Distributed and Parallel Computing Issues in Data Warehousing (Invited Talk)*

Hector Garcia-Molina, Wilburt J. Labio, Janet L. Wiener, Yue Zhuge
Stanford University
{hector,wilburt,wiener,zhuge}@cs.stanford.edu
http://www-db.stanford.edu/warehousing/warehouse.html

Abstract

A data warehouse is a repository of data that has been extracted and integrated from heterogeneous and autonomous distributed sources. The warehouse data is used for decision-support or data mining. In this paper we illustrate some of the challenges in distributed and parallel computing faced by such systems. Our examples come from research done in the Stanford WHIPS Project.

1 Introduction

A data warehouse is a repository of data that has been extracted and integrated from heterogeneous and autonomous distributed sources. For example, a grocery store chain might integrate data from its inventory database, sales databases from different stores, and its marketing department's promotions records. The store chain could then: (1) find out how sales trends differ across regions of the country or world; (2) correlate its inventory with current sales and ensure that each store's inventory is replaced in keeping with its sales; (3) analyze which promotions lead to increased product sales. For example, the store chain might discover that its salsa sells better in Texas than in California, move its salsa inventory to Texas and increase salsa promotions there, and consequently increase its overall sales and revenue. Furthermore, the freed shelf space in California could be used for a popular product there, perhaps organic tofu, also increasing sales and revenue.

The goal of this paper is to give an overview of data warehousing, focusing on the opportunities for distributed and parallel computing. As we will see, a warehousing system is naturally distributed, collecting data from many sources. The warehouse itself, where the data is concentrated, often has many processors and disks. Furthermore, since records from multiple databases are typically combined at the warehouse, it must handle massive amounts of data, requiring parallel processing whenever possible. For example, Sagent Technology,

*This research was funded by Rome Laboratories under Air Force Contract F30602-94-C-0237, by the Massive Digital Data Systems (MDDS) Program sponsored by the Advanced Research and Development Committee of the Community Management Staff, and by Sagent Technologies, Inc.

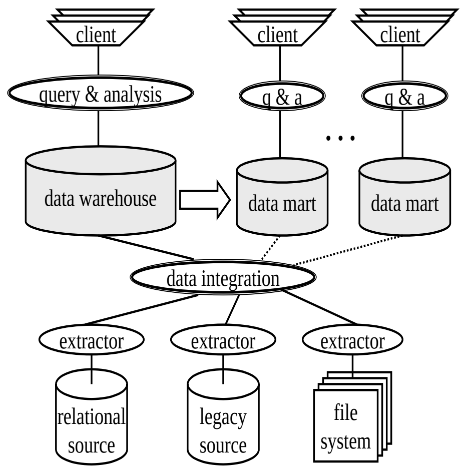

Figure 1: Warehouse architecture

whose product is used to construct data warehouses, typically builds warehouses of 100 Gb or more for its customers [8], and Walmart's data warehouse contains 24 Tb [4].

To illustrate some of the distributed and parallel computing challenges, we will describe some of the research done by the Stanford WHIPS Project (WareHousing Information Project at Stanford). The problems illustrated include how to keep warehouse data "consistent" with the distributed and autonomous sources, and how to recover from crashes during the initial warehouse load, a process that may involve many distributed components. However, before surveying our WHIPS work, we will give a brief overview of data warehousing and the overall challenges it presents.

2 Data warehousing

In Figure 1, we show the generic architecture used to create and query a data warehouse. There are several different components in this architecture, which we now describe starting from the bottom.

- *Sources* contain the raw data to be integrated. Sources may be relational databases, IMS, IDMS, or other legacy databases, or text files.
- For each source, a source-specific *data extractor* retrieves the desired source data and converts it into a uniform relational format. Generally, each data extractor is tightly coupled to its source.
- *Integration* software transforms the data into the warehouse format. Integration components may perform arbitrary transformations of the extracted tuple sequences, including byte reordering; relational project, select, and join operations that merge data from several sources; inexact duplicate elimination and other data scrubbing operations; adding timestamps and other metadata; and complex aggregate computations like computing a running total.
- The *data warehouse* stores the data.

