Posse: A Framework for Optimizing Incremental View Maintainance at Data Warehouses*

K. O'Gorman D. Agrawal A. El Abbadi
Department of Computer Science
University of California
Santa Barbara, CA 93106
{kogorman,agrawal,amr}@cs.ucsb.edu

June 11, 1999

Abstract

Incremental maintenance of materialized views consists of installing changes to the view as a result of an update to one of the base relations. This generally requires information from some or all of the other base relations comprising the view, which may be obtained by probing (i.e. querying) the base relations. In a distributed context such as a data warehouse, there may be a great deal of concurrency among updates as well as probes and replies, leading to consistency problems, for which a number of solutions have been offered. These solutions, however, have been expressed as algorithms of limited practical use because they are variously limited to restricted view queries, relational semantics, and limited concurrency. We propose the Posse[1] framework in which many of these limitations are removed, and in which the techniques of distributed query optimization can be brought to bear on the problem. To this end, we show how for a particular method of consistent probing it is possible to have the power of SQL view queries with multiset semantics, and at the same time have available a spectrum of concurrency from none at all as in some previously proposed solutions to the maximum concurrency obtained by issuing all probes in parallel. We then show how optimization of the probing process can be used to select various degrees of concurrency for the desired tradeoffs of concurrency against processing cost and message size.

1 Introduction

Data warehousing is increasingly used to collect information from disparate and possibly heterogeneous data sources and provide a platform for quick response to queries. For efficient query processing, the views are commonly materialized at the data warehouse, giving rise to the need to maintain the materialized data in order to keep it up-to-date with respect to the data sources. This maintenance can either be performed by complete re-computation of the views or by *incremental maintenance* techniques [GMS93, GL95, HZ96] propagating changes to the views.

Figure 1 illustrates a warehouse with multiple data sources. The data warehouse stores materialized views which are expressed as SQL queries over data sources. Each data source locally responds to user

*This work is partially supported by the NSF under grant numbers CCR-9712108 and EIA-9818320

[1]In the USA at least, a *posse* is a group of citizens temporarily deputized into the police force, for a (more or less loosely coordinated) search.

queries and updates. Updates are propagated as update messages to the warehouse and must be integrated into the materialized view in order to keep it up-to-date, but this poses several problems.

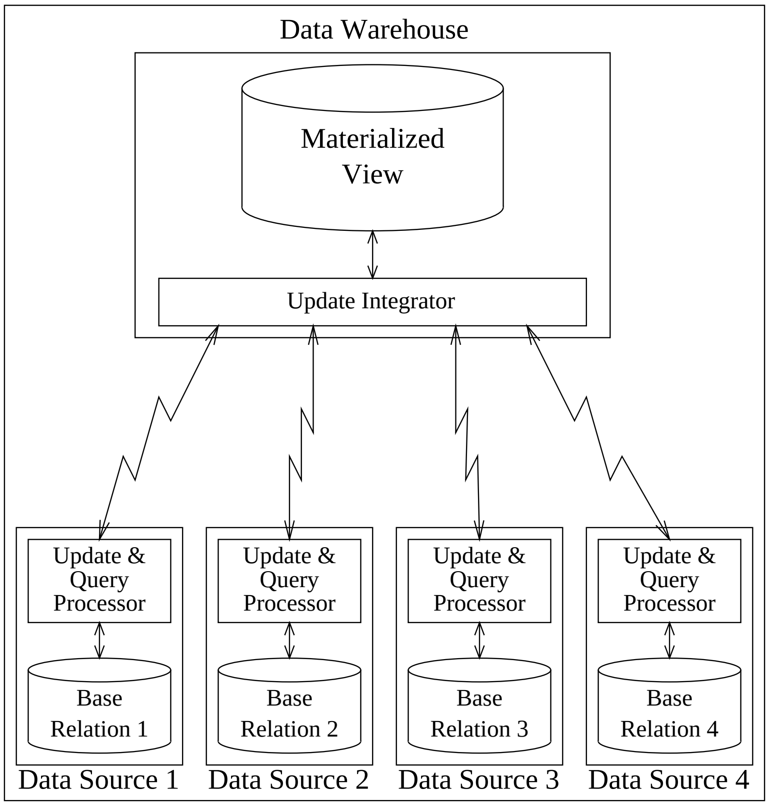

Figure 1: A Warehouse Architecture

First, the update message contains information only about changes at the data source where the update occurred, but the corresponding change to the view may depend on information at other data sources. Accordingly, the data warehouse must *probe* (i.e. query) the other data sources where the additional information resides, and must delay installation of the change to the materialized view until all the necessary information arrives. This is in effect a distributed query, which gives rise to several optimization issues regarding the sequence in which to send the probes, and the data to be requested in each probe.

Second, because of the delays introduced in probing the data sources, it can happen that updates occur concurrently at the data sources being probed, and consequently the replies to the probes may reflect updates that occur in some sense “after” the update which is being processed. Without some sort of compensation, the effects of these *interfering* updates could result in the materialized view becoming inconsistent with respect to the data sources.

We regard existing algorithms as excessively limited in several ways. They use query languages of limited expressive power, corresponding to select-project-join queries. Also, they occupy an extreme in the spectrum of possible degrees of concurrency, defined by sequential probing. We present a framework in which either end of the spectrum can be achieved while ensuring the properties of correctness and consistency. We then show how intermediate positions in the spectrum of solutions can be achieved, and how these positions represent opportunities for improvement by trading off concurrency for communications costs. It should be noted that the ends of the spectrum may be useful themselves in certain environments. Serial processing may achieve low message sizes and processing costs at the expense of relatively high response times. The other end of the spectrum involves parallel probing of all sources and hence complete concurrency, which may achieve short response times at the cost of relatively large message sizes and processing costs. The objectives of optimization and the tradeoffs involved are different depending on the particular system, its goals and its uses, and we show transformations for illustrative intermediate points between the extremes.

The rest of this paper is organized as follows: Section 2 summarizes related work. Section 3 presents the definitions and assumptions for the rest of the paper. Section 4 introduces a motivational example that will be used in the rest of the paper. Section 5 describes the theoretical and conceptual environment for addressing issues of consistency and correctness. Section 6 presents the main result of this paper, the framework in which optimizations and consistency can coexist. Section 7 concludes the paper.

2 Related Work

Our work anticipates the use of joins (and possibly semi-joins) to reduce the size of reply messages. Berenstein and Chiu [BC81] demonstrate bad worst-case behavior of cyclic queries, in that semijoins cannot be guaranteed to reduce message sizes, even for join queries whose final result is empty. Kambayashi et al [KYY82] describe a generalized semijoin where joining attributes could come from several relations, and develop procedures for processing cyclic queries with these generalized semijoins.

The properties of self-maintainable SPJ views were explored by Huyn [Huy96], although self-joins are disallowed. The possibility of adding views so that the ensemble would be self-maintainable as a unit was explored by Quass et al [QGMW96]. We do not explore the issue of self-maintainable queries *per se* since it overly restricts the space of allowable view queries. However, we use a closely related class of queries as our probes.

Maintenance of views in the presence of aggregate operations was described by Quass [Qua96]. Gupta et al [GMS93] described incremental maintenance of views colocated with their data sources, and considered very general view definitions as well as both multiset and relational semantics. Hull and Zhou [HZ96] described a framework for maintaining relational views at a data warehouse that were materialized, virtual or a hybrid. In addition, they developed *View Decomposition Plans* with formal descriptions, which in the terms of the present paper support both user queries and probes. The effects of multiset semantics were addressed in [GL95], but in the context of views colocated with the source data.

