Performance Analysis of WHIPS Incremental Maintenance

Yue Zhuge and Hector Garcia-Molina
Computer Science Department
Stanford University
Stanford, CA 94305-2140, USA
{zhuge,hector}@cs.stanford.edu

Abstract

Incremental maintenance incorporates new changes automatically and continuously into a data warehouse, and seems to be the best maintenance solution for very large warehouses. However, the performance of incremental maintenance algorithms is not well understood, and commercial incremental maintenance systems are still not widely available. In this paper, we study the performance of WHIPS, a prototype system developed at Stanford focusing on warehouse maintenance. We examine the efficiency of change propagation and study factors that determine the efficiency. We also compare the relative performance of different warehouse maintenance techniques.

1 Introduction

A data warehouse stores integrated data from multiple, often distributed and autonomous data sources, and is typically used for decision support. One of the major challenges is to correctly and efficiently update the warehouse as the source data changes. There are basically two categories of techniques for this: refreshing and incremental maintenance. Refreshing means that the entire warehouse is re-computed, either on demand or periodically. Incremental maintenance means that as source data change, new information is gathered and integrated into the warehouse without throwing away all the existing data.

Most current warehousing systems use refreshing because of its simplicity. However, incremental maintenance has important advantages that may make it more attractive than refresh. First, some applications may wish to observe the actual source changes, not just a periodic snapshot. For example, a weather warehouse may need to trace the history of temperatures in cities. A periodic snapshot of the weather may not be sufficient. Second, a complete refresh may be infeasible because of its duration. In particular, a warehouse may be unavailable to users during refresh, so refresh is often done at night or on a weekend. However, for large warehouses, the refresh time may exceed the available "update window" of a few hours. Furthermore, global corporations may need 24×7 operation and cannot have any downtime. Incremental maintenance performs a relatively small number of operations on each update, so it may be done concurrently with warehouse user transactions. Third, some applications may need updates to be reflected quickly, e.g., an application that tracks stock market changes to perform trading. Only incremental maintenance can provide this data freshness. Other advantages offered by incremental maintenance include reduced network traffic and less storage space for intermediate results.

Although incremental maintenance offers advantages, its actual performance is not well understood. Will it really offer the desired performance? Will updates be delayed because they are performed at a low level of granularity? How do schemes that enforce warehouse correctness perform? What is the best architecture for a warehouse that implements incremental maintenance? Is it better to process updates individually or in batches? Is it cost effective to copy of all source data at the warehouse, which may simplify update processing?

To answer some of these questions we built a warehouse prototype called WHIPS (WareHousing Information Project at Stanford). This prototype uses incremental algorithms to automatically and consistently maintain its warehouse. As far as we know, WHIPS is the first such working experimental prototype. In this paper we describe this prototype, and present experimental results that provide insights into the incremental operation of a warehouse. The following are our major goals:

1. To understand how a distributed incremental view maintenance system performs. For example, how long does it typically take to propagate a source update to the warehouse? Does it take seconds or milliseconds? How much data is transferred from the sources and between different warehouse modules? What are the potential system bottlenecks? What system parameters have the greatest impact on performance?

2. To compare the relative performance of different view management algorithms. As we will see in Sections 6.2 and 6.3, views can be incrementally maintained using different levels of correctness and complexity. Which algorithm works best for a given environment and update scenario? How much more expensive is it to use a more complex yet "more correct" view maintenance algorithm, as opposed to a simpler but "less correct" one? Some algorithms only work if warehouse tables contain all source keys. Do these algorithms perform better? If so, can we quantify the benefits?

3. To evaluate the WHIPS design and implementation decisions. For example, we decided to implement WHIPS as independent modules communicating through CORBA. This design offers significant flexibility. For instance, the query processing component can be modified or replaced without changing the view management component. On the other hand, we pay some performance penalties due to inter-module communication. How substantial are these penalties? As another example, we also implemented some naive distributed query processing algorithms. How much can we save if we improve the module that handles query processing?

4. To give suggestions to future warehouse designers and implementers. For example, what is the most efficient way to propagate a group of source updates to the warehouse? Which part of the system

should be optimized in order to achieve the greatest performance?

We organize the paper according to the questions we want to answer. We start with some preliminaries: related research (Section 2), the WHIPS architecture (Section 3), and our performance criteria and relevant system parameters (Section 4). In Sections 5 through 8 we focus on important performance factors: modularity, keys, consistency, source update scenarios and batching.

Appendix A discusses the confidence of the timing results presented in this paper. Appendix B presents an analytic model that roughly predicts WHIPS performance. This model helped verify the correctness of the system implementation and served as a guide for our performance experiments.

2 Related Work

Overviews of data warehousing research and technologies can be found in [LW95, SC97]. Basic incremental view maintenance algorithms are described in [BLT86, GMS93, GM95, GL95]. Reference [Han87] discusses performance tradeoffs for materialized views in a conventional DBMS. The algorithms implemented in WHIPS are developed in [ZGMHW95, ZGMW96]; they handle distributed autonomous data sources and continuously interleaving updates, and provide view consistency.

In the workshop paper [WGL+96] we presented a short preliminary description of the WHIPS architecture, applications and implementation. Another incremental maintenance prototype that handles Select-Project views is the Squirrel Project [HZ96a]. A related preliminary performance study comparing the performance of virtual and materialized approaches is given in [HZ96b].

3 WHIPS

In this section we describe the WHIPS architecture, and we illustrate its run-time functionality by following the propagation of one source update. WHIPS handles relational SPJ views, as well as derived views defined by union or aggregate operators. However, this performance study focuses on base SPJ views only.

3.1 Architecture

Figure 1 depicts the part of the WHIPS architecture that is relevant to our performance study. The key WHIPS feature is modularity. As shown in the figure, the system is composed of distinct modules that communicate with each other. Each module is a separate process. We implemented each module as a CORBA object, using the ILU implementation of CORBA [CJS+94].

We believe that this modular design achieves several important goals: (1) Heterogeneity. WHIPS is able to handle various types of data sources, as may exist in a real data warehousing environment. Any database can be used as the data warehouse. There are also a variety of view maintenance algorithms that can be used. (2) Distribution. Data sources are distributed from the integration modules and the data warehouse. Different integration modules may reside on different machines. As the load grows, maintenance work can be distributed among different processes, across multiple machines, and performed concurrently. (3) Scalability. The number of modules that handle view maintenance (know as view managers) and global query processing (know as query processors) can be determined according to the system workload. (4) Implementation flexibility. Each integration module can be implemented and optimized independently.

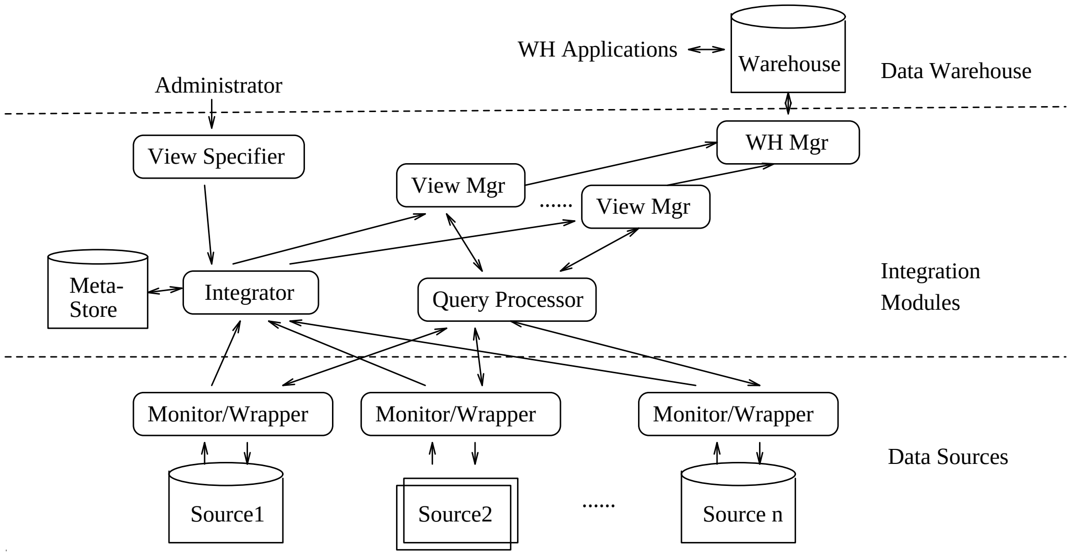


Figure 1: The WHIPS system architecture

In general, WHIPS is divided into three layers: the data sources, the integration modules, and the data warehouse. Each data source is encapsulated by a source-specific monitor and wrapper. A *monitor* is responsible for detecting modifications to the source data and notifying the integrator. We have implemented trigger-based monitors for cooperative data sources and snapshot monitors for other sources [LGM96]. A *source wrapper* translates single source queries from the internal WHIPS representation into queries in the native source language. Thus, the wrapper hides the source-specific querying details from all other modules. Data sources in WHIPS are autonomous; the only requirement is that they be capable (through their wrappers) of answering simple select queries. Currently, WHIPS wrappers handle three types of data sources: flat files, Sybase databases, and Mini SQL databases. New types of data sources can be added to the system as long as wrappers and monitors are written for them.

The *integrator* coordinates the different WHIPS components. At initialization, the integrator locates the warehouse manager and creates one or more query processors. When a new source becomes available, the integrator receives a notification from the source's monitor and wrapper. The integrator then stores the source information (such as the source type, schema, query interface) in the *meta-store*, a persistent store for system information. In the current prototype, the warehouse administrator/users/applications may define their views using a subset of SQL extended with constructs for specifying the desired view consistency (see Section 6). When a view is defined, the *view specifier* parses it and sends the corresponding view tree to the integrator. The *integrator* then creates one *view manager* to maintain this view at the desired level of consistency. After initialization, the integrator coordinates view maintenance by routing source updates to the appropriate view managers.

Each *view manager* is responsible for maintaining a view, using one of the algorithms that we will discuss in Section 6. The view manager first initializes the corresponding view, then receives relevant updates and computes the corresponding view "deltas" (by interacting with the query processor if necessary), and finally sends those view deltas to the warehouse manager for execution. Views can be added and deleted at run time, and view maintenance for the different views can be performed concurrently, possibly using different maintenance algorithms.

A *query processor* receives global queries from the view managers and answers them by posing the appropriate single-source queries to the source wrappers. It then passes the composite global query answers back to the view managers. In WHIPS, a query processor performs distributed query processing using standard techniques such as sideways information passing and filtering of selection conditions to prune the queries it poses to the wrappers [OV91].

The *warehouse manager* receives deltas (lists of the specific inserts and deletes that must be performed to bring the warehouse up to date), and applies them to the warehouse views. All modifications received by the warehouse manager in a single message are applied to the warehouse in one transaction, as needed by the view consistency algorithms [ZGMW96]. Like the source wrapper, the warehouse manager also translates the view deltas to the specific syntax of the warehouse database, thus shielding all other WHIPS modules from the particulars of the warehouse and allowing any database to be used.

3.2 Propagating one source update

We now briefly review incremental maintenance of distributed materialized views, and walk though a WHIPS update. Let V be a materialized view. When changes occur to the source data from which V is derived, an incremental maintenance algorithm computes and applies $\triangle V$, the changes to view V.

