig. 4(a2)).
- Q_3^u: Similarly, the third query is expressed as (1) $\oplus(\Delta R_1 R_2 R_3, \ \Delta R_2 R_3, \ \Delta R_3) \bowtie R_4'$, and (2) then we apply θ_4 to compensate the query results. We then get $\{\Delta R_1 R_2 R_3 R_4, \Delta R_2 R_3 R_4, \Delta R_3 R_4\}$ as the result of the third query (see Fig. 4(a3)).
- To generalize, we do the following three operations for any query Q_i^u $(1 < i \leqslant n-1)$.
  - Compose maintenance query Q_i^u by combining Q_{i-1}^u query result with ΔR_i. We get $\oplus(\Delta R_1 R_2 R_3 \ldots R_i, \Delta R_2 R_3 \ldots R_i, \ldots, \Delta R_{i-1} R_i, \Delta R_i)$.
  - Send Q_i^u to R_{i+1} and evaluate it against R_{i+1}. We get the query result $\{\Delta R_1 R_2 R_3 \ldots R_i R_{i+1}', \Delta R_2 R_3 \ldots R_i R_{i+1}', \ldots, \Delta R_{i-1} R_i R_{i+1}', \Delta R_i R_{i+1}'\}$.
  - Compensate the result using ΔR_{i+1} (θ_{i+1}). We finally get $\{\Delta R_1 R_2 R_3 \ldots R_i R_{i+1}, \Delta R_2 R_3 \ldots R_i R_{i+1}, \ldots, \Delta R_{i-1} R_i R_{i+1}, \Delta R_i R_{i+1}\}$.

After processing query Q_{n-1}^u, we get $\{\Delta R_k \bowtie R_{k+1} \bowtie \cdots \bowtie R_n \ (1 \leqslant k \leqslant n)\}$ as the result of the scroll up phase (Fig. 4(a4)). Note that in Fig. 4,



(a) Matrix Representation View (b) Query Tree View

Fig. 5. Scroll down phase: (a) matrix representation view; (b) query tree view.

deltas represented by different rectangle boxes are unioned ($\oplus$) into one combined delta and sent to the data source. Fig. 4(b) illustrates the corresponding left-deep query tree representation of this process. In this process, queries are evaluated in a bottom-up manner.

4.3.2. Scroll down phase

The $n-1$ queries in the scroll down phase are represented by $Q_1^d, Q_2^d, \ldots, Q_{n-1}^d$, respectively. These queries are built based on the result obtained from the scroll up phase. Below, we again describe this phase by its queries.

- Q_1^d: We evaluate $\oplus(\Delta R_n) \bowtie R'_{n-1}$ and get $R'_{n-1}\Delta R_n$. Note that no compensation needs to be applied in this phase (Fig. 5(a1)).
- Q_2^d: We combine the result of the first query $(R'_{n-1}\Delta R_n)$ with the result from the scroll up phase containing ΔR_{n-1} ($\Delta R_{n-1} R_n$ in this case). This results in $\oplus(R'_{n-1}\Delta R_n,\ \Delta R_{n-1} R_n)$. We then send it to R_{n-2}, evaluate $\oplus(R'_{n-1}\Delta R_n,\ \Delta R_{n-1} R_n) \bowtie R'_{n-2}$ and get $\{R'_{n-2}R'_{n-1}\Delta R_n, R'_{n-2}\Delta R_{n-1} R_n\}$ (Fig. 5(a2)).
- To generalize, we take the following two steps for any query Q_i^d $(1 < i \leqslant n-1)$.
  - Combine previous query (Q_{i-1}^d) result $(\{R'_{n-i+1}R'_{n-i+2}\ldots\Delta R_{n-k+1}R_{n-k+2}\ldots R_n, 1 \leqslant k \leqslant i-1\})$ with the result from the scroll up phase that contains ΔR_{n-i+1} $(\Delta R_{n-i+1} R_{n-i+2}\ldots R_n)$.
  - Submit the combined query to R_{n-i} and evaluate it against R_{n-i}. We get result $\{R'_{n-i}R'_{n-i+1}R'_{n-i+2}\ldots \Delta R_{n-k+1}R_{n-k+2}\ldots R_n\ (1 \leqslant k \leqslant i)\}$.

Thus, after processing query Q_{n-1}^d, we get $\{R'_1R'_2R'_3\ldots\Delta R_{n-k+1}R_{n-k+2}\ldots R_n \quad (1 \leqslant k \leqslant n-1)\}$. As we can see, this equals $\{\Delta R_1 \bowtie R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n,\ R'_1 \bowtie \Delta R_2 \bowtie R_3 \bowtie \cdots \bowtie R_n, R'_1 \bowtie R'_2 \bowtie \Delta R_3 \bowtie \cdots \bowtie R_n, \ldots, R'_1 \bowtie R'_2 \bowtie R'_3 \bowtie \cdots \bowtie \Delta R_n\}$ (see Fig. 5(a4)). By unioning the results in this collection, we clearly obtain Eq. (4) again. Fig. 5(b) depicts the query tree representation of the scroll down process. Similarly, the query tree is again evaluated in a bottom up fashion. Note that the join(s) inside the box (right-hand side of $\oplus$ operator) have already been evaluated in the scroll up phase.

To summarize, the *scroll up* phase calculates the upper part along the main diagonal of the batch maintenance plan using $n-1$ queries (Eq. (4)), while the *scroll down* phase computes the remaining part in another $n-1$ queries.