Several existing algorithms for incremental maintenance of materialized views in a data warehouse [ZGMHW95, ZGMW96, AESY97] have focused primarily on the problem of compensation for interfering updates. They have done so with a variety of restrictions on the architecture, the views, and the algorithms, which make the solutions impractical. The ECA algorithm of [ZGMHW95] considers only a single data source and requires a period of quiescence for the materialized view to become consistent with the base relations. The basic Strobe algorithm of [ZGMW96] required the materialized view to contain a key for each of the base relations, and also requires quiescence before installing any changes in the materialized view. A variation called C-Strobe avoids waiting for quiescence but at a potentially very high communication cost (factorial in the number of data sources), making this solution not scale well. These difficulties arise from the method of compensating the results of the entire query. The SWEEP algorithms of [AESY97] introduce a method of compensating each probe separately that does not require quiescence and which scales linearly. Like the Strobe algorithm, SWEEP ignores some of the potential parallelism in the problem and limits itself to one outstanding probe at a time regardless of the number of unprocessed update messages. The PSWEEP algorithm [ZR99] extends SWEEP by introducing parallel evaluation of updates, but is otherwise similar to SWEEP.

The ECA, STROBE, SWEEP and PSWEEP algorithms limit themselves to select-project-join (SPJ) view queries and relational (i.e. duplicate-free) semantics, whereas warehouse views typically use aggregation and group-by queries to reduce query processing times, and multiset (bag) semantics.

Improvements not contemplated here are possible, and we believe that they could be integrated without excessive difficulty. As early as 1988, Sellis [Sel88] explored the optimization of multiple queries over the same data sources, and Park and Segev [PS88] explored optimizations involving common subexpressions. More recently, Colby and Mumick [CM96] have explored efficient maintenance of multiple views at a data warehouse, with attention to the consistency of the views. General optimization of distributed queries is discussed in many places; one text with a reasonable survey is [ÖV91].

3 Definitions and Assumptions

The data sources report to the data warehouse all updates to their local relations, and the data sources are capable of processing and responding to SQL queries relating to their contents. The data warehouse defines one or more materialized views expressible in SQL, including GROUP BY and HAVING clauses, and aggregate functions. Queries from the data warehouse take the form of SQL queries. We refer to such a query as a *probe*. Probes may refer to attributes of the target base relation, attributes of the update message being processed, and attributes of the reply to other probes issued in processing the same update message and whose replies have been received at the data warehouse. The source of a message can always be determined, and even in the presence of multiple unanswered queries, the reply to a probe can be matched at the data warehouse with the probe that gave rise to the reply. We refer to a probe and its reply as *belonging to* the update or to the update message that gave rise to the probe. Where it is not ambiguous, we use a name of a relation as found in the view query to label a data source and updates originating from that source.

Communication is via reliable point-to-point FIFO channels; this property, which is commonly assumed in data warehousing, e.g. [ZGMW96], is used in our model of consistency and in compensation of interfering updates. Data sources are independent in that they do not synchronize or communicate with each other, and updates to the data sources are mutually independent. Data sources process updates and queries atomically, queries are processed in the order of their arrival, and messages relating to these events are transmitted in the same order as the processing of the events.

An update message takes the form of a *delta table* which is a relation whose tuples are augmented with a signed integer cardinality. The cardinality is not an attribute of the schema, but encodes both the number of duplicates of a tuple for multiset semantics and the insertion (positive) or deletion (negative) sense of an update message or probe reply. In our algorithms, the ongoing computation of the changes to be applied to a view are accumulated in a delta table. We use the terms multiset and relation interchangeably in this paper, and regard a multiset as having the form of a delta table with only positive cardinalities, although we omit the cardinality from multiset examples when the cardinalities are all $+1$.

SQL does not have the notion of negative cardinality; nevertheless we refer to delta tables in contexts in which SQL does not permit them. We reason that in an actual implementation the restrictions of SQL can be accomodated by splitting operations into a positive and a negative suboperation, perhaps at the cost of doubling the number of messages.

We extend the operations of relational algebra to apply to delta tables in the natural way, consistent with their interpretation as multisets. The operations remain as in relational algebra except for the manipulation of the cardinalities. A tuple in the cross product of two delta tables has a cardinality which is the arithmetic product of the cardinalities of the two tuples which gave rise to the result tuple. Projection initially retains the cardinality of each tuple, but like all other operations ends by combining all tuples with the same attributes, with each result tuple having a cardinality which is the arithmetic sum of the cardinalities of the tuples that were thus combined. Recalling that cardinalities may have negative sense, a resulting tuple

Enrollment

Student	Course
Kevin	CS201
Mary	CS103
Mike	CS101
Sharon	CS102

Teaches

Course	Prof
CS101	Jones
CS102	Smith
CS103	Smith
CS201	Jones

Assists

Student	Course
Kevin	CS101
Mary	CS102

Figure 2: The Example Base Relations

with a cardinality of zero is discarded. Selection operates on entire tuples, retaining the tuple and its cardinalities or discarding the tuple entirely. As usual, join is defined as a selection on a cross-product. Subtraction is performed by reversing the signs of the cardinalities of the second argument and adding the delta tables. Addition is performed by concatenating the argument delta tables and combining as described above. When applied to relations, addition corresponds to multiset union, but subtraction can result in negative cardinalities. Multiset subtraction (as opposed to delta table subtraction) is therefore performed by a deletion operation which ensures non-negative cardinalities. Normalization sets all cardinalities to +1; it is useful as a preliminary to a semijoin or in spawning a branch of a join tree.

A given relation may appear more than once in the FROM clause of a view query. In the general case we model this as multiple fictional "clones" of the given relation, whose data sources "just happen" to contain identical tuples, and emit identical update messages that are delivered to the data warehouse in consecutive order. This fiction can be maintained by a processing layer between the communications channel and the algorithms we present here. This fiction can also be penetrated in order to apply optimizations that reduce the message counts or processing costs of view maintenance. All of these things will be seen to occur in our example.

4 Motivation

We use an example to demonstrate the operation of a data warehouse. This example comprises a data warehouse with a single view defined over three data sources.

4.1 Schema and View

Consider an entity-relationship schema describing a portion of university enrollment and teaching assistant appointments comprising the three relations in Figure 2, and a data warehouse defining a single view over these relations:

```
CREATE VIEW DoubleStatus(Prof,Conflicts)
  AS SELECT T1.Prof,COUNT(E.Student)
    FROM Assists A, Enrollment E, Teaches T1, Teaches T2
    WHERE T1.Prof = T2.Prof AND T1.Course = A.Course
      AND T2.Course = E.Course AND E.Student = A.Student
    GROUP BY T1.Prof
```

The corresponding state of the view would be DoubleStatus in Figure 3A, reflecting the fact that Jones and Smith each have one student (resp. Kevin and Mary) who is simultaneously an assistant and a student.

The DoubleStatus view is an interesting example of a query that can be used to expose multiple relationships. Parenthetically, we note that depending on the subject area of the situations being modeled, DoubleStatus

Prof	Conflicts
Jones	1
Smith	1

A

DoubleStatus

Prof	Conflicts
Jones	1

B

DoubleStatus

Prof	Conflicts
Smith	1

C

Figure 3: The States of the View

update E (from Enrollment)

Student	Course	Cardinality
Mary	CS201	$+1$
Mary	CS103	-1

update T (from Teaches)

Course	Prof	Cardinality
CS201	Smith	$+1$
CS201	Jones	-1

reply(E,T) (from Teaches, for *update* E replying to probe given in the text)

Course	Prof	Student	Cardinality
CS103	Smith	Mary	-1
CS201	Smith	Mary	$+1$

Figure 4: The Messages

such situations characterize conflict-of-interest situations as well as situations where multiple commercial relationships offer competitive or negotiating possibilities.