Example 1: Incremental view maintenance

Consider a company that handles magazine subscriptions and related issues. One department maintains subscription information in a relation $S = Subscription(Name, Magazine)$, and a second department maintains promotion information $P = Promotion(Magazine, Prize)$. A third department, in charge of shipping and handling, needs to find out what free prizes to send to each subscriber. This information can be obtained by examining view $V = \Pi_{Name,Prize}(S \bowtie P)$. View V can be materialized in a data warehouse with the first two department databases as data sources. Figure 2 provides some sample data for S, P, and V. (The rest of the figure illustrates how WHIPS propagates updates; ignore for now.) For example, since Joe subscribes to Newsweek, he should receive the promotional CD prize.

The view delta corresponding to an update U_s on relation S is $\triangle V = \Pi_{Name,Prize}(U_s \bowtie P)$. For an update U_p on relation P the corresponding delta is $\triangle V = \Pi_{Name,Prize}(S \bowtie U_p)$. Whether the view delta should be inserted into or deleted from the view is determined by whether the source update is an insertion or a deletion. □

The job of the WHIPS integration components is to compute and apply $\triangle V$. When more information from the sources is needed, WHIPS sends *source queries* to obtain that information. In general, each propagation follows the message paths depicted in Figure 2. An asynchronous ILU procedure call is used to send each message.

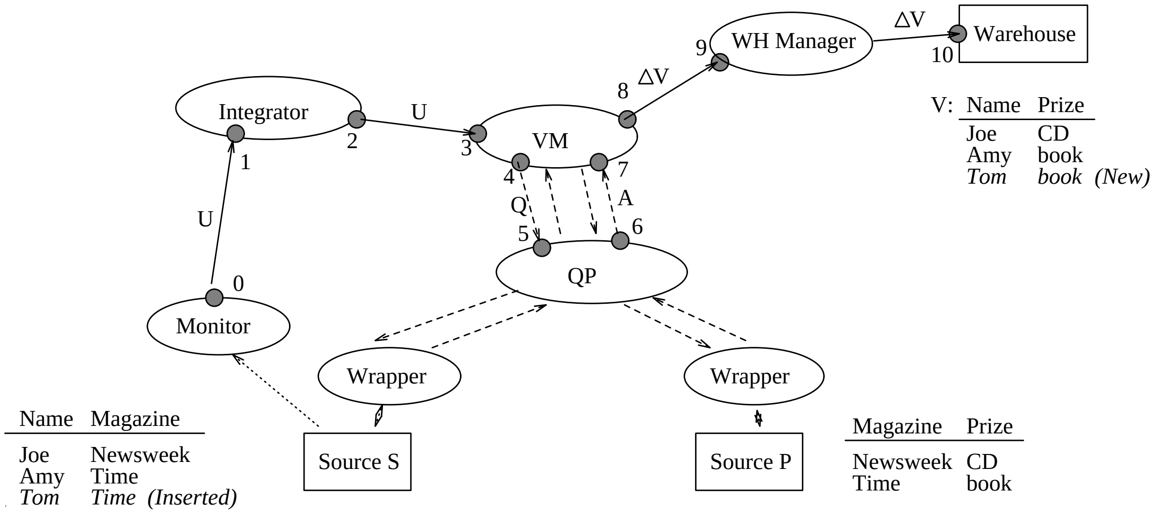


Figure 2: Propagating one source update

We now walk through an update to illustrate the WHIPS components at work. Consider view V of Example 1 and assume it is correctly initialized. Say the initial value of the base relations and the view is as shown in Figure 2. Let there be a source insertion $U_s = insert(S, [Tom, \underline{Time}])$, which indicates that Tom has subscribed to Time magazine. Update U_s is propagated as follows, where each "stop" is a point in time when the system starts or finishes an action.

Stop 0: Update $U_s = insert(S, [Tom, \underline{Time}])$ is ready at a monitor.

Stop 1: The integrator receives U_s;

Stop 2: The integrator forwards U_s to the view manager (VM) for view V;

Stop 3: This VM receives U_s;

Stop 4: The VM sends a query $Q_s = \Pi_{Name,Prize}(U_s \bowtie P)$ to the query processor (QP). We call this query an *update query* because it is triggered by a source update.

Stop 5: The QP receives query Q_s;

Stop 6: The QP finishes processing Q_s and sends the answer $A_s = [Tom, book]$ to the VM. Between stop 5 and 6, the QP sends a *wrapper query* to the source P wrapper asking for all prizes offered by Time magazine. The QP then composes answer A_s from the wrapper query answer.

Stop 7: The VM receives answer A_s;

Stop 8: The VM sends delta view $\triangle V$ to the warehouse manager. Note that $\triangle V$ contains the tuple $[Tom, book]$ obtained from A_s with an indication that it is to be inserted into the view;

Stop 9: The warehouse manager receives $\triangle V$;

Stop 10: $\triangle V$ appears in the view. That is, the transaction that applies $\triangle V$ to V commits, and the view contains the new tuple $[Tom, book]$.

Not all update propagations go through each of the above stops. The shortest path that an update has to go thorough is from the integrator to the view manager to the warehouse manager (stops 0, 1, 2, 3, 8, and 9). For example, some view managers do not send queries to the QP when processing deletions. We let all view deltas, including empty ones, reach the warehouse manager, because this is needed to ensure multiple view consistency [ZWGM97] (which is not studied in this performance paper). Because of compensating queries (Section 6.3), there may be more than one pair of queries and answers between the VM and the QP (between stops 4 and 7). In this case, we assume stop 4 is the time when the VM sends the first update query and stop 7 is the time when the VM receives the last answer.

4 Performance criteria and system setup

In this section we first outline the major performance criteria, then describe the basic setup of our performance experiments. We also present some basic timing results and discuss their confidence.

4.1 Performance criteria

The main criteria we use is *update propagation time*(UPT). We define the UPT for a specific source update and view as the time elapsed from when the integrator receives the update (stop 1 in Figure 2) to when the warehouse manager receives the corresponding view deltas (stop 9). The time is measured using the Unix function `gettimeofday()`. Since in our experiments all the integration modules are located on the same machine, the UPT is just the time difference between stop 9 and stop 1. To refer to sub-components of the UPT, we use $t_{i,j}$ to represent the elapsed time between stop i and j in Figure 2. For example, $t_{5,6}$ is the time between the QP receiving a query Q and sending the corresponding answer, and $t_{6,7}$ is the time between the QP sending the answer and the VM receiving that answer. The full UPT of an update is $t_{1,9}$. A longer time to propagate an update to the warehouse indicates more stale views and an increased possibility of interference between consecutive update propagations.

We did not include in the UPT the time before the update reaches the integrator (before stop 1) because $t_{0,1}$ is specific to the connection between the monitor and the integrator. Similarly, we did not include the time to apply $\triangle V$ to the warehouse (after stop 9) because $t_{9,10}$ is specific to the database used as the warehouse. Neither of these times is determined by our integration techniques.

For some experiments, we study the system *throughput* — the number of updates processed in one unit of time (one second). For a group of B updates, if T is the elapsed time from when the first update starts at stop 1 to when the last update reaches stop 9, then the system throughput is B/T. Another performance criterion is the number of messages passed between modules during the propagation of one or more updates. The most interesting numbers of messages are the number of queries sent by the view manager to the query processor, and the number of all source wrapper queries. These two numbers are different for different views, for different types of view managers, and they affect UPT substantially.

In general, UPT and throughput are indications of the system efficiency, and the number and size of messages are indications of system load. Other performance criteria may include the view initialization time, the network traffic between the sources and the integration modules, source I/O due to view maintenance, and system bottlenecks. We will not discuss these criteria in detail but only point out that most of them are related to the UPT, throughput, or the number of messages between modules.

4.2 System

For all experiments presented in this paper, both the warehouse views and the source relations are stored in a Sybase database installed on a RS/6000 machine. The WHIPS integration modules run on a DEC Alpha 3000/300LX under OSF/1 3.0 with 125MH CPU, 64MB memory, and 128MB swap space. No other user processes were run on the system during the experiments. Source wrappers are on separate DEC Alpha machines with the same configuration. Any two machines are connected by 10BaseT Ethernet. As we have mentioned, most of the WHIPS system is implemented using C++ and ILU, a CORBA compliant object system developed by Xerox PARC. The entire system is about 19,000 lines of C++ code.

4.3 Views and base relations

View parameters that may affect the update propagation include the number of base relations, the sizes of base relations, selectivity (the fraction of tuples that satisfies the select condition(s)), project factor (the size of all projected attributes, which determines the tuple size of the view), and the join factor (the number of tuples in a relation that joins with one tuple in a neighboring relation). Given this diversity of factors, it is not easy to obtain a few "representative" scenarios. However, after some experimentation we chose the following "reasonable" views, representing views with different numbers of base relations and different result sizes.

$V_1 = \Pi_{A,B}(R), \qquad V_2 = \Pi_{R.A,R.B,S.B,S.C}(R \bowtie S),$

$V_3 = \Pi_{R.A,R.B,S.B,S.C,T.C,T.D}(R \bowtie S \bowtie T), \qquad V_4 = \Pi_{R.A,R.B,S.B,S.C,T.C,T.D,U.D,U.E}(R \bowtie S \bowtie T \bowtie U)$

In the view definitions, R, S, T, U are base relations, their schema is given in Appendix C. In the default setup, the cardinality of the base relations is set to 100 tuples. Each tuple in a relation is 100 bytes, and each projection attribute is 10 bytes. By default, we populate the tables such that one tuple of R joins with exactly one tuple of S, then joins with exactly one tuple of T. This is also true for new tuples; that is, if a new tuple is inserted into R, and this tuple is to be joined with S to compute $\triangle V$, we set the attribute value such that this new tuple will join with one tuple in S. This fixed join factor reduces variability and makes our results easier to interpret.

We use small relations mainly because there is an ILU upper bound on message size, which limits the sizes of the initial views, as well as of the query answers. To get around these limits we could have ported WHIPS to another CORBA implementation (e.g., Orbix has higher limits on message sizes). We could have also implemented a facility to break up large messages into "packets" to be sent in pipelined fashion. We did not consider this effort worthwhile for our evaluation because most of the performance results of interest are *not* affected by the base relation size:

1. We are studying the performance for *incremental* maintenance. Base relation cardinality plays no role in computing $\triangle V$; the more important factors are the join factors between relations.

2. We are focusing on the performance of WHIPS integration components. Larger base relations may increase the time to evaluate a source wrapper query if the relation is not properly indexed. However, this is just one feature of the data source that the integration modules may take into consideration.

3. We often want to compare the relative cost of different algorithms or design choices. For example, in Section 6.3 we are interested in comparing "complete consistency" strategies with "no consistency" ones. Absolute timings are not very relevant to this comparison.

4.4 Update Scenarios

In our performance experiments we use an *update simulator* to generate source updates according to specified parameters. The simulator updates the source data and sends the updates to the integrator. Thus, the simulator behaves like a WHIPS monitor. We consider the following two types of update propagation scenarios:

- Non-interleaving: The updates are separated by sufficiently long intervals, such that consecutive updates do not interfere with each other. In this scenario the UPT on an update is not affected by other updates.
- Interleaving: At least the propagation of one update U_k interferes with other updates. This interference may cause resource conflicts, (as well as additional compensation work, as described later). Thus, the UPT of U_k may be longer than in the non-interleaving scenario.