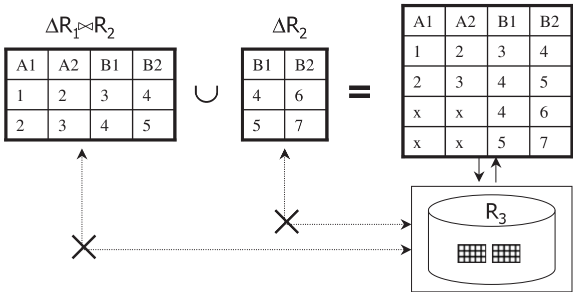


Fig. 6. Example of unifying deltas with different schemata.

4.4. Grouping deltas together

Next, we address the engineering problem of combining the heterogeneous deltas. For example, consider building a combined delta for $\oplus(\Delta R_1 \bowtie R_2, \Delta R_2)$. If the query engine at the data source were advanced, it could exploit the similarity among the deltas to scan the source relation once when processing this $\oplus$ operator even if we send the results separately. However, query engines may not be that advanced. Thus, we instead propose a non-intrusive method to address this issue of unifying various deltas from different data sources. This guarantees the general applicability of our method.

The basic idea is to construct one large table that contains the schema of different deltas and fill the respective unrelated fields with default values. This table is shipped to the data source as one large delta and evaluated in one set. The view manager splits the large query result back into different deltas per source. We may append certain identification related information to the delta so we can split the query result back into deltas more easily.

As shown in Fig. 6, instead of sending delta tables $\Delta R_1 \bowtie R_2$ and ΔR_2 to the data source R_3 separately, we first build a union table which contains the information of both deltas and then send this combined delta together to R_3 to evaluate the maintenance result in one pass. For the issues of building a maintenance query from a delta table, either a composite SQL query or temporary table approach can be applied based on whether the data source is cooperative or not. We will discuss this in more detail in Section 7.2.

5. Generalizing the maintenance strategies

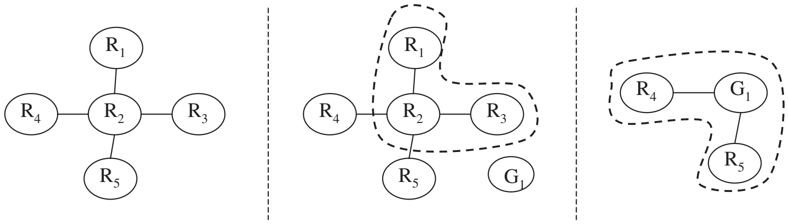
5.1. Handling concurrent updates

In the grouping strategies proposed above, we have thus far assumed that there is no concurrency interfering with a given view maintenance plan. This can be easily achieved by a multiversion system [6] because we can always retrieve the right data source states from the versioned source data. However, if a compensation-based approach were to be used, such as [15], concurrent updates would have to be considered. To address this, we now propose a method to maintain the view even in concurrent environments.

We use two vectors to hold source updates: the *current vector* (CV) holds the *deltas* per source that currently are being maintained, while the *concurrent vector* (CRV) holds all updates that occur concurrently to the current maintenance plan. Initially, CRV is empty because all source updates will be put into CV. After we begin to maintain the deltas in CV, newly incoming updates will be put into CRV. As usual, we use R_i $(1 \leqslant i \leqslant n)$ to represent its original data source state, and R_i' $(R_i' = R_i + \Delta R_i)$ to represent the state that incorporates the effect of source updates in CV. We use R_i^c to represent the state that reflects $R_i' + \Delta R_i^c$, where ΔR_i^c denotes the corresponding deltas accumulated in CRV that are concurrent with the current maintenance plan.

As done in most of the literature [1,10], we assume that all message transfers between sources and the view manager use a FIFO scheme. That is, all updates that happen on a data source after the evaluation of the maintenance query upon this source will also arrive at the view manager (vector CRV) after the arrival of the result of this maintenance query. That is, we can use *deltas* in both vectors $(\Delta R_i, \Delta R_i^c)$ to restore the appropriate data source states (either R_i' or R_i), when the view manager gets the result of a maintenance query.

Now, we are ready to extend the original compensation operator θ_i to θ_i^{i+c} and θ_i^c. Here θ_i^{i+c} compensates the query result using $\Delta R_i + \Delta R_i^c$. That is $\theta_i^{i+c}(\mathscr{D} \bowtie R_i^c) = \mathscr{D} \bowtie R_i$. The θ_i^c compensates the result using ΔR_i^c. That is, $\theta_i^c(\mathscr{D} \bowtie R_i^c) = \mathscr{D} \bowtie R_i'$. Given that, the above *conditional grouping* algorithm can be adapted as follows for a concurrent environment: (1) For any query Q_i^u in the *scroll up* phase, we use $\theta_{i+1}^{(i+1)+c}$ to compensate the result. (2) For any query Q_i^d in the *scroll down* phase, we then use θ_{n-i}^c to compensate the result.



(a) A Star-View Definition (b) Maintain R_1-R_2-R_3 (c) Maintain R_4-G_1-R_5

Fig. 7. Handling general view definitions: (a) a star-view definition; (b) maintain R_1–R_2–R_3; and (c) maintain R_4–G_1–R_5.

Thus, we compute view delta (ΔV) which exactly only reflects the source updates in CV. Once we refresh the view extent, we simply move the deltas in CRV to CV and set $R_k = R'_k$ ($1 \leqslant k \leqslant n$). Thereafter, we can repeat the maintenance process for the next set of collected updates.

5.2. Handling general view definitions

The grouping strategies we have described so far have 0 on linear join view definitions, i.e., $R_1 \bowtie R_2 \bowtie \cdots \bowtie R_n$, as also implicitly assumed by most previous works in the literature [1,3,13,19]. However, practical view definitions may have arbitrary shapes beyond just linear join view definitions, i.e., they may include acyclic and in some cases even cyclic join relationships within the view definitions. For these general view definitions, we use *view graphs* to represent the view definitions. A node in a view graph represents the data source, while an edge denotes the join conditions that appear in the view definition. We then propose the following view graph transformation technique to maintain general join view definitions: (1) Find a linear path and apply the grouping strategies to the parts of the view definition related to the linear path. (2) Transform the graph using the partial results from (1) and recursively apply this find-and-transform technique.