From a strictly formal point of view this view query is interesting because the Teaches relation appears more than once in the `FROM` clause, the query contains the `COUNT` aggregate function, and because its joins form a cycle. Since this view query is not equivalent to any acyclic query, there are limits to the guarantees that can be made by some optimization techniques[BC81]; however, we demonstrate the usefulness of optimization for reasonable instances of the base relations.

4.2 Updates and Incremental View Maintenance

Now suppose that an update occurs at the Enrollment relation, replacing the tuple (Mary, CS103) with the tuple (Mary, CS201) and that the data source sends the update message *update* E of Figure 4, and that there is an update to the Teaches relation replacing the tuple (CS201, Jones) with the tuple (CS201, Smith) resulting in the update message *update* T of Figure 4. Let us also suppose that these update messages arrive at the data warehouse in the order given, before processing begins on either message.

The first update (*update* E) should change the contents of the DoubleStatus view so that Mary no longer has double status with regards to Smith, as in Figure 3B. However, the warehouse does not have sufficient information to compute this change to the view, because there are no attributes in common between the view and the tuples that have changed. Accordingly, the update messages have to be joined with information from other base relations. The view contains none of the attributes that appear in the first update, *update* E, so that it becomes necessary to obtain information from, for example, the Teaches relation. But since *update* T has already arrived from that relation, and is reflected in the contents of the Teaches relation, the current state of Teaches would be incorrect for computing the effect of *update* E by itself to produce Figure 3B. Some sort of compensation for the effects of the concurrent *update* T must be applied.

The second update (*update* T) has similar considerations, and should lead to the state DoubleStatus

$comp(E, T)$

Course	Prof	Student	Cardinality
CS201	Jones	Mary	-1
CS201	Smith	Mary	$+1$

$corrected(E, T)$

Course	Prof	Student	Cardinality
CS103	Smith	Mary	-1
CS201	Jones	Mary	$+1$

Figure 5: Compensating Delta Tables

in Figure 3C, reflecting the fact that the change of professor in course CS201 takes Kevin out of double relationship with Jones, but returns Mary to double relationship with Smith.

The ECA algorithm [ZGMHW95] cannot be used for our example view because ECA contemplates only a single data source. The STROBE algorithms [ZGMW96] cannot handle the view because they require the view to contain a key for each base relation. The SWEEP algorithms [AESY97] cannot support the view because of their limitation to select-project-join view queries.

4.3 Compensation for Interfering Updates

The difficulty mentioned above in arriving at the correct value of DoubleStatus after *update E* arises because the updates are concurrent (i.e. the second has occurred at its data source before the probe of its relation has arrived) which in turn is a natural consequence of independent data sources in a distributed system with communication delays.

Evidently to achieve consistency we have to somehow compensate for the concurrent update (*update T*). To do this we closely follow the local compensation method of [AESY97], which is outlined below.

Let us suppose, for example, that the update to the Enrollment relation (*update E*) has arrived at the data warehouse, and a probe has been sent to the Teaches relation. For example, let the probe simply call for a join with the update message being processed, by the SQL query $probe(E, T)$

SELECT * FROM Teaches NATURAL JOIN *update E*

The difficulty arises because the update to the Teaches relation (*update T*) occurs before the probe for *update E* arrives at Teaches, but is delivered at the data warehouse after *update E*. Accordingly, the order of delivery at the data warehouse is *update E*, *update T*, and finally $reply(E, T)$. In such a case the second update message is said to *interfere* with the probe, and this can be detected by the data warehouse by comparison of the delivery times or by another equivalent means. Our framework will perform the comparison by keeping the messages in a list ordered by delivery time.

Once $reply(E, T)$ has arrived at the data warehouse, and before it has been processed, the relevant part of the Message List contains the two update messages followed by the reply of Figure 4, that is *update E*, *update T* and $reply(E, T)$. Note that $reply(E, T)$ incorporates the effects of *update T* in showing Smith as the teacher of CS201, but that correct processing for *update E* requires using the state of the Teaches relation prior to *update T*. This requires compensation to construct the desired state of the reply from the Teaches relation prior to *update T*.

Compensation for interfering updates (in this case just *update T*) is calculated by applying the probe query to the interfering updates and reversing the effect of the interfering updates on the reply to the probe. Applying $probe(E, T)$ to *update T* (joining *update T* with *update E*) produces a compensating delta table $comp(E, T)$ of Figure 5, which represents the interfering contribution from *update T*. Subtracting *update T* from $reply(E, T)$ removes the effects of this interference and produces the compensated reply $corrected(E, T)$ of Figure 5. This correctly reflects the relevant contents of the Teaches relation prior to the interfering update, and this reply results in a change to the view.

5 The Environment

In this section we present the computational and conceptual environment for our framework for view maintenance. The correctness model is the conceptual standard for our definitions of consistency. The computational model comprises the machinery available to our algorithms. The Concurrent Posse algorithm operates in the computational model to define an extreme of concurrency at the cost of large message sizes, so that a spectrum of solutions can be realized between Concurrent Posse on the one hand and the existing algorithms [ZGMHW95, ZGMW96, AESY97] on the other.

5.1 The Correctness Model

Consider the system architecture illustrated in Figure 6. In addition to the materialized view, the data warehouse materializes copies or *mirrors* of all of the base relations. Updates to the base relations at the data sources are sent as update messages to the data warehouse. The processing of an update message at the data warehouse is a single atomic action which includes updating the materialized copy of the base relation, reevaluation of the view query, and installation of a new materialized view.

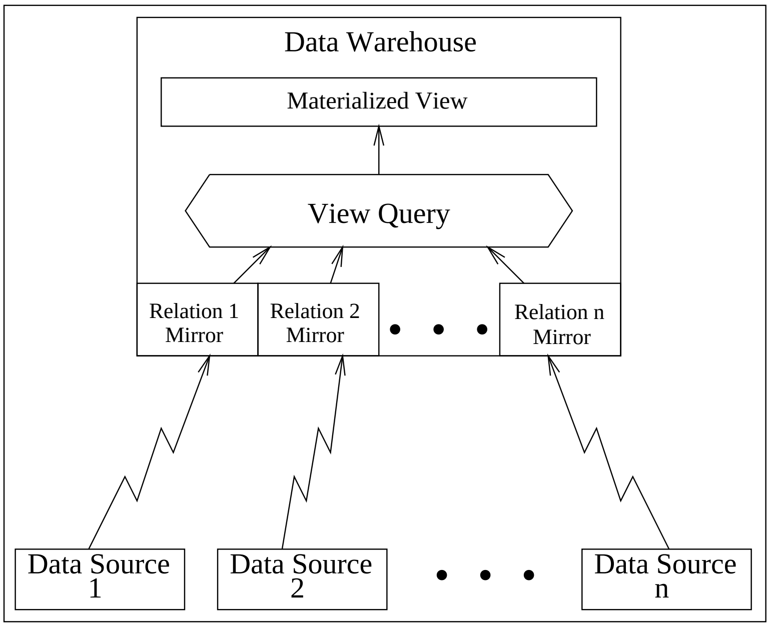

Figure 6: The Correctness Model

A number of properties can be defined to describe the degree of convergence or consistency of an algorithm with respect to the correctness model. Following the ideas of [ZGMW96], we define these representative properties:

complete consistency in which the algorithm takes the materialized view through the same states in the same order as the correctness model, namely an initial state plus one state for each update from a data source..

strong consistency in which the algorithm takes the materialized view through some subset of the states visited by the correctness model, and does so in the same order as the correctness model.

convergence in which the algorithm arrives at the same state as the correctness model, given a period of quiescence long enough to process all unprocessed updates.