Unless otherwise specified, our default scenario is a non-interleaving one. Section 7 discusses finding the minimum interval between updates that eliminates interference.

4.5 Confidence of timing results and some basic timing results

To measure propagation times we use the average time $\bar{T} = (t_1 + t_2 + \ldots + t_P)/P$ from P repetitive experiments. In other words, each data point in the graphs that follow is an average number obtained from P repetitive experiments. For all the experiments in this paper, we have chosen a large enough P such that with a degree of confidence 0.99 we guarantee that the expected value of T is within $\bar{T} \pm 0.1\bar{T}$. Details on our choice of P and the evaluation of confidence intervals is given in Appendix A.

There are two basic timing results that are determined by the specific system configuration and are not affected by the view maintenance algorithms. The first one is the time to pass an asynchronous message between two modules. With the described configuration, a message containing 100 bytes of data takes about 0.004 seconds if the two modules are located on the same machine, and about 0.005 seconds if they are located on different machines. The second factor is the time to evaluate a source query, including the time to communicate with the source wrapper, execute the query, and get the results. When the result size is one tuple with 100 bytes of data, the time to execute a selection wrapper query with one selection condition takes about 0.028 seconds. This number remains roughly constant as the base relation ranges from 1 to 1000 tuples. We will study the effect of message size on message time further in Section 5.

5 Cost of modularity

The modular design of WHIPS inevitably introduces communication overhead. To understand this price of modularity, we measure the time spent in exchanging messages among the WHIPS integration modules. Remember that we defined the UPT of an update as $t_{1,9} = \sum_{j=1}^{j=8} t_{j,j+1}$. Among those times, $t_{2,3}$, $t_{4,5}$, $t_{6,7}$ and $t_{8,9}$ are communication times between WHIPS integration modules. These times are mainly determined by the size of the message. For example, $t_{2,3}$ is the time that the integrator takes to forward a source update to the view manager. It is roughly the same for all single-tuple updates, and it increases with the batch size if source updates are combined. Similarly, $t_{6,7}$ is the time the query processor takes to send a query answer to the view manager, and it is mainly determined by the answer size. Let $t_{commu} = t_{2,3} + t_{4,5} + t_{6,7} + t_{8,9}$, the communication overhead entailed by WHIPS's modular design.

Let $t_{computation} = t_{1,2} + t_{3,4} + t_{7,8}$. During these periods the WHIPS modules process data, e.g., decide the type of an update or check the correctness of a query answer. Let $t_{query} = t_{5,6}$, the time to process a distributed query. The value t_{query} may involve both communication (with source wrappers) and computation (collecting results). The length of t_{query} is determined by the number of wrapper queries, the network speed, and the time it takes sources to execute queries.

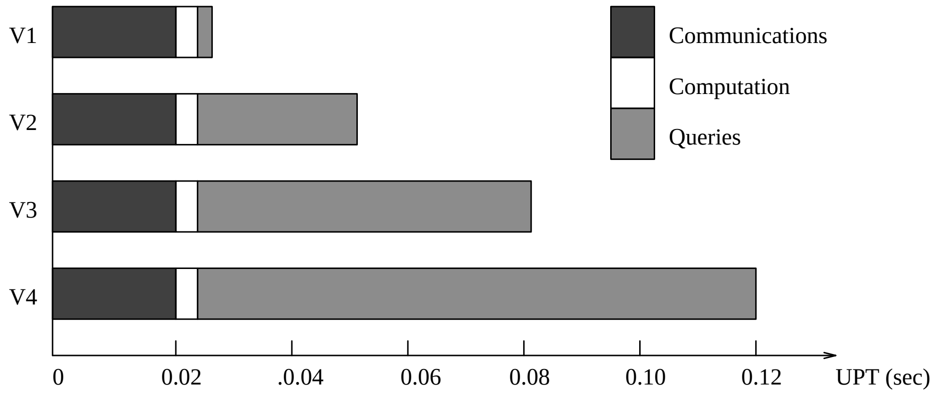


Figure 3: Update propagation time breakdown for different views

In Figure 3 we show the UPT breakdown for propagating a source insertion. The results we present here are obtained using a view manager of type CECA, but results for other view managers are almost identical when update propagations do not interleave (see Section 6.3). The final answer size is 1 for all four views in the default setup.

In the non-interleaving scenario, the UPTs for the four views given in Section 4.3 in the default set-up are 0.025 seconds, 0.053 seconds, 0.083 seconds and 0.120 seconds. As expected, for different views, the t_{commu} times are almost the same, because messages with the same type and almost the same size are exchanged. The $t_{computation}$ times are also similar for all views, and they are shorter than the communication and query times. For V_1, which contains only one relation, no wrapper query is necessary to propagate source updates. Therefore, communication time dominates, and t_{commu} is about 85% of the total UPT. As the number of base relations in the view increases, the query time quickly increases and becomes the dominant factor. For V_4, t_{commu} is about 17% of the UPT and t_{query} is about 80%. While the absolute communication time remains essentially constant, its fraction of the entire process decreases.

Figure 4 shows the update propagation time breakdown for different answer sizes of update queries. The results are for V_3 using algorithm CECA. As expected, in general the average UPT increases with the size of answer relation. This is due to the increase of both the communication time (shown as the shaded area in Figure 4) and query time, both of which include transferring the answer relation between modules.

For V_3, the inter-module communication time is about 1/3 to 1/2 of the entire UPT. Notice that message transmission time does not increase linearly with message size. For example, there is a significant jump in message times when the message size increases from 20 to 30 tuples. We believe this jump is caused by ILU changing its strategy for packaging messages.

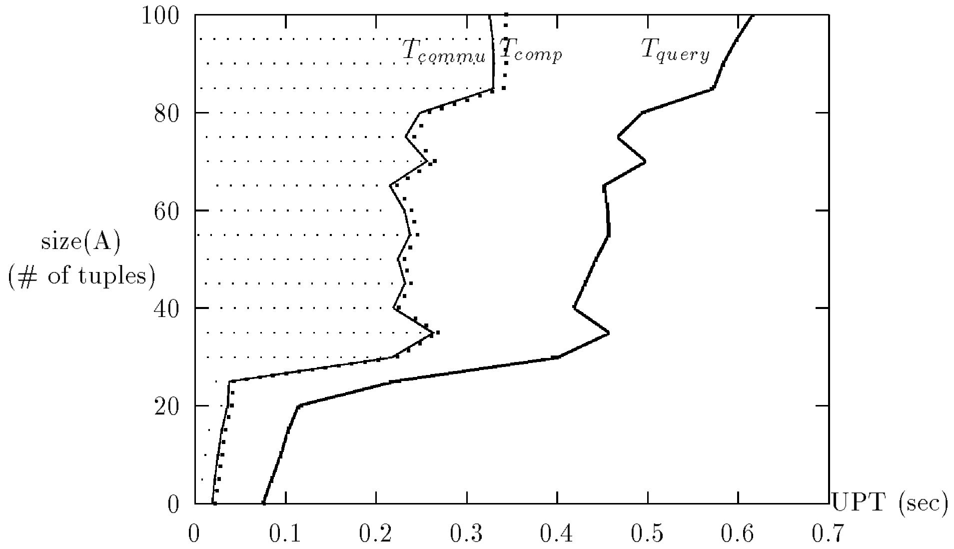


Figure 4: Update propagation time breakdown for different answer sizes

There are two striking features in Figures 3 and 4: query times can be very significant for some views, and the communication times are a significant overhead. Let us discuss these in turn.

Query times are high when the sources have to be contacted as part of the maintenance process. These times can be reduced by having faster sources, or by improving the query processor (e.g., through smart join ordering or condition filtering). Query times can be reduced even more dramatically by copying *all* source relations to the warehouse, avoiding going back to the sources entirely. (WHIPS can maintain copies of source data by defining views that are simply the same as source data.) In this case, queries are still performed, but entirely with warehouse data. The benefits of copying source data must be balanced with the costs: increased storage space and a larger number of updates that must be processed. The latter is because the copied source data must be maintained at the warehouse. That is, the more source data that is captured, the more updates to it that must be processed at the warehouse. The results in Figures 3 and 4 quantify some of the costs and potential savings involved, and hence can be useful for determining if it is worthwhile building a better query processor or copying more source data at the warehouse.

The communication costs can be reduced by using a more efficient CORBA implementation, or by coding the message passing directly. Another option is to combine all the integration modules, including the warehouse manager, into the same process, i.e., put all the integration modules close to the warehouse. With this approach, different integration modules may share a memory structure to exchange information.

Combining modules in a single process may reduce communication costs, but it makes harder to exploit parallelism. For instance, if we wished to have multiple view managers as in WHIPS, they would have to be implemented as concurrently executing threads within the single process, and we would have to be very careful with access to shared data structures. We believe that using separate modules is better for a testbed such as WHIPS, where ease of implementation is important, and that a combined approach would be more effective for a high performance system. This choice between tightly and lose integration of components is one faced by many system designers.

An option for reducing communication overhead while still maintaining modularity is to give each view manager its own query processor. Thus, each view manager and its instance of the query processor would share a process, and data exchanges between them would be efficient. This approach is especially beneficial when there are many queries from the view manager to the query processor, e.g., when updates frequently interleave and the view manager sends many compensating queries (Section 6.3). The main disadvantage is that it limits our ability to do load balancing for query processing.

In summary, if the efficiency of update propagation is critical, then a modular design based on CORBA is probably not be the best choice. If it is feasible to copy all source data to the warehouse, this can significantly reduce query times and should be considered. Of course, the types of views being maintained play an important role in these decisions, and the results shown in this section may help quantify the costs and benefits.

6 View maintenance algorithms

WHIPS implements six view maintenance algorithms, divided into two groups, as shown in Table 1. *Key algorithms* assume that the projection list (in the view definition) contains all key attributes from the base relations. *Duplicate algorithms* can maintain all Select-Project-Join views, with or without key information. WHIPS has available three view maintenance algorithms for each group, with different consistency levels.

	Key Algorithms	Duplicate Algorithms
Inconsistent	KeyConventional	Conventional
Strongly Consistent	Strobe	ECA
Complete	CStrobe	CECA

Table 1: Consistency levels of view maintenance algorithms

In Section 6.1 we illustrate view consistency levels through an example, and show how key algorithms and duplicate algorithms achieve consistency. In Section 6.2 we compare the performance of the two groups of algorithms. Then in Section 6.3 we compare algorithms that guarantee different levels of view consistency. Finally we discuss the coding complexity of all algorithms implemented.