For example, Fig. 7(a) represents an acyclic view (V) that involves five data sources. To maintain this view using grouping strategies, we first find a linear path, i.e., $R_1 \bowtie R_2 \bowtie R_3$. For simplicity, we use G_1 to represent this part of the view definition. We then maintain G_1 by the grouping strategy (Fig. 7(b)). After that, we transform the original graph by replacing the linear path using G_1. Here, edges that connect G_1 to any of the nodes in the linear path are changed to G_1. Multiple edges between two nodes are merged into one. The delta change of G_1 (ΔG_1) can be got from the maintenance result of G_1 (Fig. 7(c)). We repeat the above processes until we get the final view maintenance result ΔV. Note that we do not materialize G_1. Thus a maintenance query involving G_1 (or $G'_1 = G_1 + \Delta G$) has to be sent to each of the data sources, i.e., $R_1 \bowtie R_2 \bowtie R_3$ in this case.

6. Cost model and analysis

We now introduce cost models we have developed to analyze the proposed maintenance strategies. In this work, we focus on the following two cost metrics since they are the main factors that affect the overall performance: the cost of transferring data between the view manager and the data sources, and the cost of evaluating maintenance queries (join operations) at the data sources. We note that the compensation cost would be rather small if we were to apply a multiversion-based concurrency control strategy [6]. This happens indeed to be the environment we have at our disposal for our experimental study (Section 7). Hence, in the cost model, we do not consider the compensation cost.

We use the following assumptions to further simplify the models we develop: (1) Assume all data sources are identical in terms of the cost of answering similar maintenance queries. Thus, we use R to represent each data source R_i ($1 \leqslant i \leqslant n$). (2) Assume all ΔR_i ($1 \leqslant i \leqslant n$) are identical in terms of the cost of evaluating them against a data source R, i.e., all ΔR_i have same number of insert and delete tuples involved. Thus, we can simplify our expressions by using the symbol $\mathscr{D}$ to represent each delta ΔR_i.

To represent the result delta of a maintenance query composed from a source delta $\mathscr{D}$, we define $\mathscr{D}_{i+1} = \mathscr{D}_i \bowtie R$ ($1 \leqslant i \leqslant n-1$) with $\mathscr{D}_1 = \mathscr{D}$. For simplicity, we use S_i to represent the size of a delta $\mathscr{D}_i$.

The cost of the batch maintenance is given by T_b with $T_b = n\sum_{i=1}^{n-1} [f_{net}(S_i) + f_{join}(S_i) + f_{net}(S_{i+1})]$, which is a summation of individual maintenance query costs. Here f_{net} and f_{join} represent the unit cost functions of data transfer and maintenance query answering, respectively. Here, $f_{net}(S_i)$ represents the network cost of sending $\mathscr{D}_i$ from the view manager to the data source. $f_{net}(S_{i+1})$ denotes the network cost of transferring the corresponding query result from the data source to the view manager. $f_{join}(S_i)$ denotes the join cost of evaluating the corresponding maintenance query.

The cost of adjacent grouping can be estimated by T_a assuming that we divide the maintenance steps evenly into groups of size m where $m<n$. Thus, n maintenance steps are divided into n/m groups with each having m maintenance steps. In each group, $m\sum_{i=1}^{m-1}[f_{net}(S_i)+f_{join}(S_i)+f_{net}(S_{i+1})]$ represents the cost of grouping and processing m source deltas (a $m\times m$ matrix along the main diagonal in Eq. (4)), while $\sum_{i=m}^{n-1}[f_{net}(mS_i)+f_{join}(mS_i)+f_{net}(mS_{i+1})]$ denotes the cost of processing the result of the above $m\times m$ matrix on the remaining $n-m$ data sources.

$$T_a=\frac{n}{m}\left\{m\sum_{i=1}^{m-1}[f_{net}(S_i)+f_{join}(S_i)+f_{net}(S_{i+1})]\right.\\ \left.+\sum_{i=m}^{n-1}[f_{net}(mS_i)+f_{join}(mS_i)+f_{net}(mS_{i+1})]\right\}.$$

The cost of conditional grouping is given in T_c. Corresponding to the two phase operations as described in Section 4, T_c is composed of scroll up and scroll down costs. $\sum_{i=1}^{n-1}[f_{net}(\sum_{j=1}^{i}S_j)+f_{join}(\sum_{j=1}^{i}S_j)+f_{net}(\sum_{j=2}^{i+1}S_j)]$ denotes the scroll up phase cost, which simply sums up the cost of each maintenance query. While $\sum_{i=1}^{n-1}[f_{net}(iS_i)+f_{join}(iS_i)+f_{net}(iS_{i+1})]$ denotes the scroll down phase cost. It is also a simple summation of queries in the scroll down phase.

$$T_c=\sum_{i=1}^{n-1}\left[f_{net}\left(\sum_{j=1}^{i}S_j\right)\right.\\ \left.+f_{join}\left(\sum_{j=1}^{i}S_j\right)+f_{net}\left(\sum_{j=2}^{i+1}S_j\right)\right]\\ +\sum_{i=1}^{n-1}\left[f_{net}(iS_i)+f_{join}(iS_i)+f_{net}(iS_{i+1})\right].$$

The above formulae show the basic relationship between the number of maintenance queries and the complexity (size) of each query as expected.