We present algorithms within the Posse framework that achieve complete consistency with the correctness model, and mention variants which achieve strong consistency and convergence.

These definitions in effect define consistency in terms of the message delivery order at the data warehouse, in accordance with our focus on the algorithms at the data warehouse. Consistency with the states at the data sources is related to this according to the transaction support at the data sources and the properties of the communication links between the data sources and the data warehouse. In the presence of FIFO communications, these definitions are slightly stronger than those presented in [ZGMW96], which are stated as equivalence to some serial schedule of the data source updates; here the specific schedule is stated ([ZGMW96] defines the additional consistency criterion of *weak consistency* which does not arise in our framework.) As in the earlier algorithms it is the schedule corresponding to the sequence of delivery of update messages at the data warehouse.

5.2 The Computational Model

The Posse framework uses the computational model illustrated in Figure 7. In this execution environment, the data warehouse does not contain mirrors of the base relations. As a result, it is necessary for the data warehouse to send probes to the data sources to obtain the information to complete processing of each update message.

The incoming messages at the data warehouse, both update messages and replies to probes, are kept in a *Message List*, and each message remains in the list until it has been completely processed. In the case of a probe reply, this completion may be more or less immediate, but it is helpful for such replies to reside in the Message List in order to keep track of their sequence of arrival with respect to other messages, because this sequence identifies the updates that are interfering. In the case of update messages, completion occurs after all needed probes have been sent and replies have been processed, and when subsequently the update message is at the head of the Message List. At that time the computation of the appropriate change to the materialized view is complete, and the change is installed. The imposed sequence of installation of changes is thus the same sequence in which the update messages arrived, and can easily be seen to be the same order in which the same changes would be installed in the correctness model. The proof of correctness is omitted here, but follows directly from the order of installation of the updates and the correctness of the message compensation, developed in [AESY97].

The normal sequence of processing for an update message is as follows, referring to the numbers in Figure 7. When an update occurs at a data source, the data source sends (1) an update message to the warehouse, which puts the message (2) in the Message List. The contents of the Message List are available (4) to the algorithm, which comprises a *Strategy* module containing schemas and query plans. The Strategy in general calls for one or more probes of base relations which are sent by (3) and (1). Replies to the probes return by the same path (1) and (2) as the update messages, and are placed in the Message List where any needed compensation to the reply is computed. The reply is removed from the list when its effects have been incorporated into the computation for the update it belongs to. When processing of an update message is complete and the update message has reached the head of the Message List, the computed changes are installed (5) in the materialized view and the update is removed from the Message List.

Since the probe replies must be capable of compensation for interfering updates, we refer to them as *locally compensatable*. This is related to the property that qualifies a view as being self-maintainable, but the domain of applicability is different. Recall that our view queries are not required to be self-maintainable, but are maintained by locally-compensatable probes. These are queries which access only a single data

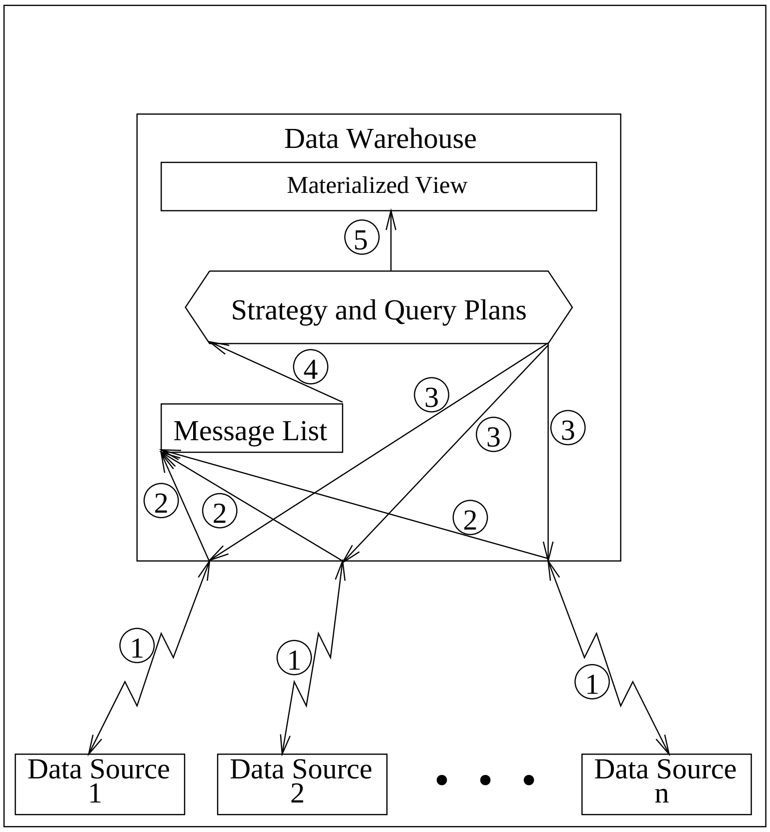

Figure 7: The Computational Model

source using information already available at the data warehouse. Because of the requirement that probes be locally compensatable, we generally limit our probes to conjunctive queries with a single atom (relation term) involving the relation at the target of the probe, and values (perhaps previous results) known at the data warehouse at the time the probe is constructed. The single exception is a form of self-join of the target relation, with two atoms involving the target, that is used in one of our examples and is described in more detail below.

In [GL95, Qua96] formulas are given for the incremental computation of bag algebra queries. As pointed out in [Qua96] the maintenance of views containing the MIN or MAX functions requires special treatment when the current extreme value is deleted, no obvious replacement is inserted, and the group is not completely emptied. In our computational model, this is supported by an additional round of probes of the base relations to compute the view for the group or groups for which this condition holds. Implementation of this additional round can take the form of an SQL query, subject to the query optimization techniques of distributed queries (see, for example, [ÖV91]), and is not considered further in this paper. The AVG function can be implemented by maintaining the values of COUNT and SUM [Qua96] and we assume that these values are maintained and available in any view containing AVG.

The presence of a HAVING clause in the view query for a materialized view can introduce great difficulty for incremental maintenance. Depending on the degree of difficulty, several solutions are possible. One solution is for the warehouse to materialize an intermediate view with a view query lacking the HAVING clause and derive the given view from this intermediate view. In special cases of HAVING predicates, it

M (initial)

A	B	C	Cardinality
1	2	A	+2

Update M

A	B	C	Cardinality
3	1	B	+1

M (updated)

A	B	C	Cardinality
1	2	A	+2
3	1	B	+1

Reply

A1	B1	C1	A2	B2	C2	Cardinality
1	2	A	3	1	B	+2

Corrected Reply

A1	B1	C1	A2	B2	C2	Cardinality
1	2	A	3	1	B	0

Figure 8: Case 1 - A candidate for compensation

is possible to take an approach similar to that suggested above for supporting aggregates, i.e. to detect conditions where the group must be recomputed. Finally, the approach of the Concurrent Posse algorithm presented below always works because it obtains all of the base relation data, and can compute an arbitrary SQL query based on that information.