6.1 View maintenance algorithms and view consistency

Recall that sometimes more information from the sources is needed to compute $\triangle V$, and WHIPS sends source wrapper queries to obtain that information. If there are concurrent updates at data sources while these wrapper queries are evaluated, conventional view maintenance algorithms such as those described in [BLT86, GMS93] may yield incorrect results. The following is an example:

Example 2: Inconsistent view maintenance

Consider the view $V = \Pi_{Name,Prize}(S \bowtie P)$ defined in Example 1 and the two initial relations depicted in Figure 2. Initially, the view contains two tuples $[Joe, CD]$ and $[Amy, book]$. This view content is consistent with the base data. Now, say Joe discontinues his subscription to Newsweek, at the same time that Newsweek cancels its promotional CD for its subscribers. There are two source updates: $U_s = delete(S, [Joe, \underline{Newsweek}])$ and $U_p = delete(P, [\underline{Newsweek}, CD])$. Corresponding to U_s, query $Q_s = \Pi_{Name,Prize}([Joe, \underline{Newsweek}] \bowtie P)$ is evaluated at the source of P. Now that the tuple containing Newsweek is removed from P, the result of Q_s is empty. The query corresponding to U_p, $Q_p = \Pi_{Name,Prize}(S \bowtie U_p)$ also returns an empty answer because currently no one is subscribing to Newsweek. The view will not be changed if one simply applies both results as suggested by conventional view maintenance algorithms. The final view (containing tuple $[Joe, CD]$) is inconsistent with the final source data. □

We have designed algorithms to avoid this type of inconsistency. As mentioned, some algorithms take advantage of base relation keys. For Example 2, if $Name$ is a key of $Subscription$, (i.e., each person subscribes to only one magazine), then for $U_s = delete(S, [Joe, \underline{Newsweek}])$, the view maintenance algorithm just needs to delete all tuples in the view with attribute $V.Name = Joe$. If $Prize$ is a key of $Promotion$ (i.e., each prize is offered by only one magazine), then for U_p, one may delete all tuples in the view with attribute $V.Prize = CD$. For views with duplicate semantics, *compensating queries* must be sent to correct the missing or redundant query results caused by update interleaving. In the above example, when U_p is processed, the maintenance algorithm may realize that update U_p will cause the result of Q_s to be incorrect. Then the query for U_p can be set to $Q_p = \Pi_{Name,Prize}(S \bowtie U_p) \cup \Pi_{Name,Prize}(U_s \bowtie U_p)$ ($\Pi_{Name,Prize}(U_s \bowtie U_p)$ is a compensating query), such that the result of Q_p will contain tuple $[Joe, CD]$. The final view is correct after this tuple is removed from the view.

We believe that for many applications, requiring base keys in view definitions is a reasonable assumption, since keys make it easier for the applications to interpret and handle the warehouse data. Furthermore, if a user-specified view does not contain sufficient key information, the warehouse can sometimes add the key attributes to the view and project them out when presenting the view content to the user. Nevertheless, it is not always appropriate to keep all base key attributes in a view, e.g., when base data use duplicate semantics, when the warehouse has no knowledge of base keys, or when the warehouse does not want to change user-defined views. In all these cases duplicate tuples or tuple counts need to be preserved in the views in order to incrementally maintain the view corresponding to source deletions [BLT86, GMS93, ZGMHW95].

A materialized view should reflect base data as it changes. In [ZGMHW95, ZGMW96] we formally defined different levels of materialized view consistency. Here we use a simple example to illustrate these levels informally. Let there be three source updates U_1, U_2, U_3, which occurred in that order at a data source. Let V be a related materialized view. A *complete* view maintenance algorithm will maintain V such that V reflects each of the U_is in the order they occurred. For the three updates in the example, the complete algorithm will update the view three times, each reflecting one update. A *strongly consistent* algorithm, on the other hand, may skip some source states but still guarantees eventual correctness. For example, it may update V to reflect U_1 and U_2 together, then reflect U_3. An *inconsistent* algorithm may maintain V such that its intermediate or final value is different from anything that could have been computed from source data. For example, it may finish all maintenance work and leave V in a state inconsistent with base data after U_3. As we saw in Example 1, a conventional (centralized) view maintenance algorithm is inconsistent if directly applied to the warehousing environment. We implemented the conventional algorithms to compare their performance with that of other algorithms.

6.2 Keeping keys in a view

Now we compare the performance of key and duplicate algorithms. We specifically compare key algorithm Strobe with duplicate algorithm ECA. In the non-interleaving scenario, the other two key algorithms behave the same as Strobe, and the other two duplicate algorithms behave the same as ECA (Section 6.3). Figure 5 shows the UPT of an update on base relation R.

As we see from the results, for the same view, the UPTs for insertions are similar for both algorithms. This is because they both need to send source queries and obtain answers. The UPT using Strobe is sometimes slightly shorter because Strobe uses a simpler data structure. For deletions, the UPTs are quite different. Duplicate algorithm ECA treats deletions exactly the same as insertions. Strobe, on the other hand, always forwards the deleted key attributes directly to the warehouse and lets the warehouse manager apply a delete action. Therefore for Strobe, the UPTs of all deletions are the same, regardless of the view definition, and they are shorter than those for insertions.

Obviously, the benefit of using a key algorithm is the efficient processing of deletions. The higher percentage of deletions in the source update streams, or the longer it takes to propagate insertions (e.g., when the view contains many base relations), the more beneficial it is to use a key algorithm.

When source updates include deletions, maintaining a view using key algorithms also incurs fewer messages between the VM and the QP, and thus generates fewer wrapper queries and less work at the source.

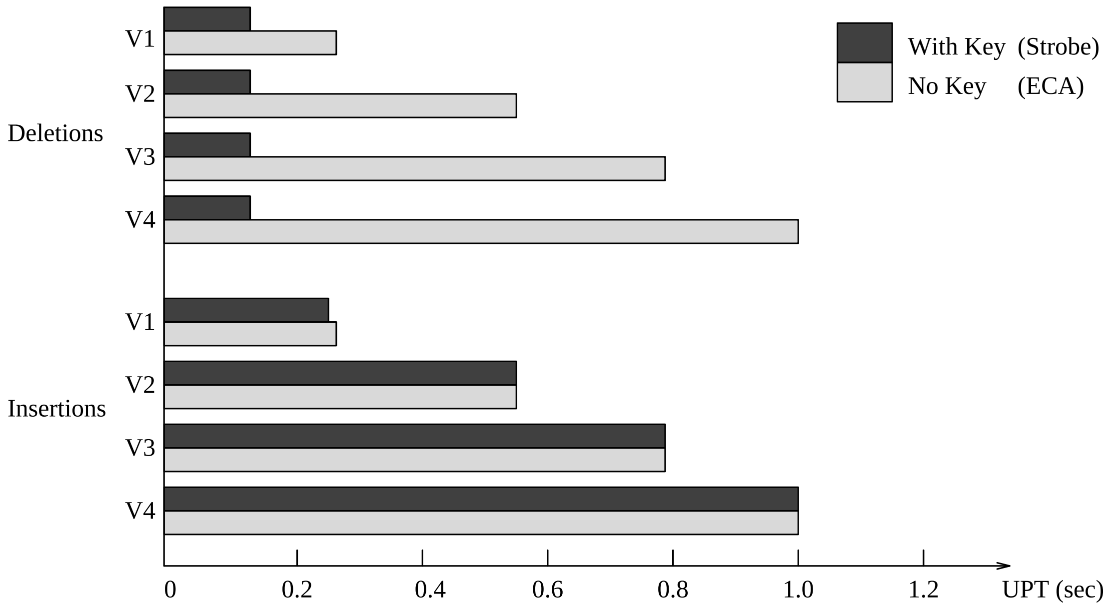


Figure 5: Update propagation time for algorithms with and without key information

To propagate a total of K updates including K_1 insertions, there are K update queries sent from the VM to the QP for duplicate algorithms and K_1 for key algorithms.

It is important to note that the two groups of algorithms maintain views using different operations. Key algorithms maintain their views by applying *actions* to them. An *action* either inserts tuples or deletes tuples that satisfy certain conditions. For example, an action can be "Delete from view V_2 where $A = 2$." On the other hand, duplicate algorithms maintain their views by applying *plus deltas* and *minus deltas* to them. That is, before maintaining a view, the tuples to be inserted into the view are stored in a temporary "plus table," and the tuples to be deleted from the view are stored in a "minus table." When the view is ready to be maintained, tuples in the plus table are inserted into the view and tuples in the minus table are deleted from the view, using duplicate operations.

Whether it is more efficient to use actions or delta tables depends largely on whether the underlying database supports duplicate semantics, and on how many actions/deltas can be combined and applied to the view in a batch. For example, performing a delete action for which the key attribute value is given may be very efficient (e.g., given an index over that key). On the other hand, deleting one tuple from a view containing duplicates may require specialized procedures when such deletion is not supported by the underlying database (e.g., Sybase). If there is a list of actions to be applied, they have to be applied in the order given, while a list of delta tables may be combined into a single pair of plus and minus tables and applied to the view once.

In the rest of this paper, we focus mainly on duplicate view managers under insertions. This is because performance for the other view managers can be inferred from these results. In particular, for duplicate algorithms, UPTs are the same for deletions and insertions. For a key algorithm, insertion UPTs are almost the same as those for the duplicate algorithm with the same consistency level. Deletion UPTs for key algorithms are all the same, and are as shown in Figure 5.

6.3 View consistency

Now we study the cost and complexity of achieving different levels of view consistency. We performed experiments under both the non-interleaving scenario and a predictable interleaving scenario. The average UPT results are shown in Figure 6.

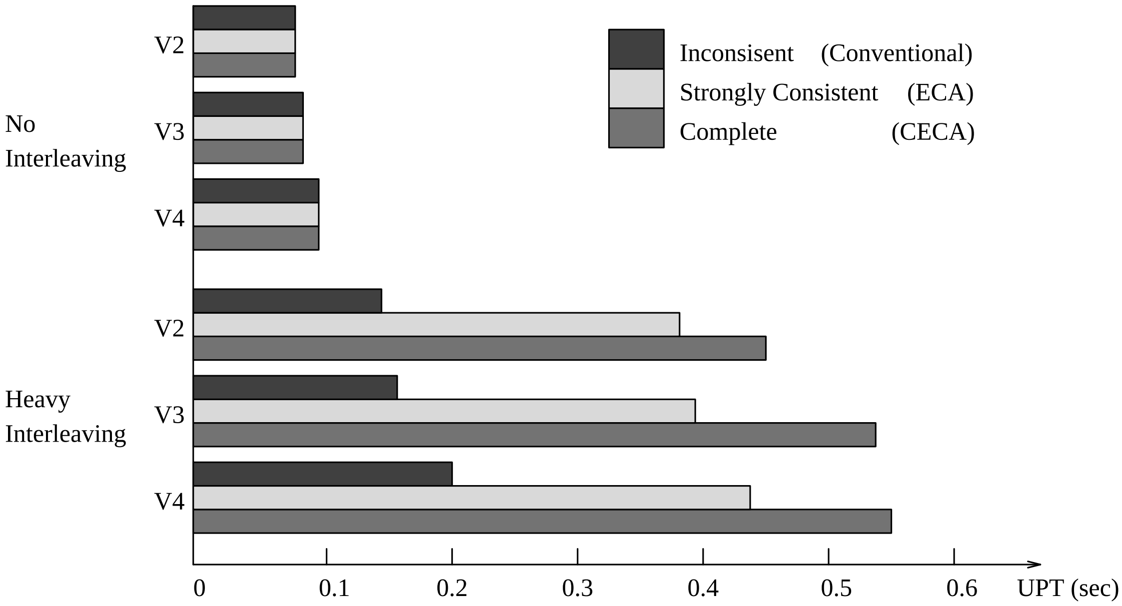


Figure 6: Update propagation time for views maintained to different consistency levels

When there is no source update interleaving, all three types of algorithms maintain materialized views correctly. The average UPTs for the three algorithms are almost the same. This is as expected since all view managers should send and receive the same messages with the same contents. Between stop 7 and 8, the strong and complete view managers execute more CPU operations due to their more complex logic, but the effect is insignificant. In this scenario, all three algorithms also reflect each single source update in the view.