To highlight the key factors in this trade-off, we now further simplify the above cost functions. Note that in a local network environment, the unit cost (f_{net}) is rather small. We thus can simplify the cost functions by removing the network cost factors. To further accentuate this difference, we use S to represent each S_i (assume the size of each delta $\mathscr{D}_i$ is the same, $1\leqslant i\leqslant n-1$). Given these two assumptions, the cost expressions T_b, T_a and T_c can be simplified as shown in Fig. 8. The relationship among our strategies regarding the key cost factors is also described in Fig. 8. Here the x-axis represents the number of required maintenance queries, while the y-axis denotes the average delta size in each maintenance query. Note that for the adjacent grouping approach, we let $m=\sqrt{n}$ since it is shown to minimize the number of maintenance queries in this approach. As can be seen, if the query answering cost for a large delta is less than that of the sum of the costs of handling multiple smaller deltas, performance improvements are expected by reducing the number of maintenance queries.

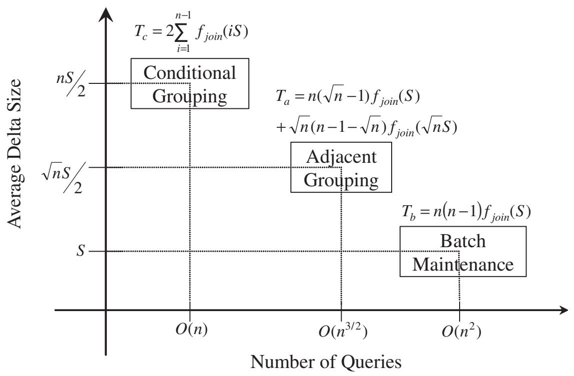


Fig. 8. Relationship of average delta size to the number of queries to sources.

7. Experimental evaluation

7.1. Experimental testbed

We have implemented the proposed strategies based on the TxnWrap system [6]. TxnWrap is a multiversion-based view maintenance system which removes concurrency control concerns from its maintenance logic. Thus, it is not necessary to apply compensation for handling concurrent source updates in our setting. The basic TxnWrap system maintains one single source update at a time using the known SWEEP algorithm [1]. The batch TxnWrap [13] combines the updates from the same data source and maintains the view extent using the source-specific deltas.

We have conducted our experiments on four Pentium III 500 MHz PCs connected via a local network. Each PC has 512 M memory with Windows

2000 and Oracle 8*i* installed. We employ six data sources with one relation each over three PCs (two data sources per PC). Each relation has 1,000,000 (1 M) tuples with 64 bytes on average of each tuple size. A materialized join view is defined through equi-joins upon these six source relations residing on a separate (the fourth) machine. The view has 1 M tuples with each tuple having 384 bytes on average (having the attributes of the source relations included). All the source deltas are composed of approximately the same number of insert and delete tuples. Note that two actual queries are needed when a single delta contains both insert and delete tuples.

7.2. Composing maintenance queries

Two ways of composing a maintenance query from a delta can be distinguished based on source-dependent properties, namely, whether the source is *cooperative* or *non-cooperative*. A non-cooperative source only answers maintenance queries (SQL queries), but offers no other services or control to the view manager. A cooperative data source would cooperate with the view manager by allowing to synchronize processes or to lock its data. To compose an appropriate maintenance query from a delta submitted to a non-cooperative data source (i.e., evaluating $\Delta R_i \bowtie R_j$), we have to use a composite SQL query which unions maintenance queries for a single source update to evaluate the result. A cooperative source would allow the view manager to build a temporary table directly at the data source, ship the delta data, evaluate it locally and send the result back.

We now experimentally compare batch maintenance costs using these two methods against sequential maintenance. In Figs. 9–11, we let the number of data updates vary from 10 to 100 (and then from 500 to 3000) with all updates from the same data source (on *x*-axis). The *y*-axis represents the total maintenance query processing time.

From Fig. 9, the processing time using a composite query increases slowly. For the temporary table approach, the increase of the total cost is even smaller compared to using a composite query. This is due to the fact that the setup cost (create temporary table and populate its extent) dominates the actual maintenance query expenses for small cases. This also explains that with a small number of updates, the temporary table approach is more expensive than the composite query-based approach. The sequential maintenance processing time increases linearly as expected.

Fig. 10 displays the ratio of the sequential processing time divided by batch processing using the data obtained from Fig. 9. The higher the ratio, the larger the performance improvement. We observe that the improvement of the composite query approach does no longer increase when the number of updates is larger than 50 in our current setting. While for batch maintenance using temporary tables, the ratio increases steadily.

In Fig. 11, we see that the cost of batch maintenance using the composite query approach increases when the number of updates increases. This

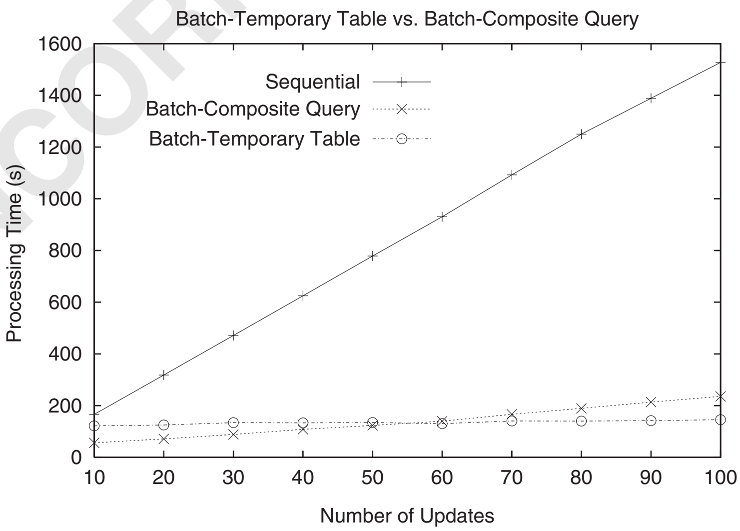


Fig. 9. Batching a small number of updates.