5.3 Compensation of Join Queries

This paper does not attempt to map the space of locally compensatable queries, but we would like to justify our statements that joins in general are not admissible, but that a special case of self-joins can be used. This may serve to indicate some of the difficulties to be encountered in mapping this space.

Since probes interrogate only a single data source, we illustrate three self-join queries by means of abstract examples. To illustrate a join which is not locally compensatable let us consider the relation M with attributes A, B and C. Let the probe be defined as

```
SELECT M1.A AS A1, M1.B AS B1, M1.C AS C1, M2.A AS A2, M2.B AS B2, M2.C AS C2
  FROM M M1, M M2
  WHERE M1.A = M2.B
```

Consider the initial relation in Figure 8 and a conflicting update (Update M) consisting of the insertion of a single tuple. The probe sees the relation after the update (M updated) and returns the result relation (Reply).

Since the probe applied to the initial state of the relation should return an empty relation, it appears that the appropriate compensation for the interfering update is to reduce the cardinality of the tuple in the reply by 2, resulting in a tuple with cardinality 0, i.e. deleting it.

However, there is another case that the data warehouse cannot distinguish from this one, in which this compensation is not correct. Consider the scenario illustrated in Figure 9. Here, the initial state of the relation and the corrected reply are different, but the other elements of the scenario are all the same as in Figure 8. Since the probe is the same and all data available at the warehouse are the same (i.e. the

M (initial)

A	B	C	Cardinality
1	2	A	+1
3	1	B	+1

Update M

A	B	C	Cardinality
3	1	B	+1

M (updated)

A	B	C	Cardinality
1	2	A	+1
3	1	B	+2

Reply

A1	B1	C1	A2	B2	C2	Cardinality
1	2	A	3	1	B	+2

Corrected Reply

A1	B1	C1	A2	B2	C2	Cardinality
1	2	A	3	1	B	+1

Figure 9: Case 2 - An incompatible compensation

interfering update and the reply,) the data warehouse cannot distinguish these two cases without further information from the data sources, and this probe query is not locally compensatable.

One might suspect from this that all joins fail to be locally compensatable, but there is a special form of self-join that returns sufficient information about the tuples contributing to the result that the query can be compensated. The characteristic that allows this is that the cardinality of every tuple that contributes to the reply successfully can be inferred from the reply. The most straightforward form of such a self-join is one in which all of the self-join conditions are equality involving the same attribute on both sides, and there are no other conditions.

Consider a modification of the probe query so that the WHERE clause compares the same attribute on both terms.

```
SELECT M1.A AS A1, M1.B AS B1, M1.C AS C1, M2.A AS A2, M2.B AS B2, M2.C AS C2
  FROM M M1, M M2
  WHERE M1.A = M2.A
```

We illustrate an example similar to Case 1 in Figure 10, and adjust the values in the "B" tuples so that the reply to the probe is not empty, but the example is otherwise the same. In this case, because tuples join with the same attribute, there are more different tuples in the join. In particular, each tuple of the base relation that contributes to the reply at all includes a contribution joined to itself with cardinality equal to the square of the cardinality of the tuple in the base relation. This allows the data warehouse to infer the cardinality of the relation and compute the correct compensation.

Figure 11 illustrates a change to the initial value, similar to the change that exposed the problem with Case 1 and 2. In this case, because the cardinalities of the tuples in the base relation are reflected in the cardinalities in the reply, the replies are different, and the correct compensation can be computed.

Although a pure self-join of this type is locally compensatable, the addition of a single other condition to the query can cause problems, particularly in compensating for interfering updates containing deletes. We consider now an additional modification of our query.

M (initial)

A	B	C	Cardinality
1	2	A	+2

Update M

A	B	C	Cardinality
1	3	B	+1

M (updated)

A	B	C	Cardinality
1	2	A	+2
1	3	B	+1

Reply

A1	B1	C1	A2	B2	C2	Cardinality
1	2	A	1	2	A	+4
1	2	A	1	3	B	+2
1	3	B	1	2	A	+2
1	3	B	1	3	B	+1

Corrected Reply

A1	B1	C1	A2	B2	C2	Cardinality
1	2	A	1	2	A	+4
1	2	A	1	3	B	0
1	3	B	1	2	A	0
1	3	B	1	3	B	0

Figure 10: Case 3 - a possible compensation

M (initial)

A	B	C	Cardinality
1	2	A	+1
1	3	B	+1

Update M

A	B	C	Cardinality
1	3	B	+1

M (updated)

A	B	C	Cardinality
1	2	A	+1
1	3	B	+2

Reply

A1	B1	C1	A2	B2	C2	Cardinality
1	2	A	1	2	A	+1
1	2	A	1	3	B	+2
1	3	B	1	2	A	+2
1	3	B	1	3	B	+4

Corrected Reply

A1	B1	C1	A2	B2	C2	Cardinality
1	2	A	1	2	A	+1
1	2	A	1	3	B	+1
1	3	B	1	2	A	+1
1	3	B	1	3	B	+1

Figure 11: Case 4 - A compatible compensation

M (initial)

A	B	C	Cardinality
1	2	A	+1
1	3	B	+1

Update M

A	B	C	Cardinality
1	2	A	−1

M (updated)

A	B	C	Cardinality
1	3	B	+1

Reply

A1	B1	C1	A2	B2	C2	Cardinality
empty						

Corrected Reply

A1	B1	C1	A2	B2	C2	Cardinality
1	2	A	1	3	B	+1

Figure 12: Case 5 - A problem with side conditions and deletes

```
SELECT M1.A AS A1, M1.B AS B1, M1.C AS C1, M2.A AS A2, M2.B AS B2, M2.C AS C2
  FROM M M1, M M2
  WHERE M1.A = M2.A AND M1.C="A"
```

Now consider the scenario illustrated in Figure 12. Here the interfering update is the deletion of a single tuple, but as a result of this deletion, the reply to the probe is an empty relation, and there is not enough information at the data warehouse to compute the correct compensation for this update because the "side" condition hides the remaining tuple.

5.4 Concurrent Posse

The computational model as presented here can achieve both complete consistency and the maximum possible concurrency. We outline an algorithm called Concurrent Posse which achieves these results. Its value lies in realizing full concurrency and expressive power, while retaining both correctness and complete consistency, and thus serves to indicate the full range of solutions available within our conceptual framework. Moreover, we use this extreme as the starting point for the optimizations we introduce later. We use a conceptual model for comparison and evaluation of update strategies.

The concurrency is achieved by installing a Strategy that, in service of each update, simply queries all data sources for their entire content, applying the view query to the compensated replies to compute a new materialized view. These queries can be issued concurrently for all updates and all data sources, as each update message arrives. The compensation algorithm if Section 4.3 assures consistency while the restriction of installing views in the order of the arrival of their corresponding updates assures the completeness of the consistency.

The Concurrent Posse algorithm operates in the computational model to define an extreme of concurrency at the cost of large message sizes, so that a spectrum of solutions can be realized between Concurrent Posse on the one hand and the existing algorithms [ZGMHW95, ZGMW96, AESY97] on the other. That such an extreme approach is feasible demonstrates the flexibility of the framework proposed here, and suggests the possibilities that lie between extremes.

This algorithm queries all data sources for each update, including the data source whose update message is being processed. It is in effect, recreating afresh a snapshot of the Correctness Model of Section 5.1 for each update. Complete consistency with the correctness model follows directly from this property of the algorithm and the order of installation of the results..