In the interleaving scenario, two tuples are simultaneously inserted, one into base relation R and the other into S, and reported to the integrator one after the other. Let these two updates be U_r and U_s. We also make U_r join with U_s and join with one existing T tuple. Update U_s occurs before the propagation of U_r finishes, i.e., before the update query corresponding to U_r is evaluated. Thus U_s interferes with the propagation of

U_r. We measure the average UPT of this group of two updates and repeat the experiment enough times to satisfy the specified confidence level. We stress that this interleaving scenario is a pessimistic one because every single update observes interference. In Section 7 we study a scenario with stochastic interference.

For the interleaving scenario, the Conventional algorithm may lead to incorrect view maintenance. The UPT for the Conventional algorithm is a lower bound since that algorithm logically processes the updates as if there were no conflicts among them. The UPT increase for the Conventional algorithm between the no-interleaving and interleaving scenarios is due to resource contention; we discuss this further in the following section.

The strong and complete algorithms will send compensating queries to correct potential inconsistencies. Specifically, a compensating query of the form $U_r \bowtie U_s \bowtie T$ is sent to the QP for the two updates U_r and U_s. Since evaluating compensating queries requires communication between the VM and QP, and possibly between the QP and source wrappers, the impact of compensating queries on UPT is substantial. For example, for V_3, the UPT using the ECA algorithm is about 2.6 times longer than the Conventional UPT, and that using CECA algorithm is about 3.0 times longer than the Conventional UPT. Complete algorithms take the longest time to propagate updates because updates must be processed one by one, while strongly consistent algorithms may combine updates. The increase of the average UPT for complete and strongly consistent view managers is the cost of maintaining data consistency.

Now we look at the effect of interleaving on the number of messages exchanged. Let the total number of updates be K, where K is an even number. Let the number of messages exchanged by modules m_1 and m_2 be $msg(m_1, m_2)$. For the non-interleaving scenario, we have $msg(VM, QP) = K$, $msg(QP, WR) = 2K$ (WR represents all source wrappers) and $msg(VM, WH) = K$ for all three algorithms. For the interleaving scenario, every pair of updates triggers two update queries and one compensating query, thus $msg(VM, QP) = 3K/2$. Each update query invokes two source wrapper queries, and each compensating query invokes one wrapper query, so the total number of wrapper queries is $msg(QP, WR) = 2 * K + 1 * K/2 = 5K/2$. For the ECA view manager, every two updates are bundled together and invoke only one view-update message, so $msg(VM, WH) = K/2$. For the CECA view manager, each update invokes one view update message, so $msg(VM, WH) = K$. The number of messages is summarized in the following table.

	$msg(VM, QP)$	$msg(QP, WR)$	$msg(VM, WH)$
All three algorithms (non-interleaving) and Conventional (interleaving)	K	2K	K
ECA (interleaving)	3K/2	5K/2	K/2
CECA (interleaving)	3K/2	5K/2	K

As we can see, the increase in number of messages is not dramatic. The total message overhead goes from $4K$ in the non-interleaved case, to at most $5K$ with interleaving. Thus, the main costs of guaranteeing consistency when updates interleave are delays and the overhead of additional source queries.

6.4 Implementation complexity

The following table shows the number of lines of code for the view manager algorithms, excluding comments and common code (e.g., initialization). As we can see, key algorithms are slightly less complex to implement; for example, the implementation of Strobe view manager is about 70 lines of code shorter than the ECA view manager. Within the same group, the two complete view managers are the most complex, as expected, followed by the strongly consistent view managers, followed by the Conventional view managers.

View Managers	Lines of Code	View Managers	Lines of Code
vmCStrobe.cc	468	vmCECA.cc	479
vmStrobe.cc	311	vmECA.cc	381
vmKeyConv.cc	201	vmConv.cc	190

7 Source update scenarios

Interleaving updates impact the system in two ways:

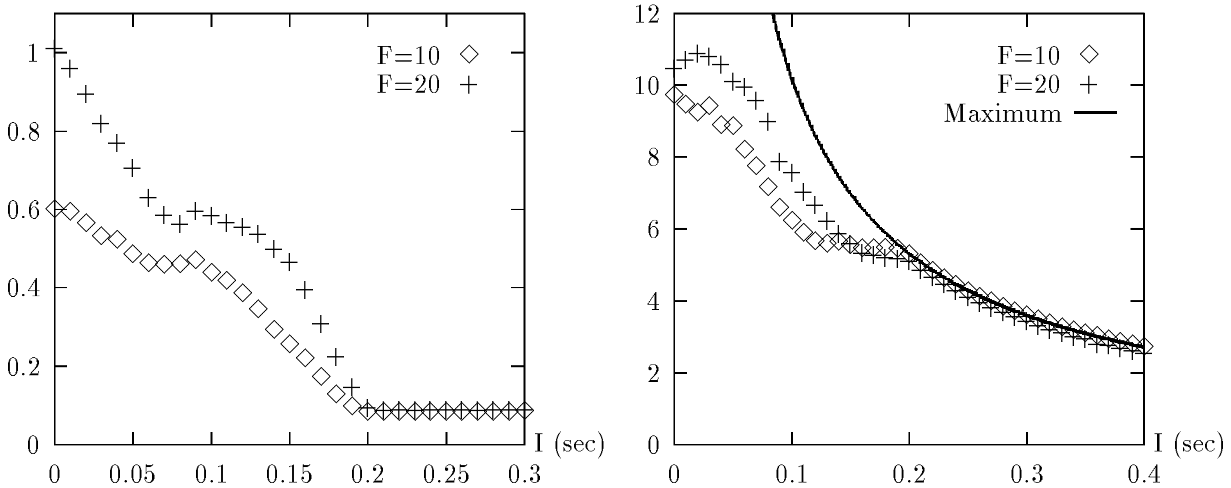
1. Resource contention. A message for one update may be blocked if the module it tries to reach is busy processing another update.
2. Generation of extra work. Some of the algorithms, such as ECA or CECA, compensate for interleaving by sending compensating queries and processing the results of those queries. This creates additional work.

In Section 6.3 we focused on the delays caused by the extra work generated by compensating algorithms. In this section we first study the effect of resource contention on UPT and system throughput. We then discuss one experiment where both contention and extra work affect system performance.

7.1 Fixed update intervals

For each test in this experiment, a sequence of F updates on the same base relation are sent to the integrator, separated by a fixed interval I. The average UPT and the system throughput are then measured. For different tests, a different time interval is used to separate updates. Because all the updates are on the same base relation, there is little extra work generated by the algorithms, and most of the delays are caused by contention, i.e., queuing of messages. Figure 7 shows the average UPT and throughput for view V_3 using CECA algorithm for $F = 10$ and $F = 20$.

As expected, when the update frequency is the highest (update interval I is 0), we have the longest average UPT due to update interference. We also have the highest throughput, meaning that the system is kept busy and tries to process as many updates as possible. Therefore, to maximize system throughput, it is better to propagate updates as fast as they become available, regardless of the delay caused by propagation



(a) Update Propagation Time (seconds)

(b) Throughput (# updates per second)

Figure 7: UPT and throughput for updates with different update intervals

interference. The average UPT decreases as the update interval grows, and once the time interval is greater than a certain threshold, the average UPT stays almost constant. Let this time interval be I_b. Intuitively, the propagations of consecutive updates do not interfere with each other if they are separated by a time interval larger than I_b. Let the UPT with I_b intervals be T_b. For view V_3, we have $I_b \approx 0.2$ seconds and $T_b \approx 0.08$ seconds.

In Figure 7(b), the curve "MAX" shows the maximum possible throughput for updates sent in the specified intervals, which is roughly equal to $1/I$. As expected, when $I > I_b$, all updates are propagated without any delay, and the throughput reaches maximum. When $I < I_b$, because there are delays due to propagation interference, the throughput is less than the maximum possible. We can see that when $I < I_b$, the throughput for $F = 20$ is greater than that for $F = 10$, but the average UPT is also longer.

Keep in mind that each data point in Figure 7 represents an average over many executions. Even though our inputs have a very regular pattern, the timing of each execution can vary significantly, depending on how and when messages are queued. Figure 8 shows a few different scenarios of two consecutive update propagations when $I = 0$.

In Figure 8, a dot with number n means that the corresponding update reaches stop n. The time axis shows the relative time between events. For example, since time 9 of U_5 comes before time 3 of U_6, we know that update U_6 did not reach the view manager (time 3 of U_6) before U_5 reaches the warehouse(time 9 of U_5). Fortunately, often the behavior of a group of updates in a given scenario can be roughly predicted, sometimes, the best and worst case performance can be predicted.

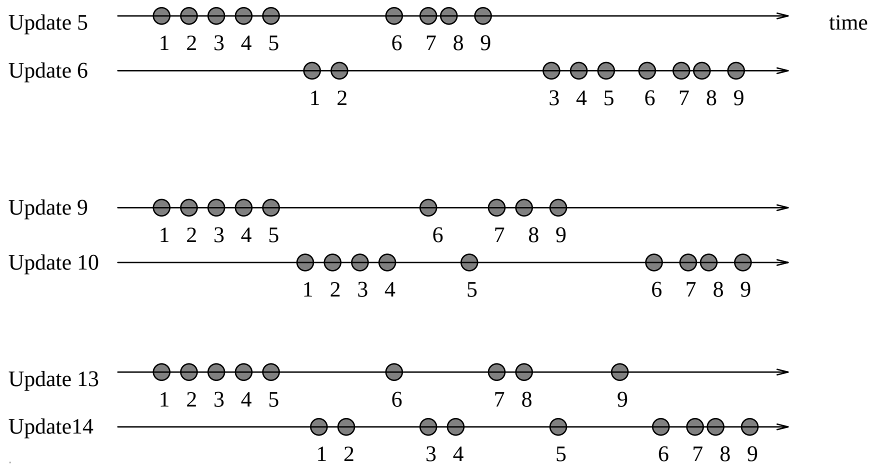
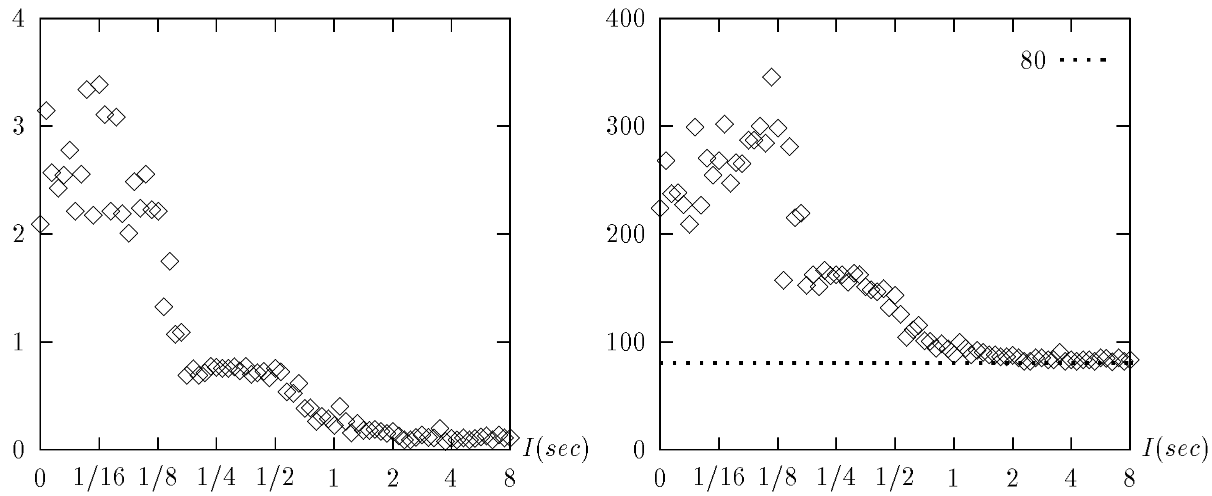


Figure 8: Overlapping update propagations

7.2 Random update intervals

So far, we have studied regular update patterns that let us understand the main tradeoffs in incremental update processing. Next we perform a series of experiments with randomly-generated source updates. These experiments provide a more realistic scenario and a better idea of how WHIPS may operate in practice.