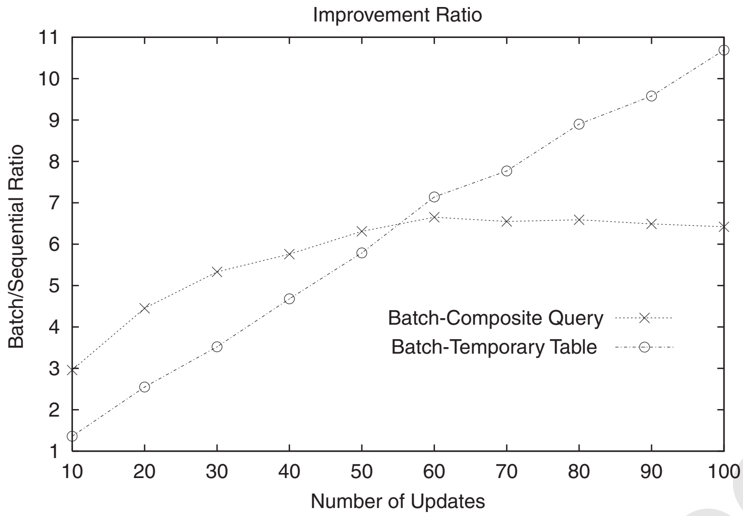


Fig. 10. Performance ratio of sequential-cost divided by batch-cost.

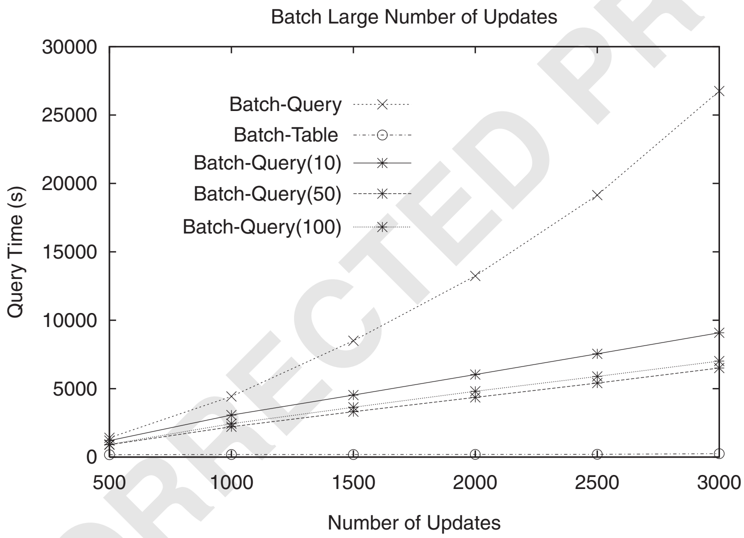


Fig. 11. Batch a large number of updates.

is because a composite query composed of the union of a large number of queries will result in a huge cost increase. Thus we instead suggest to divide such a large number of updates into smaller sub-batches of queries of size k based on the ratio measured in Fig. 10. The cost of the sum of these subqueries will be smaller than the cost of this one large composite query. As seen in Fig. 11, when we choose k equal to 50, the total maintenance cost using a composite query approach will reach its optimum in our experiment. However, if we use the temporary table approach, the total cost is even much lower than that of the optimized composite query approach. This is because the ratio of the increase of each such batch maintenance query to the increase in the number of source updates is very low.

To summarize, the cost of sequential maintenance is linear in the number of source updates. The batch maintenance has a fixed number of maintenance queries $O(n^2)$ with n being the number of data sources. However, the performance of answering the batching maintenance queries depends on the methods for composing maintenance queries from multiple source updates. For the temporary table approach, the cost does not increase too much as the number of source updates increases in each delta. While the batch-query approach does increase non-linearly as the number of source updates increases (Fig. 11).

Thus, we expect another crosspoint when comparing the batch-query approach with the sequential approaches for large numbers of source updates. While for the temporary table approach, we still expect the batch maintenance approach to be much more efficient. The reason is that answering a join query with a delta containing 1,000,000 tuples may not be 1000 times higher than answering a maintenance query containing 1000 tuples. Without loss of generality, from now on we utilize this more efficient temporary table approach to compose maintenance queries from deltas when comparing our proposed strategies.

7.3. Changing the number of source updates

Fig. 12 shows the average maintenance time (on the *y*-axis) for different maintenance approaches by varying the number of source updates from 100 to 1000 (on the *x*-axis). These updates are evenly distributed among six data sources. That is, for the k updates, each source delta experiences approximately $k/6$ updates. From Fig. 12, the maintenance cost of all these strategies increases very slowly because we compose and issue maintenance queries using the temporary table approach. Seen from Fig. 12, the batch processing is almost four times slower than the conditional grouping. We also see the following maintenance cost relationship: *conditional grouping* $<$ *adjacent grouping* $<$ *batch processing*. Thus, with a smaller number of maintenance queries, we do have less processing time even when the complexity (size) of each maintenance query increases. Given that the adjacent grouping is a medium performer between the batch and conditional grouping, we will focus on comparing batch with conditional grouping in more depth below.

Fig. 13 shows the performance changes of batch and conditional grouping given an increasing number of source updates. The maintenance cost of both approaches increases steadily as the size of each delta increases. The conditional grouping still outperforms batch maintenance due to the size of the delta not being a major factor on the Oracle query cost if we use the temporary table approach and the conditional grouping has a smaller number of maintenance queries.