Having established the fully concurrent extreme of the spectrum of solutions, and having the opposite extreme of serial probing in prior work, where both extreme points may be impractical for one reason or another, it becomes possible to explore the points in the middle of the spectrum for opportunities for optimization. We do this by trading concurrency for message cost, in the framework provided by our computational model and the optimizing view maintenance algorithm presented in the next section.

6 The Posse Framework

The Concurrent Posse algorithm may be impractical because of the message sizes that would result from asking for the transmission of all base relations in response to each update. On the other hand, if the data warehouse is going to send all queries immediately on receiving the update message, there may be insufficient information at hand to refine the query at that moment. This presents us with an opportunity to trade off two contributors to overall efficiency: concurrency and message size.

The Optimized Posse algorithm that we present in this section, operating in the computational model we have already presented constitute the Posse framework, in which such tradeoffs can be conveniently exploited. We impose a conceptual structure on the schema and update processing, using rooted directed graphs to represent the organization of the algorithms we use.

6.1 Schema and Rooted DAG

At the data warehouse the Strategy associates a *query plan* with each data source, to be used for each update message received from that data source. Each query plan represents a schedule for issuing probes in response to an update message from a particular data source. We present each such query plan as a rooted directed acyclic graph (rooted DAG), with one vertex for each occurrence of a relation in a FROM clause in the view query.

Each non-root vertex is associated with a query to be used to probe the corresponding data source for the data relevant to the use of that instance of the relation in the processing of the update. Each vertex is labeled; the same name is used for any update message, probe or data associated with that vertex. Edges represent a dependency that can be expressed as "is completed before composing the probe of," and indicate that the probe for the destination vertex is delayed until the data from the source of that edge is available. For Concurrent Posse the DAG is star-shaped: it contains an edge from the root to each leaf; this is our usual starting point for developing the final DAG.

In general there is also a final query over the original update message and the probe responses, which produces the set of changes to be installed in the materialized view. We assume that this query is executed at the data warehouse after all probe replies have been received and corrected for interfering updates. In our examples this is always a join of the update with all the delta tables and is not explicitly presented.

Returning to our example view query of Section 4, we have the graphs of Figure 13 which correspond to choices similar to the Concurrent Posse algorithm in that all probes are sent as soon as the update message arrives. This similarity to Concurrent Posse is reflected in the fact that each graph in the figure has the star configuration, i.e. all edges originate in the root, and there is an edge leading to each non-root vertex. As can be seen from the label on the root of each graph, Figure 13A represents the query plan for update messages from the Assists relation, Figures 13B and 13C represent the query plans for update messages from the two fictitious clones T1 and T2 of the Teaches relation, and Figure 13D represents the query plan for update messages from the Enrollment relation.

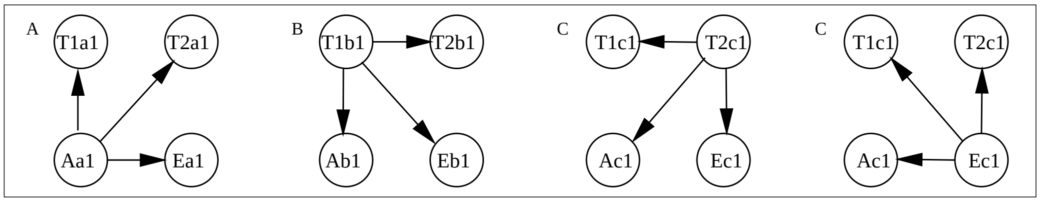

Figure 13: Directed star graphs for unoptimized query plans for DoubleStatus

The queries associated with the non-root nodes in these two figures are simple requests for an entire relation:

- Figure 13A, used for updates from Assists

  T1a1 `SELECT * FROM Teaches`

  T2a1 `SELECT * FROM Teaches`

  Aa1 represents the update message

  Ea1 `SELECT * FROM Enrollment`

- Figure 13B, used for updates from Teaches, with similar probes.
- Figure 13C, used for updates from Teaches, with similar probes.
- Figure 13D, used for updates from Enrollment, with similar probes.

We concern ourselves in this section with optimizations that correspond to adding or deleting one or more edges to the star-shaped graph for a query plan and choosing an appropriate query to associate with the destination vertex. Edges are deleted when the probe at the destination node has been combined with a probe at another node for the same data source. Added edges reduce the concurrency in the query plan by delaying the probe at the destination vertex until completion of the probe at the source vertex. The advantage in doing this derives from the possibility of greatly reducing message size, local processing cost or both. Properly chosen, these optimizations can improve query performance at the data source and message size in the response at the expense of loss of concurrency in processing.

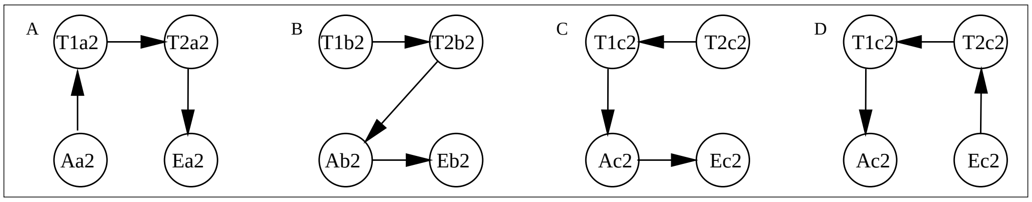

Figure 14: Directed graphs in the spirit of Strobe and SWEEP

The other extreme of the spectrum of algorithms is represented by the graphs of Figure 14. These are in the spirit of the Strobe and SWEEP algorithms, as they might be extended to apply to our example. These algorithms as published do not specify details of handling self-join, or how they would arrange the nodes of a cyclic query, so we have made some assumptions in these extensions. For instance, regarding this example as an extension of SWEEP, we consider the nodes to be arranged linearly as Assists-Teaches-Teaches-Enrollment, but if an update arrives from the Teaches relation, we will process the self-join first. Thus in Figure 14C, in which an update arrives from the T2 clone of the Teaches relation, the self-join is processed, then the "left" neighbor Assists is probed, and finally the "right" neighbor Enrollment is probed. Each probe consists of a query of a single base relation and the reply is joined to the accumulating data. The probes on behalf of a given update message are performed serially.

6.2 Optimized Update Processing

In planning the maintenance of the DoubleStatus view one can apply various transformations to the query plan, as might be done by a query optimizer. We start with a "star-shaped" graph with edges from the root directly to each of the leaves and consider transformations that can be represented using the graph and associated probes.

These transformations are used in the graphs of Figure 15, which correspond to the example introduces in Section 4. First, each edge in the graph can be associated with a join involving information from the source node of the edge, and any previous nodes on the path from the root. Compared to simply requesting the entire content of the relation, this probe can (vastly) reduce message size.

Second, a probe can be delayed as seen in Figure 15A at node Eb3; this probe of the Enrollment relation is delayed until the reply to the previous probe is available at the warehouse, providing information that can be used in the join.

Third, in a few cases multiple probes of a single relation can be combined, as seen in Figure 15C at node T1c3 and T2c3. Node T1c3 has actually been bypassed because appropriate manipulation of the reply to the probe T2c3 can reconstruct the reply that would be received for T1c3. This optimization applies because the join between the two nodes is an equijoin with corresponding attributes.

Fourth, there is no query plan rooted at T2 because we have used a self-join at node T2c3, as described in Section 5.3.