Figure 9 shows performance results for view V_3. We concurrently generate 40 random insertions to base relation R and 40 random insertions into relation S. Each new tuple has about a 1/5 chance to join with an existing tuple in a neighboring relation. The update interval is a random number between 0 and I seconds.



(a) Update Propagation Time (seconds)

(b) Number of queries from VM to QP

Figure 9: Performance results for propagations with random update intervals

Figure 9 shows the results for the CECA algorithm. Each data point represents one run where the 80 updates are processed. Figure 9(a) shows the average UPT. The general shape of the resulting curve is similar to that in the fixed-interval scenario. When the update interval is relatively long, there is little interference and the UPT is roughly constant ($\approx 0.08sec$). The average UPT increases as the update frequency grows. When I is small, the UPT is quite variable, since the precise ways in which the updates interfere is very unpredictable.

Notice that the point at which updates effectively stop interfering ($I > 1$) is different from the previous experiment (where I_b was 0.2 seconds). This is partly because now updates are separated by times in the interval 0 to I (and not all separated by I.) Another reason is that in the current experiment updates are delayed both because of resource contention and because of compensating queries. To illustrate this, Figure 9(b) shows the number of queries sent from the VM to the QP. Of all the queries, 80 are update queries, and the rest are compensating queries. We can see that the query number increases almost directly with the increase in average UPT. There are many compensating queries when the update frequency is high; the actual number depends on how updates interfere with each other.

We also noticed that there is no significant increase in the number of actual wrapper queries, as shown in Figure 10. Because the chance of two updates U_r and U_s joining is small (about 1/200), most of the compensating queries of the form $U_r \bowtie U_s \bowtie T$ result in an empty answer and can be evaluated by the QP locally. Thus, we see a good optimization opportunity for the heavy interleaving scenario: the view manager may screen compensating queries before sending them to the QP. If a query will clearly yield an empty answer (as is the case with $U_r \bowtie U_s \bowtie T$ when U_r and U_s do not join), the view manager can skip it and reduce the compensation load.

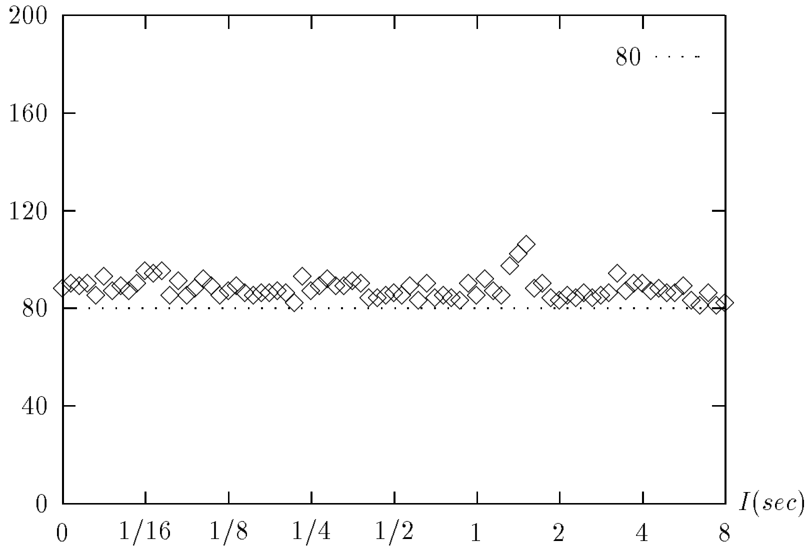


Figure 10: Number of queries from the QP to the source wrappers

Finally, we want to mention that the behavior of view manager using strongly consistent ECA algorithm is different from that of CECA, because ECA may combine the view deltas corresponding to multiple updates into one single message to the warehouse. From Figure 11, we can see that the number of view deltas sent from the view manager to the warehouse decreases with the update interval. This is because with high update frequency, there are more chances that the propagation of consecutive updates interfere with each other, and the corresponding view deltas are bundled together. This decrease of messages to the warehouse also synchronizes with the increase of UPT, since when view deltas are bundled together, the corresponding updates wait at the view manager before being propagated to the warehouse. When the update interval is less than 1/16 seconds, most of the view deltas are bundled and there are only 2 or 3 messages to the warehouse for all 80 updates.

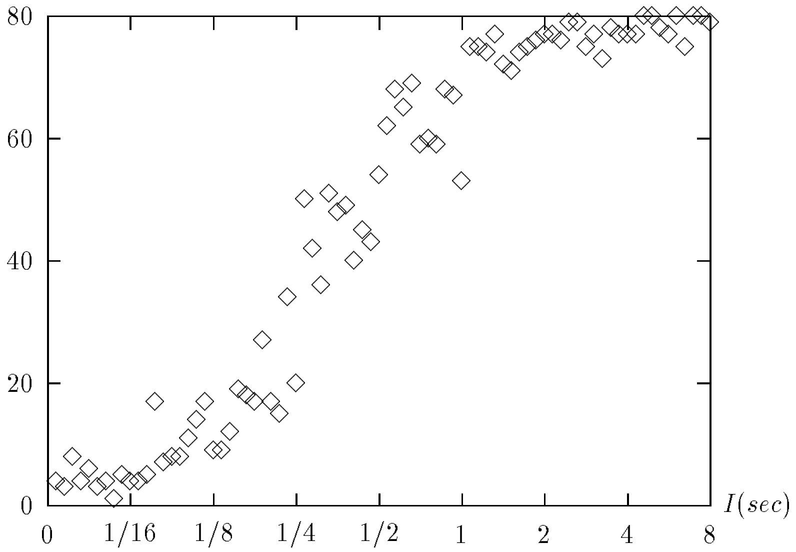


Figure 11: Number of messages from the VM to the warehouse

One disadvantage of strongly consistent algorithms is that when update frequency is high and updates are continuously generated without long enough halting time between them, the algorithms may keep accumulating updates and never send view deltas to the warehouse. Theoretically, when the source update interval is smaller than I_b, all updates may be bundled together. In reality, the update frequency may not ALWAYS be very high, thus most of the time, some but not all updates are bundled. Batching should be considered if update frequency is always high.

8 Batching

Source updates can be batched and propagated to the warehouse together. In general, batching updates not only reduces the number of messages, but it also reduces the chance of interference between updates. However, batching may cause updates to wait before being propagated, increasing their UPT. Batching can be done at various steps along the update propagation path, i.e., at the source monitor, the integrator, or the view manager. In this section, we assume that source monitors collect updates and send them together to the integrator, according to one of the following batching policies.

1. Batch every B updates, where the batch size $B \geq 1$. The $B = 1$ scenario has been discussed in previous sections, where each update is propagated individually. When $B > 1$, some updates may wait at the source monitors to be batched with later updates, and this waiting times would be considered part of the propagating time. As we will find out later in this section, for certain update scenarios, there is an optimal batch size B_o.

2. Batch all updates generated in certain time interval, e.g., in a minute or in an hour. This policy is useful if it is only possible to propagate updates during certain time windows, so as to not interfere with other activities (e.g., user queries). At the beginning of an update window, it is possible to break up a batch into smaller ones.

3. Batch when the "oldest" update has waited a for certain time. This method may ensure that a view does not become too "stale." It can also be used in combination with Policy 1 to make sure all updates are propagated to the warehouse after waiting for a certain time.

In this section, we will discuss only Policy 1 involving the optimal batch size. Once we have determined the optimal batch size for Case 1, its value may be used in Policies 2 and 3. For instance, in Policy 3 we may wait to collect the optimal number of updates, unless the first update is delayed too long.

There are two major goals for batching. The first is to reduce the workload. Batching reduces the number of messages, so it is most useful when messages are expensive. Batching may also reduce the number of operations done by the integration module. For example, say we need to compute $\triangle V = \triangle R \bowtie S$ where there are n updates in $\triangle R$. If the n updates are processed individually, then n joins to relation S are needed. If n updates are batched together into one message and the S wrapper can perform joins, then only one join is necessary to compute $\triangle V$. In general, the larger the batch size, the more the workload is reduced. Of course, considering view freshness requirements and message size limitations, there is a practical limit on the largest reasonable B.

The second major goal of batching is to speed update propagation by reducing the interference among update propagations. The following experiment shows how batching reduces the average UPT compared with individual updates. It also examines the optimal batch size under certain update scenarios. We study view $V_2 = R \bowtie S$ using algorithm CECA. A total of K updates on base relation R are grouped into batches of size B, and sent to the integrator with a fixed interval I between batches. Figure 12 shows the average UPT results for $K = 20$ and $B = 1, 2, 5, 10$. As an example, when $B = 2$, there are 10 batches of updates, each containing 2 tuples. These 10 batches are sent to the integrator separated by interval I.