7.4. Impact of the join ratio

We set up 200 updates on six sources (each source delta change experience about 30 updates) and vary the join ratio from 0.5 to 3.0 (on *x*-axis). Join ratio here represents the average number of tuples affected by a source change. For example, a join ratio equals to 2 means that a single update which changes a tuple in the source may cause 2^5 tuples to be updated in the view extent given the view is defined over six sources. From Fig. 14, we see that the higher the join ratio, the higher both maintenance costs. A high join ratio increases the size of each temporary maintenance

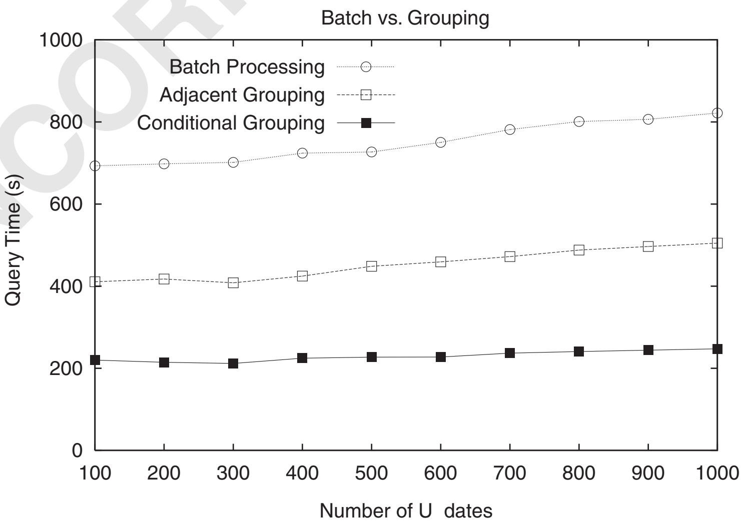


Fig. 12. Group a small number of source updates.

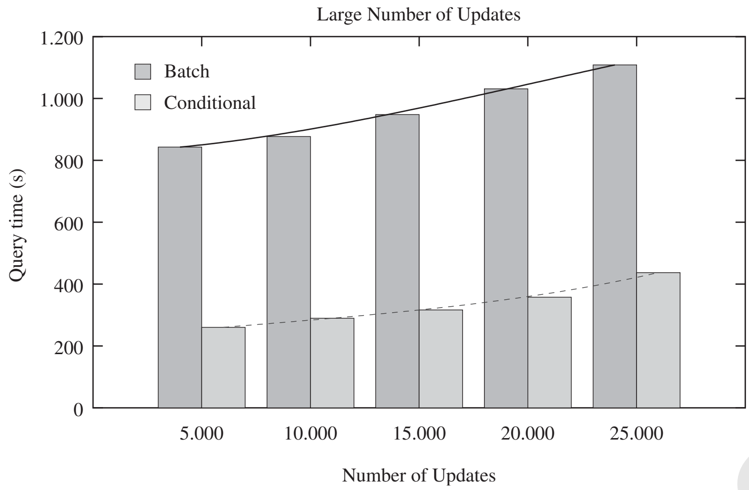


Fig. 13. Group a large number of source updates.

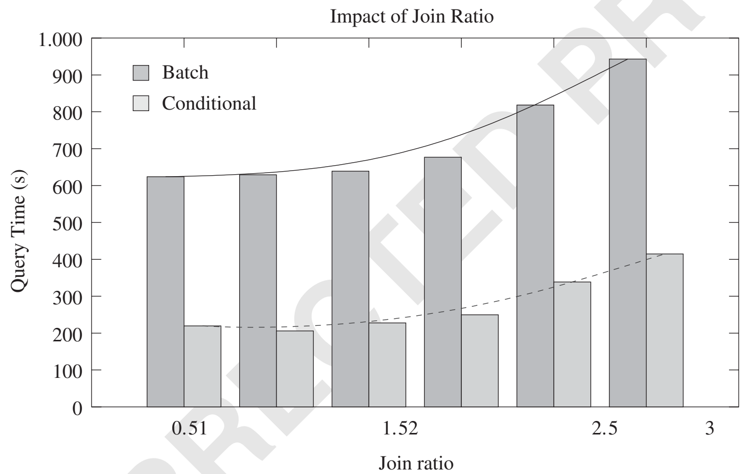


Fig. 14. Change the join ratio in the view.

result, which in turn increases the time to answer the maintenance query. In this experiment, the rates, defined as the batch maintenance cost divided by the grouping maintenance cost, are 3.06, 2.81, 2.71, 2.42, and 2.27 for join ratios 1, 1.5, 2, 2.5, and 3, respectively. Thus, the higher the join ratio, the closer these two maintenance costs become. This is because any change in the temporary result size will be amplified by the join ratio and the conditional grouping has extra data (null values) that needs to be processed in the scroll up phase. Thus, the benefit of having a smaller number of maintenance queries will be slowly overtaken by the increase of each query cost.

7.5. Changing the distribution of source updates

We examine the impact of the distribution of 1000 updates among the data sources on the maintenance performance (Fig. 15). On the *x*-axis, a distribution of 1 denotes that we only have one source delta with 1000 updates, while k ($2 < k \leqslant 6$) indicates that we consider k source deltas with each delta change having around $1000/k$ updates. Fig. 15 presents the cost ratio (batch maintenance cost divided by conditional grouping cost). Clearly, the more data sources are involved, the higher the performance improvement. This is because the total number of maintenance queries in the batch maintenance

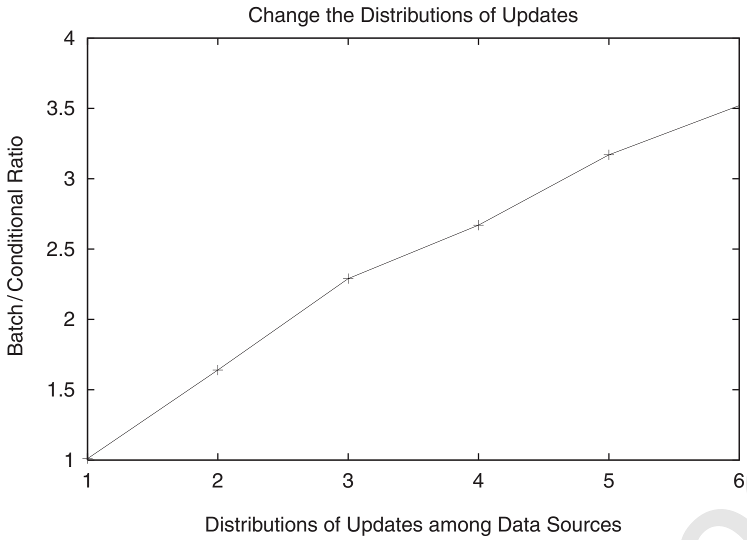


Fig. 15. Change the distributions of updates.