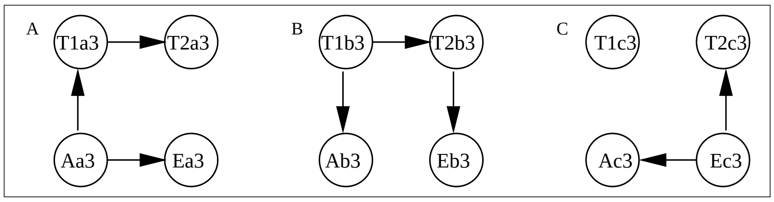

Figure 15: Directed graphs for optimized query plans for the DoubleStatus view. Graph A is used for updates from the Teaches relation, Graph B for updates from Assists, and Graph C for updates from Enrollment.

These transformations are used in the graphs of Figure 15, with the following probes:

- Figure 15A, used for updates from Assists

T1a3 SELECT * FROM Aa3, Teaches T1
WHERE Aa3.Course = T1.Course

T2a3 SELECT * FROM T1a3, Teaches T2
WHERE T1a3.Prof = T2.Prof

Aa3 represents the update message

Ea3 SELECT * FROM Aa3, Enrollment E
WHERE Aa3.Student = E.Student

Here, we have used joins to potentially reduce the size of replies, and delayed the T2a3 probe by adding the $T1a3 \rightarrow T2a3$ edge so that a join can be used. The $Aa3 \rightarrow T2a3$ edge is now redundant, and has been dropped.

- Figure 15B, used for updates from Teaches

T1b3 represents the update message

T2b3 SELECT * FROM T1b3, Teaches T2
WHERE T1b3.Prof = T2.Prof

Ab3 SELECT * FROM Assists NATURAL JOIN T1b3

Eb3 SELECT * FROM Enrollment NATURAL JOIN T2b3

Here, we have used the same transformations as in Figure 15A, applied to different edges and nodes. The Eb3 node is the one delayed in this case.

- Figure 15C, used for updates from Enrollment

T1c3 null: combined into T2c3

T2c3 SELECT * FROM Teaches T1, Teaches T2
WHERE T1.Prof = T2.Prof

Ac3 SELECT * FROM Ec3, Assists A
WHERE Ec3.Student = A.Student

Ec3 represents the update message

Here, we have used joins to potentially reduce the size of replies, and have combined the two vertices representing the Teaches relation, and used a self-join query. In effect this moves the computation from the data warehouse to the data source, and reduced the number of probes by one. As a result, there is no activity for vertex T1c3, and no probe is delayed.

The transformations chosen in these examples may or may not be the best in a given case; in particular the self-join for T2c3 in Figure 15C has quadratic worst-case message size behavior. Moreover, it may be better to use one of the variants of semijoin [KYY82] for the other joins, and it may be better to choose different or additional dependency edges to add to the graph. Our purpose here is not the optimal choice of transformations but the description of a framework in which transformations can be applied to the algorithm for view maintenance. In any event, the goal of optimization is often achieved by the avoidance of very bad solutions, and it appears that this goal is achieved in these examples, and that a high degree of concurrency is maintained. There are also properties common to warehouses that may give rise to special opportunities for optimization not exploited here, such as multiple views in a warehouse [PS88, Sel88], or the arrival of multiple updates in a short period of time [CM96].

6.3 The Optimized Posse Algorithm

The Optimized Posse algorithm formalizes the notion that the probes are initiated in a particular order rather than all at once as in Concurrent Posse. The opportunity for optimization lies in the fact that having information from prior probes may reduce the size of subsequent replies and intermediate results.

Figure 16 presents the Optimized Posse algorithm, which uses query plans to schedule possibly concurrent probing to perform incremental view maintenance. We presume that the query plans are the output of an optimizer process which runs when the view query is defined. The algorithm is presented here from the point of view of a scheduler; in effect each update and each branching withing the query plan for that update creates a separate thread of control which is not represented here, but is inherent in the DAG for the query plan..

There are several functions and methods needed by the Optimized Posse algorithm. Suitable concrete instances of these features will necessarily be included in any actual implementation.

- $sender(message)$ is the data source which sent the given message.
- $belongs(message)$ is the update message to which this message belongs.
- $BuildDeltaTable(DeltaTable, message)$ includes the effects of a subsequent message into the ongoing compensation computation.
- $ComputeCorrection(reply, DeltaTable)$ corrects a reply by applying a DeltaTable computed from interfering updates.
- The object data type PLAN refers to a class of objects which represent a query plan. An instance of a PLAN object is used in the Optimized Posse algorithm to organize the response of the data warehouse to an update from a particular data source.
- PLAN.status[n] is an instance variable array representing status of queries for each node.
- PLAN.$buildProbe(j, \Delta V)$ is the class method to build the query of the data source corresponding to j, given the current accumulated update ΔV.
- PLAN.$applyChange(\Delta V, j, Reply, DeltaTable)$ builds a new ΔV given the current ΔV, the $Reply$ response to a probe of data source j, and $DeltaTable$ the accumulated correction computed from conflicting updates.

This algorithm begins by initiating one copy of the *LogAndStartUpdates* process for each data source. This process listens for and appends incoming messages to *MessageList* and in the case of update messages (ΔR_i) starts a *ViewChange* process to compute the effect of the update and to incorporate the change into the materialized view.

Each instance of *ViewChange* operates under the control of the delta query plan which is associated with the data source from which the update message originated. It proceeds by repeating a cycle until all probes have been completed, the cycle consisting of these steps:

- Initiate any probes which have not already been initiated, and for which all the necessary information is at hand (i.e. all probes whose replies are "used in" this probe have been completed).
- As each probe reply arrives, examine the *MessageList* for conflicting update messages (i.e. those from the same data source whose probe reply is being considered, and which arrived between the time of arrival of the original update message and that of the probe reply). Calculate the correction to the probe reply corresponding to all such conflicting updates, and incorporate the compensated effect of the probe reply in the calculated change to the materialized view (ΔV). Remove the probe reply from *MessageList*; it is used only by the instance of *ViewChange* which issued the probe.