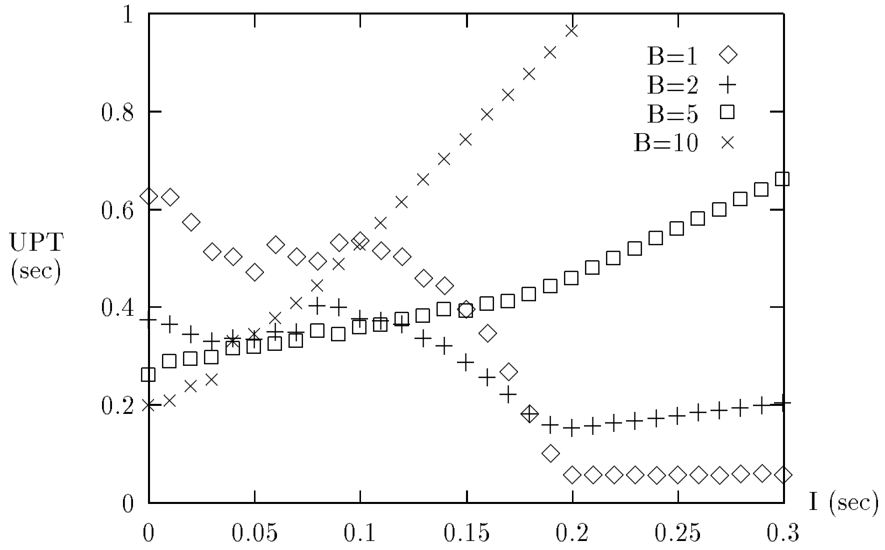
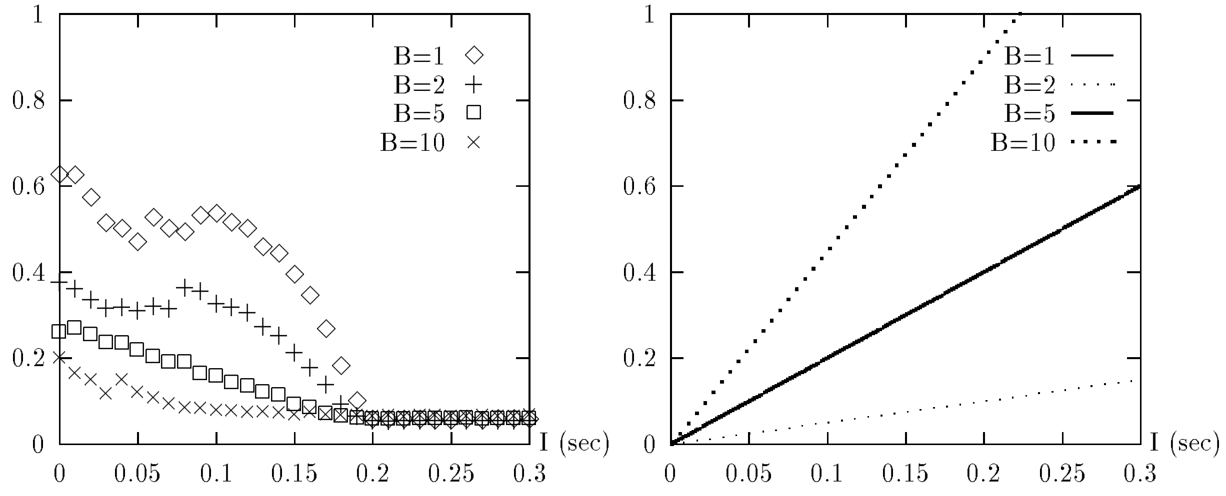


Figure 12: Average UPT for different batch sizes

Figure 12 shows that each batch size is optimal for some update frequency. In general, when the update frequency is high (I is small), updates bundled in larger batches are propagated faster. When the update frequency is low, smaller batches are better. When $I \geq I_b$, sending updates individually (B=1) is the fastest way to propagate them.



(a) Actual update propagation time (sec)

(b) Wait time (sec)

Figure 13: Breakdown of average UPT for batching

The average UPT shown in Figure 12 can be broken into two components: the actual update propagation time and the wait time. (The latter is the average time updates wait at the source before being batched and sent to the integrator.) These two components are shown separately in Figure 13(a) and Figure 13(b).

For the actual update propagation time only, when $I < I_b$, a larger batch size is always better, since there are fewer messages and less contention. When $I > I_b$, the actual propagation time for all B from 1 to 10 is similar. For the wait time, the source waits until B updates are available before sending them together, giving the average wait time for batch size B as $I * (1 + 2 + \ldots + (B - 1))/B = (B - 1) * I/2$. The wait time increases linearly with the update interval as well as with B. So when I is long, a larger B causes updates to wait longer.

When we consider both the actual propagation time and wait time, the former dominates when I is short, and a larger batch size gives shorter UPT. One special case is when $I = 0$ (all updates are ready to be propagated). This is the case when updates are generated by the same source transaction or updates accumulate during non-propagation times. Batching them in the largest possible batch size is often the best. In addition, one needs to consider the non-linear increase of message passing time pointed out in Section 5. Thus, it may be advisable to keep the batch size limited, so that message sizes are below those that cause a jump in transmission time. When I is long, wait time becomes the dominant factor and smaller batches may perform better. In particular, when $I \geq I_b$, sending the updates individually is the most efficient. In reality, optimal batch size or batch window may be determined according to the expected frequency of updates.

9 Conclusion

In this paper we studied the performance of WHIPS, an incremental warehouse maintenance testbed. Performance depends on a variety of factors, including hardware parameters, the base relations and views defined, the update frequency and batch size, relation selectivities and join factors, and the maintenance algorithm used. It is thus hard to make particular performance predictions. Nevertheless, we believe that our experiments shed some light on the major design tradeoffs faced in incremental maintenance. In particular,

- We evaluated the costs of a modular system design, by studying update propagation and quantifying the cost of using CORBA as an infrastructure (Section 5). We concluded that CORBA is quite expensive, but offers great design and implementation flexibility.

- We evaluated the delay caused by sending queries to data sources (Section 5). The results can help us decide whether to copy base data at the warehouse, and how beneficial it may be to optimize global query evaluation.

- We analyzed the behavior of WHIPS under a spectrum of source update scenarios, quantifying the impact of update interleaving (Section 7).

- We compared the relative performance of different view maintenance algorithms and studied the cost of maintaining view consistency (Section 6).

- We presented results that provide insights to system designers. For example, we discussed how to determine optimal batch size (Section 8), how to send updates (Section 7, 8), which query processor optimizations are crucial (Section 5, 6, 7), and what is a superior method for evaluating compensating queries (Section 6, 7).

References

[BLT86] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently updating materialized views. In *SIGMOD*, pages 61–71, Washington, D.C., June 1986.

[CJS+94] A. Courtney, W. Janssen, D. Severson, M. Spreitzer, and F. Wymore. Inter-language unification, release 1.5. Technical Report ISTL-CSA-94-01-01 (Xerox accession number P94-00058, Xerox PARC, May 1994.

[Fre71] John E. Freund. *Mathematical Statistics.* Prentice Hall, Englewood Cliffs, NJ, 1971.

[GL95] T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In *SIGMOD*, pages 328–339, San Jose, California, May 1995.

[GM95] A. Gupta and I.S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–18, June 1995.

[GMS93] A. Gupta, I. Mumick, and V. Subrahmanian. Maintaining views incrementally. In *SIGMOD*, pages 157–166, Washington, D.C., May 1993.

[Han87] E.N. Hanson. A performance analysis of view materialization strategies. In *SIGMOD*, pages 440–453, San Francisco, California, 1987.

[HZ96a] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *SIGMOD*, pages 481–492, Montreal, Canada, June 1996.

[HZ96b] R. Hull and G. Zhou. Towards the study of performance trade-offs between materialized and virtual integrated views. In *Workshop on Materialized Views*, pages 91–102, June 1996.

[LGM96] W. Labio and H. Garcia-Molina. Efficient snapshot differential algorithms in data warehousing. In *VLDB*, pages 63–74, September 1996.

[LW95] D. Lomet and J. Widom, editors. *Special Issue on Materialized Views and Data Warehousing,* IEEE Data Engineering Bulletin 18(2), June 1995.

[OV91] M.T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems.* Prentice Hall, Englewood Cliffs, NJ, 1991.

[SC97] U. Dayal S. Chaudhuri. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[WGL+96] J.L. Wiener, H. Gupta, W.J. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A system prototype for warehouse view maintenance. In *Workshop on Materialized Views*, pages 26–33, Montreal, Canada, June 1996.

[ZGM98] Y. Zhuge and H. Garcia-Molina. Performance analysis of WHIPS. Technical report, Stanford University, May 1998. http://www-db.stanford.edu/pub/papers/zg-whipsperf.ps.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *SIGMOD*, pages 316–327, San Jose, California, May 1995.

[ZGMW96] Y. Zhuge, H. Garcia-Molina, and J.L. Wiener. The Strobe algorithms for multi-source warehouse consistency. In *PDIS*, pages 146–157, Miami Beach, Florida, December 1996.

[Zhu97] Y. Zhuge. Whips performance: model and experiments. Technical report, Stanford University, December 1997. Available via anonymous ftp from host `www-db.stanford.edu` as `pub/papers/zg-perftech.ps`.

[ZWGM97] Y. Zhuge, J. L. Wiener, and H. Garcia-Molina. Multiple view consistency for data warehousing. In *ICDE*, pages 289–300, Birmingham, UK, April 1997.

A Confidence of performance timing results

When we measure any timing result, we use the average time $\bar{T} = (t_1 + t_2 + \ldots + t_P)/P$ from P repetitive experiments. In other words, one data point in the later sections, such as one point in Figure 7, is an average number obtained from P repetitive experiments. For all the experiments in this paper, we choose large enough P such that with a degree of confidence 0.99 we guarantee that the expected value of T is within $\bar{T} \pm 0.1\bar{T}$. That is, the possibility of the T falls within the interval $\bar{T} \pm 0.1\bar{T}$ is higher than 0.99.

Now we discuss how to test whether the mean of timing results satisfies the specified confidence interval. Let μ be the estimated value of the mean $\bar{T}$, and $s^2 = \frac{1}{n-1}\sum_{i=1}^{P}(T_i - \bar{T})^2$ be the sample variance. Assume that random variable T is from normal distribution, then we can obtain the $1-\alpha$ confidence interval for μ using the following formula, with t as the $t-$distribution [Fre71]:

$$\bar{T} - t_{\alpha/2,P-1}\frac{s}{\sqrt{P}} < \mu < \bar{T} + t_{\alpha/2,P-1}\frac{s}{\sqrt{P}}$$

We want the interval between the max and min value to be less than 10% of the mean:

$$2 * t_{\alpha/2,P-1}\frac{s}{\sqrt{P}} < 0.2 * \bar{T}$$

For $\alpha = 0.05$ and sample size $P \geq 20$, we have

$$2 * t_{\alpha/2,P-1}\frac{s}{\sqrt{P}} < 2 * t_{0.005,19}\frac{s}{\sqrt{P}} = 2 * 2.861 * \frac{s}{\sqrt{P}}$$

That is,

$$2 * 2.861 * \frac{s}{\sqrt{P}} < 0.2 * \bar{T}$$

or

$$28.61^2 s^2 < P\bar{T}^2 \tag{A.1}$$

We use equation A.1 to test all timing results in previous sections, except for the results of the updates with random intervals (Section 7.2).

The required accuracy can often be achieved with averaging over a few tens of update propagations for the none-interleaving scenarios, and a few hundreds for the interleaving scenarios, Although theoretically a larger P gives smaller errors, it may not always yield better answer in reality because the experiment results are also sensitive to system and network load, which may change over time. For example, the refresh time depends on network load since the data sources are located on a different machine from where the the query processor is. It is more likely that the network load fluctuate over a long period of time, and resulting in larger variances of refresh time and less accurate results.

B Analytic results of WHIPS performance

In this appendix we present some analytic results of WHIPS performance. Especially, we estimate the number of source wrapper queries and answer size, which are the two most important factors that affect UPT. We also estimate the number of compensating queries for interleaving scenarios. The results shown here are for single tuple updates. For batched updates, depending on whether the query processor processes a set of tuples together or one by one, a batch factor B may be multiplied to the estimated results.

B.1 Estimating the number of source wrapper queries and answer size

There are two types of queries sent from a view manager (VM) to the query processor (QP): update queries and compensating queries. We now estimate the number of wrapper queries that the QP need to sent in order to evaluate these two types of queries.