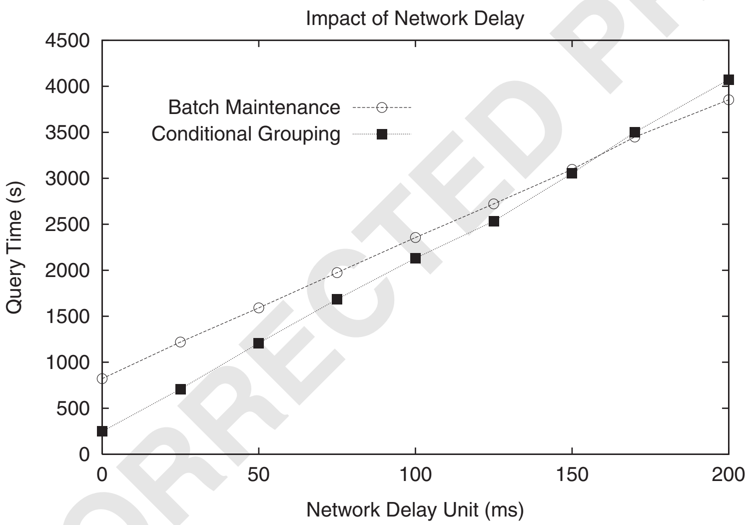


Fig. 16. The impact of network delay.

changes from 5 to 30 queries if we increase the distribution from 1 to 6 sources, while the conditional grouping only changes from 5 to 10 correspondingly. Thus more improvement is achieved by further reducing the number of maintenance queries.

7.6. *Impact of the network delay*

To evaluate the impact of different data transfer rates of the network, we insert delay factors to model the data shipping costs. The delay is generated based on the average time to transfer one tuple. For example, if we assume that the average time of transferring a tuple with 64 bytes is ℓ, then it takes $100 * 2 * \ell$ to transfer one delta with 100 tuples with 128 bytes each. We set up six source delta changes with about 180 updates each (a total of 1000 data updates) and we let ℓ vary from 0 to 200 ms. In Fig. 16, both maintenance costs grow steadily as the network cost of each maintenance query is increasing. In a typical network environment where the transfer time of one tuple with 64 bytes is less than 100 ms, conditional grouping is more efficient than the batch method because we have a smaller number of maintenance queries. However, in a slow network, i.e., when the average transfer time for one tuple is larger than 200 ms, then the gain achieved by reducing the number of maintenance queries is overtaken by the increase in the network cost of each query. This is caused by having some extra data (null values) to be transferred in the conditional grouping. This extra data becomes a burden in a slow network.

8. Related work

Maintaining materialized views under source updates is one important issue in many applications such as information integration and data warehousing [10]. Early work has studied incremental view maintenance assuming no concurrency [7,20]. In approaches that need to send maintenance queries to the data sources, especially in a environment with autonomous data sources, concurrency problems can arise. Maintenance strategies such as [1,5,6,10,11] have focused on handling anomaly problems due to concurrent updates among data sources.

From both a resource and performance perspective, incrementally maintaining batches of updates is of particular interest. That is, changes to the sources can be buffered and propagated periodically to maintain the view extent. Refs. [3,7,12,13,21,22] propose algorithms to maintain materialized views incrementally using source-based batching. Salem et al. [3] proposed an asynchronous view maintenance algorithm using delta changes of data sources. Labio et al. [12] proposed a batch maintenance algorithm which can be applied to maintain a set of views. In our previous work [13], we have proposed a batch view maintenance strategy that works even when both data and schema changes may happen on data sources. However, all these existing approaches are only concerned with batching updates from the same source. Recent work [23] proposes an efficient maintenance strategy that exploits the asymmetry among different components of the maintenance cost. Lee et al. [19] introduce a delta propagation strategy that also reduces the number of maintenance queries to data sources. It is close to our proposed adjacent grouping approach. However, none of the above have considered how to group heterogeneous deltas to further reduce the number of maintenance queries—as undertaken by our work.

Posse [24] introduced a view maintenance optimization framework. This work focuses on the order in which these source deltas are to be installed (to be maintained). While in our work here, we explore the optimizations at a lower level. That is, given delta changes, we study how to compose maintenance queries to data sources to calculate the maintenance results more efficiently.

Distributed query optimization [16,25,26] focuses on query optimization in a distributed environment. It provides algorithms for join ordering and for allocating query operators to resources. This is orthogonal to what we have explored here since our work focuses on how to reduce the number of maintenance queries given the join ordering has been decided. Making use of shared common expressions has been well studied in multiple query optimization [18,27]. As we have discussed, such common expressions are usually too large to be evaluated in a maintenance plan. For example, the common subexpression such as $R_3 \bowtie R_4 \bowtie \cdots \bowtie R_n$ for the first two maintenance steps in Fig. 4(a) is too expensive to evaluate. This is because each data source may be huge compared to the deltas. Instead, we identify the common sources and share the access to them. In our context, such common sources could possibly even be manually identified because the maintenance queries are relatively fixed given a view definition.