When the cycle is complete, special processing is performed as required for MIN and MAX aggregate functions and for group selection, as discussed in section 5.2. If these features do not appear in the view query, there is no special processing. There are several approaches to the support of group selection; depending on the nature of the HAVING predicate and the approach taken, this special processing will take different

```
MODULE DataWarehouse;
    GLOBAL DATA
        V: RELATION;                  /* Initialized to the correct view */
        n, m: INTEGER;                /* Inititalized: number of data sources, nodes in DAG; n ≤ m */
        QueryPlanList[n]: PLAN;       /* Initialized to the query plans */
        MessageList: LIST;            /* Initially empty */
        NextMessage: INTEGER;         /* Initially 0 */

    PROCESS ViewChange(ΔR : RELATION, UpdateSource: INTEGER, ThisMessage: INTEGER)
    VAR
        ΔV: RELATION;
        Correction: DELTATABLE;
        Probe[m]: QUERY;
        j: INTEGER;
        QueryPlan: PLAN;
        M: MESSAGE;
    BEGIN
        ΔV = ΔR;
        QueryPlan = QueryPlanList[UpdateSource];    /* Initialized: all QueryPlan.status[*] = "new" */
        QueryPlan.status[UpdateSource] = "done";
outer:  LOOP (FOREVER)                /* Block when waiting for messages */
            FOR j = EACH j such that QueryPlan.status[j] == "new" and all incoming edges of node j are marked "done" DO
                Probe[j] = QueryPlan.buildProbe(j, ΔV);
                SEND (Probe[j],ThisMessage) TO Data Source j;
                QueryPlan[j] = "queried";
            ENDFOR;
            FOR M = each new message in MessageList from ThisMessage on DO /* i.e. without repeating */
                j = sender(M);
                IF M is a probe reply AND belongs(M) = ThisMessage THEN
                    Correction = ∅;
                    FOR EACH message N = ΔR_k where node j represents data source k FROM ΔR TO M DO
                        Correction = BuildDeltaTable(Correction, N, Probe[j]); /* Correct all nodes representing this source */
                    ENDFOR;
                    ΔV = QueryPlan.applyChange(ΔV, j, M, Correction);
                    QueryPlan[j] = "done";
                    REMOVE M FROM MessageList;
                    IF all QueryPlan.status[j] are marked "done" THEN BREAK outer; ENDIF;
                ENDIF;
            ENDFOR;
        ENDLOOP;
        IF any groups have lost their MIN or MAX extremum THEN evaluate the full view query for those groups;
        IF any groups require recomputation for a HAVING clause THEN evaluate the indicated query plan for those groups;
        BLOCK UNTIL ThisMessage is at the head of MessageList;
        V = V + ΔV;
        REMOVE ThisMessage FROM MessageList;
    END ViewChange;

    PROCESS LogAndStartUpdates(i : INTEGER);
    VAR
        M: MESSAGE;
        i: INTEGER;
    BEGIN
        LOOP
            RECEIVE M FROM Data Source i;
            IF M is ΔR_i THEN
                APPEND (M, i, NextMessage) TO MessageList; /* atomic */
                StartProcess(ViewChange(M, i, NextMessage ++))
            ELSE                /* M is Reply(msgnum, node) */
                APPEND (M, node, msgnum) TO MessageList; /* atomic */
            ENDIF;
        FOREVER;
    END LogUpdates;

VAR
    i: INTEGER;
BEGIN /* Start DataWarehouse Processes */
    NextMessage = 1;
    FOR (i = 1; i ≤ n; i ++) DO /* One listener per data source */
        StartProcess(LogAndStartUpdates(i));
    ENDFOR;
END DataWarehouse.
```

Figure 16: The Optimized Posse Algorithm

forms or be omitted entirely. Finally, *ViewChange* waits until the update message that is being processed becomes the first item in *MessageList*, then applies its change to the materialized view and removes the update message from *MessageList*.

The final step of *ViewChange* ensures that updates are incorporated into the materialized view in the same order that their update messages arrived at the data warehouse. This property of Optimized Posse ensures that complete consistency with the correctness model is maintained.

6.4 Remarks

For clarity, some further opportunities for optimization have been ignored but may be added without introducing undue complexity to the algorithm. In particular, if the resulting view change can be determined at an early stage to be null, it would make sense to stop the process of issuing probes. Similarly, if the results of a probe can be known in advance (say, to be empty) it would make sense not to issue the probe. These situations can arise in a number of ways, notably when the view query is at least partially a join over the base relations, and at some point the accumulated relation being joined with is null, and also in regard to the data source that originated the update message being processed, when the update message itself contains the desired information.

It is possible to improve performance by the batching of updates. The resulting algorithm could acheive strong consistency rather than complete consistency. It is also possible to commit the updates in an arbitrary order. With some care to hide temporarily negative cardinalities such an approach could easily produce convergence; maintaining consistency in this case seems to require a data structure that is exponential in the number of simultaneous uncompleted updates, and the consistency would be weaker than the strong consistency defined above.

As another example, as noted in [ZGMW96], the presence in the DoubleStatus view of a key for the Enrollment relation allows the processing of deletes to that relation without reference to other data sources. The Optimized Posse algorithm also calls for a separate probe for each non-root occurrence of a base relation in a FROM clause, even when the same relation occurs more than once. As noted in Section 6.2, some of these probes may be optimized away. For clarity, provisions for omitting these probes have been omitted from the algorithm in Figure 16.

7 Conclusion

We have presented a correctness model for evaluating the consistency and correctness of algorithms for the transport of base relation information for the incremental maintenance of a materialized view in a data warehouse. We have shown that a completely concurrent algorithm (Concurrent Posse) obtains complete concurrency of the probing process, albeit at the cost of increased message sizes. We have presented the Posse framework in which distributed query optimizations can be applied to the Concurrent Posse approach in order to obtain reasonable performance of the incremental maintenance process, and have indicated areas in which further improvements might be obtained. We have maintained the criterion of complete consistency with a correctness model that honors the order of arrival of update messages from the data sources, although we mention variations with different behavior. Our present approach uses fixed query plans determined in advance. In the future we plan to investigate dynamically determining the query plan and to explore alternate optimization criteria for choosing the query plans, and to fully characterize locally compensatable probes.

References

[AESY97] D. Agrawal, A. El Abbadi, A. Singh, and T. Yurek. Efficient View Maintenance in Data Warehouses. In *Proceedings of the 1997 ACM International Conference on Management of Data*, pages 417–427, May 1997.

[BC81] Philip A. Bernstein and Dah-Ming W. Chiu. Using semi-joins to solve relational queries. *JACM*, 28(1), January 1981.

[CM96] Latha S. Colby and Inderpal Singh Mumick. Staggered Maintenance of Multiple Views. In Views 96 Workshop [Vie96].

[GL95] T. Griffin and L. Libkin. Incremental Maintenance of Views with Duplicates. In *Proceedings of the ACM SIGMOD 1995 International Conference on Management of Data*, pages 328–339, May 1995.

[GMS93] A. Gupta, I. S. Mumick, and V. S. Subramanian. Maintaining Views Incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 157–166, May 1993.

[Huy96] Nam Huyn. Efficient view self-maintenance. In Views 96 Workshop [Vie96].

[HZ96] Richard Hull and Gang Zhou. A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 481–492, June 1996.

[KYY82] Y. Kambayashi, M. Yoshikawa, and S. Yajima. Query Processing for Distributed Databases using Generalized Semi-Joins. In Mario Schkolnick, editor, *Proceedings of the ACM SIGMOD 1982 International Conference on Management of Data*, pages 151–160, May 1982.

[O'G98] Kevin O'Gorman. A Paralllel Framework for Maintaining SQL Views at a Data Warehouse. Master's thesis, University of California at Santa Barbara, Department of Computer Science, UCSB, Santa Barbara, CA 93106, 1998.

[ÖV91] M. Tamer Özsu and Patrick Valduriez. *Principles of Distributed Databases Systems*. Prentice Hall, 1991.

[PDI96] *Proceedings of the International Conference on Parallel and Distributed Information Systems*, December 1996.

[PS88] Jooseok Park and Arie Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, February 1988.

[QGMW96] Dallan Quass, Ashish Gupta, Inderpal Mumick, and Jennifer Widom. Making Views Self-Maintainable for Data Warehousing. In PDIS 96 [PDI96].

[Qua96] Dallan Quass. Maintenance Expressions for Views with Agregation. In Views 96 Workshop [Vie96].

[Sel88] Timos K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1), March 1988.

[Vie96] *Proceedings of the ACM Workshop on Materialized Views: Techniques and Applications*, Montreal, Canada, June 1996.

[ZGMHW95] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View Maintenance in a Warehousing Environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, May 1995.

[ZGMW96] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In PDIS 96 [PDI96].

[ZR99] Xin Zhang and Elke A. Rundensteiner. Psweep: Parallel view maintenance under concurrent data updates of distributed sources. Technical Report WPI-CS-TR-99-14, Worcester Polytechnic Institute, Computer Science Department, Worcester, MA, 01609, March 1999.