For an SPJ view defined by $V = \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_N))$, corresponding to a source update U on relation r_i, the *update query* that the view manager sent to the query processor is $UQ = V\langle U \rangle = \Pi_{proj}(\sigma_{cond}(r_1 \bowtie r_2 \bowtie \ldots \bowtie r_{i-1} \bowtie U \bowtie r_{i+1} \ldots \bowtie r_N))$. There are $N-1$ base relations in this query.

In WHIPS, we assume that each base relation has its own wrapper. The query processor processes query UQ using a nested-loop-join method in the following manner. It first applies the select condition of r_i (if any) to tuple U. Then, it picks a base relation r_j that joins with r_i (it picks a random one if relation joins with r_i), translates the join conditions into selections using attribute values in U, then sends to the wrapper of r_j these select conditions together with any select condition on r_j obtained from the *cond* in view definition. When the answer relation from the wrapper of r_j is received, the QP treats each tuple in the answer relation the same as a known tuple U, and joins them with the next relation. This process repeats until all base relations are processed or until the answer relation is empty after a selection or a join.

Now for a query UQ, let us rename the base relations according to the join order, such that r_i is renamed to be R_1, and the rest relations are renamed as $R_2, R_3, \ldots, R_N$ according to the order of join processing done by the QP. Now $UQ = \Pi_{proj}(\sigma_{cond}(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_N))$.

Next we define precisely a few parameters used in the following analysis. The select condition *cond* in the view definition can be break into conditions on each of the base relation. Let the select factor $\sigma_i (1 \le i \le N, 0 \le \sigma_i \le 1)$ of a base relation R_i to be the percentage of tuples in R_i that satisfies the select condition. $\sigma_i = 0$ means no tuple in R_i satisfies the condition, and $\sigma_i = 1$ if there is no select condition on R_i.

Let A^1 be the answer relation after σ_1 is applied to U, and let A^k be the answer relation after the selection on relation R_k and the join with R_k is done. A^N is the query answer for UQ. Define the join factor $J_i (2 \le i \le N)$ of relation R_i to be the estimated number of tuples in R_i that joins with one tuple in A^{i-1}.

Now we can estimate the total number of wrapper queries in order to evaluate UQ. To compute A^k, one wrapper query is needed for *each tuple* in A^{k-1}. So the number of queries to compute A^k is $card(A^{k-1})$. Since $A^k = A^{k-1} \bowtie \sigma_k(R_k)$, we have $card(A^k) = \sigma_k * J_k * card(A^{k-1})$ and $card(A^1) = \sigma_1$. From this we obtain $card(A^k) = \Pi_{i=2}^{k}\sigma_i * J_i * card(A^1) = \sigma_1 \Pi_{i=2}^{k}\sigma_i J_i$

During the step by step evaluation of UQ, the number of wrapper queries as well as the answer size is shown in the following table. The last column of the table shows $card(A^i)$ when assuming $\sigma = \sigma_1 = \ldots = \sigma_N$ and all $J = J_2 = \ldots = J_N$, where we refer as the uniform scenario.

Step	Query	# of queries	$card(A^i)$	$card(A^i)$ (uniform)
1.	$A^1 = \sigma_1(U)$	0	σ_1	σ
2.	$A^2 = A^1 \bowtie \sigma_2(R_2)$	σ_1	$\sigma_1 * \sigma_2 * J_2$	$\sigma^2 * J$
3.	$A^3 = A^2 \bowtie \sigma_3(R_3)$	$\sigma_1 * \sigma_2 * J_2$	$\sigma_1 * \sigma_2 * \sigma_3 * J_2 * J_3$	$\sigma^3 * J^2$
	...	...		
N-1.			$\sigma_1 \Pi_{i=2}^{N-1}\sigma_i J_i$	$\sigma^{N-1} * J^{N-2}$
N.	$A^N = A^{N-1}\sigma_N(R_N)$	$\sigma_1 \Pi_{i=2}^{N-1}\sigma_i J_i$	$\sigma_1 \Pi_{i=2}^{N}\sigma_i J_i$	$\sigma^N * J^{N-1}$

So the total number of wrapper queries W to evaluate update query UQ is

$$W = \begin{cases} 0 & : \quad N = 1 \\ \sigma_1 & : \quad N = 2 \\ \sigma_1 \sum_{k=2}^{N}(\Pi_{i=2}^{k-1}\sigma_i J_i) & : \quad N > 2 \end{cases} \tag{B.2}$$

When we assume all the σs and Js are the same,

$$W = \begin{cases} 0 & : \quad N = 1 \\ \sigma \sum_{i=0}^{N-2}(\sigma^i J^i) & : \quad N > 1 \end{cases} \tag{B.3}$$

For example, when $N = 3$ as in most of our experiments, we have $W_3 = \sigma + \sigma^2 J$.

We also obtained the final answer size $card(A^N) = \sigma_1 \Pi_{i=2}^{N}\sigma_i J_i$ or in the uniform scenario, $card(A^N) = \sigma^N J^{N-1}$.

A compensating query have the following format: $CQ = U_1 \bowtie U_2 \bowtie \ldots \bowtie U_n \bowtie R_{n+1} \bowtie \ldots \bowtie R_N$. This query contains n constant relations (tuples if no batching) and $N - n$ base relations. Let $j_i (2 \leq i \leq N, 0 \leq j_i \leq 1)$ to be the possibility that U_i joins with one tuple in A^{i-1}. In the uniform scenario, the number of wrapper queries to evaluate this query is:

$$W_n = \begin{cases} 0 & : \quad n = N \\ \sigma^n j^{n-1} \sum_{i=0}^{N-n-1}(\sigma^i J^i) & : \quad N > 1 \end{cases} \tag{B.4}$$

This formula is used in Appendix B.2 to estimate the number of wrapper queries for compensating queries.

B.2 Estimating the number of compensating queries

For ECA, CECA and CStrobe view managers that send compensating queries (CQs), each compensating query Q is generated because a recent update interferes with a particular previous query Q'. We can think of Q as a "direct child" of Q', and write $Q \in CHILD(Q')$. If Q' itself is a compensating query, it must be a child of another previous query. Let $UQ = V\langle U \rangle$ be an update query. We consider all the CQs that are "descendents" of UQ. Let CQ^i be the collection of the ith generation of compensating queries of UQ. That is, let $CQ^1 = CHILD(UQ)$, $CQ^2 = \cup_{q \in CQ^1} CHILD(q)$, ..., $CQ^{i+1} = \cup_{q \in CQ^i} CHILD(q)$. Since a compensating query always has one less base relations than its parent query, the query UQ with $N-1$ base relations has at most $N-1$ generations of descendent queries.

Let I be the maximum possible number of interfering updates between the view manager sends a query Q to the QP and receives the corresponding answer. Then, the query Q can have at most I direct children queries. The following table shows the properties of all compensating queries that were originated form an update query UQ, column n refers to the number of constant relations in one of the queries.

Query	# of queries	n	# of base relations	# of wrapper queries per Q
UQ	1	1	N-1	$\sigma \sum_{i=1}^{N-2}(\sigma^i J^i)$
CQ^1	I	2	N-2	$\sigma^2 j \sum_{i=1}^{N-3}(\sigma^i J^i)$
CQ^2	I^2	3	N-3	$\sigma^3 j^2 \sum_{i=1}^{N-4}(\sigma^i J^i)$
...	...			
CQ^{N-2}	I^{N-2}	N-1	1	$\sigma^{N-1} j^{N-2}$
CQ^{N-1}	I^{N-1}	N	0	0

Corresponding to an update query UQ, the maximum number of all queries (including UQ and all compensating queries) is $C = \sum_{i=0}^{N-1} I^i$. This is the number of all queries sent from the view manager to the query processor corresponding to an update U.

From Appendix B.1, the number of wrapper queries corresponding to a query with $N-n$ base relations is

$$W_n = \begin{cases} 0 & : \quad n = N \\ \sigma^n j^{n-1} \sum_{i=0}^{N-n-1}(\sigma^i J^i) & : \quad N > 1 \end{cases}$$

So for all the queries corresponding to UQ, we can estimate the number of wrapper queries in the worst case scenario, when there are I interfering updates for *each* query.

$$W = \sum_{k=1}^{N-1}(num\_queries * num\_wrapperQ) = \sum_{k=1}^{N-1}(I^{k-1}\sigma^k j^{k-1} \sum_{i=0}^{N-i-1}(\sigma^i J^i)) \tag{B.5}$$

For example, when $N = 3$, we have $W_3 = 1 * \sigma * (1 + \sigma * J) + I * \sigma^2 * j = \sigma(1 + \sigma J) + I\sigma^2 j$. We can intuitively see that the first term is the total number of wrapper queries to evaluate UQ itself, and the second term is the number of queries to evaluate I compensating queries, each with format $U_1 \bowtie U_2 \bowtie R_3$. If I is the average number of interfering updates for a query, then W presents the average number of wrapper queries per update query. For example, in the experiment in Section 6, we have $I = 0.5$, $J = j = 1$ and $\sigma = 1$, so $W = \sigma(1 + \sigma J) + I\sigma^2 j = 1 * (1 + 1) + 0.5 * 1 * 1 = 2.5$, and for K updates, the total number of wrapper queries is $K * W = 2.5K$.

C Glossary

Modules:

INT	integrator	one
VM	view manager	one for each view
QP	query processor	one or more
WR	source wrappers	one for each source
WH	warehouse manager	one

Base Relations:

$R(A, B, RX)$, $S(B, C, SX)$, $T(C, D, TX)$, $U(D, E, UX)$. Among the attributes, A, B, C, D, E are 10 bytes each, they are key attributes of their corresponding relations. RX, SX, TX, UX are 80 bytes each, they are not keys. Each tuple of each relation is 100 bytes.

Views:

$V_1 = \Pi_{A,B}(R)$

$V_2 = \Pi_{R.A,R.B,S.B,S.C}(R \bowtie S)$

$V_3 = \Pi_{R.A,R.B,S.B,S.C,T.C,T.D}(R \bowtie S \bowtie T)$

$V_4 = \Pi_{R.A,R.B,S.B,S.C,T.C,T.D,U.D,U.E}(R \bowtie S \bowtie T \bowtie U)$

View Parameters:

N	number of base relations in view definition
K	total number of updates for one test
K_1	total number of insertions
K_2	total number of deletions
σ	selectivity, $0 \leq \sigma \leq 1$
J	join factor
$ts(r)$	tuple size of relation r
$card(r)$	cardinality of relation r

Message contents:

U_i	source update i	from a source monitor to INT, then to VM
Q_i	update query corresponding to U_i	from VM to QP
A_i	answer relation of Q_i	from QP to VM
$\triangle V_i$	delta view corresponding to U_i	from VM to WH

Queries:

Update query	from VM to QP	one per update (0 for deletions and key VMs)
Compensating query	from VM to QP	depending on the update interleaving scenario
Wrapper query	from QP to source wrappers	depending on the QP query processing

Timings:

UPT: update propagation time, the elapsed time from the system receives a source update to the corresponding view delta appears in the warehouse manager.

I_b: the shortest time interval between updates that guarantees no propagation interference.

T_b: the UPT of a single update in the non-interleaving scenario. I_b and T_b are particular to each view and are obtained from experiments.