9. Conclusion

In this paper, we have taken a fresh new look at how to restructure a batch view maintenance plan to optimize the view maintenance performance when maintaining a large batch of source updates. This optimization is achieved by dramatically reducing the number of maintenance queries to remote data sources. A series of novel grouping maintenance strategies have been proposed and implemented in a working prototype. Our experimental studies illustrate that maintenance performance can be significantly improved by having a smaller number of maintenance queries. In particular, our conditional grouping strategy is almost four times faster compared with the typical batch maintenance in a majority of the cases.

As a next step, we are investigating how to combine the distributed query processing techniques with grouping strategies we have described in this paper to further optimize the maintenance performance. For example, we explore how to choose the best linear path in the join graph for grouping maintenance among many possible linear paths, and how to efficiently maintain complex (i.e., cyclic) join views.

References

[1] D. Agrawal, A.E. Abbadi, A. Singh, T. Yurek, Efficient view maintenance at data warehouses, in: Proceedings of SIGMOD, 1997, pp. 417–427.

[2] A. Gupta, I. Mumick, Maintenance of materialized views: problems, techniques, and applications, IEEE Data Eng. Bull. 18 (2) (1995) 3–19.

[3] K. Salem, K.S. Beyer, R. Cochrane, B.G. Lindsay, How to roll a join: asynchronous incremental view maintenance, in: SIGMOD, 2000, pp. 129–140.

[4] J.A. Blakeley, P.-A. Larson, F.W. Tompa, Efficiently updating materialized views, in: Proceedings of SIGMOD, May 1986, pp. 61–71.

[5] J. Chen, S. Chen, E.A. Rundensteiner, A transactional model for data warehouse maintenance, in: ER'02, September 2002, pp. 247–262.

[6] S. Chen, B. Liu, E.A. Rundensteiner, Multiversion based view maintenance over distributed data sources, ACM Trans. Database Syst. (TODS) 29 (4) (2004) 675–709.

[7] L.S. Colby, T. Griffin, L. Libkin, I.S. Mumick, H. Trickey, Algorithms for deferred view maintenance, in: Proceedings of SIGMOD, 1996, pp. 469–480.

[8] A. Gupta, I.S. Mumick, V.S. Subrahmanian, Maintaining views incrementally, in: Proceedings of SIGMOD, 1993, pp. 157–166.

[9] X. Zhang, E.A. Rundensteiner, L. Ding, Parallel multi-source view maintenance, VLDB J. 13 (1) (2004) 22–48.

[10] Y. Zhuge, H. García-Molina, J. Hammer, J. Widom, View maintenance in a warehousing environment, in: Proceedings of SIGMOD, May 1995, pp. 316–327.

[11] Y. Zhuge, H. García-Molina, J.L. Wiener, The strobe algorithms for multi-source warehouse consistency, in: Parallel and Distributed Information Systems, 1996, pp. 146–157.

[12] W.J. Labio, R. Yerneni, H. García-Molina, Shrinking the warehouse updated window, in: Proceedings of SIGMOD, June 1999, pp. 383–395.

[13] B. Liu, S. Chen, E.A. Rundensteiner, Batch data warehouse maintenance in dynamic environments, in: CIKM'02, November 2002, pp. 68–75.

[14] B. Liu, E.A. Rundensteiner, D. Finkel, Restructuring batch view maintenance efficiently, in: CIKM'04, Poster, November 2004, pp. 228–229.

[15] J. Chen, X. Zhang, S. Chen, K. Andreas, E.A. Rundensteiner, DyDa: data warehouse maintenance under fully concurrent environments, in: Proceedings of SIGMOD Demo Session, 2001, p. 619.

[16] D. Kossmann, The state of the art in distributed query processing, ACM Comput. Surv. (CSUR) 32 (4) (2000) 422–469.

[17] B. Liu, E.A. Rundensteiner, Cost-driven view maintenances in distributed environments, Technical Report WPI-CS-TR-03-30, WPI, 2003.

[18] T.K. Sellis, Multiple-query optimization, ACM Trans. Database Syst. (TODS) 13 (1) (1988) 23–52.

[19] K.Y. Lee, J.H. Son, M.H. Kim, Efficient incremental view maintenance in data warehouses, in: CIKM'01, November 2001, pp. 349–356.

[20] J.J. Lu, G. Moerkotte, J. Schue, V.S. Subrahmanian, Efficient maintenance of materialized mediated views, in: SIGMOD, 1995, pp. 340–351.

[21] I. Mumick, D. Quass, B. Mumick, Maintenance of data cubes and summary tables in a warehouse, in: Proceedings of SIGMOD, May 1997, pp. 100–111.

[22] D. Quass, J. Widom, On-line warehouse view maintenance, in: Proceedings of SIGMOD, 1997, pp. 393–400.

[23] H. He, J. Xie, J. Yang, H. Yu, Asymmetric batch incremental view maintenance, in: Proceedings of ICDE, 2005, pp. 106–117.

[24] K. O'Gorman, D. Agrawal, A.E. Abbadi, Posse: a framework for optimizing incremental view maintenance at data warehouse, in: Data Warehousing and Knowledge Discovery, 1999, pp. 106–115.

[25] L.M. Haas, D. Kossmann, E.L. Wimmers, J. Yang, Optimizing queries across diverse data sources, in: Proceedings of VLDB, 1997, pp. 276–285.

[26] M. Stonebraker, P. Aoki, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, A. Yu, Mariposa: a wide-area distributed database system, VLDB J. 5 (1) (1996) 48–63.

[27] P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe, Efficient and extensible algorithms for multi query optimization, in: Proceedings of SIGMOD, 2000, pp. 249–260.