- *Query and analysis* software allows user clients to query arbitrary subsets of the warehouse data, visualize it, display it in spreadsheets, and run data mining queries like the ones illustrated earlier.

- *Data marts* receive replicated copies of portions of the warehouse data, e.g., that a department will use to make local decisions. For example, the grocery store chain's California headquarters might have a data mart containing all inventory data but only California sales data.

The warehouse itself may be a relational database, tuned specifically as a warehouse, or it may use a specialized data model. In general, warehouse applications differ from traditional database applications in several key features. First, the quantity of data is often much larger. Second, while traditional databases are tuned for short update transactions (one customer buys a cartful of groceries), a data warehouse is tuned for long-running queries (find all products that sold more than 1 million items today). Third, traditional databases usually contain only the current state of the world (today's inventory). A data warehouse is likely to archive many previous states and add temporal information such as timestamps. The previous states are necessary for historical queries, e.g., does barbecue sauce sell better in June or September? Furthermore, the warehouse may contain precomputed summaries of the data, such as total sales for the day, month, and year, in addition to the individual sales records.

In database terminology, the integration software computes *materialized views* over the source data. A view is just a named query. A materialized view is a table containing the query result. In our framework, the warehouse table definitions are views over the source data. During warehouse *creation*, the software integration operations execute the "query" over the sources and populate the warehouse table with the result. The table then stores a materialized view.

Warehouse *maintenance* occurs after the source data changes, to incorporate those changes in the materialized view. There are two options for updating the view. (1) *Recomputation*: The old view contents are discarded and the view definition query is re-evaluated over the new data. (2) *Incremental maintenance*: The current view contents and the changes to the underlying data are used to compute the changes to the view. Both recomputation and incremental maintenance may occur after each source update or periodically. In the warehouse context, view maintenance may be simplified if the view is defined over *copies* of the source data that are also stored in the warehouse. However, these copies must also be updated.

As mentioned in Section 1, data warehousing comprises many computing elements, some tightly coupled, some geographically distributed. There are many opportunities for distributed and parallel processing. In particular,

- The sources are distributed, heterogeneous and autonomous.

- The integration software is a separate entity from both the sources and the warehouse. In addition, the integration software itself can be many individual distributed components. In our warehouse prototype at Stanford, for example, we break integration into many smaller tasks and use a separate CORBA object for each one. Some commercial products, such as Sagent's Design Studio software, divide integration into many smaller transformations, each of which can be a separate process. In addition to being distributed, the integration components can often execute in parallel.

- The warehouse itself may be a parallel database. For example, the data warehouse used by Walmart is a 96 node Teradata database [4]. If the warehouse is a conventional database system, it may still run transactions concurrently.
- Data marts are distributed copies of warehouse data.
- Data warehouse and data mart clients may be distributed geographically over a very large area, even worldwide.

For many of these components, standard technology can be used. For example, data marts can be implemented using data replication software, and parallel database software can be used as a warehouse. Similarly, standard protocols can be used for communication between the distributed components. However, the specific requirements of data warehousing also render some standard distributed database protocols inapplicable. For instance, two-phase commit between the sources and the warehouse cannot be used to prevent inconsistencies as warehouse data is being updated. (The sources are autonomous and may not export a commit interface. Even if they did, the performance overhead would be prohibitive.) We address some of these challenges in Section 4. First, we overview general challenges in data warehousing.

3 Challenges in data warehousing

Quite a few commercial products are aimed at the data warehousing market. Some examples include Prism, Sagent, and Apertus to build a data warehouse; Oracle, DB2, Sybase, and Redbrick to serve as relational data warehouses and Pilot and Essbase as multi-dimensional warehouses; and Andyne, Brio, and Cognos as client access tools. A much more extensive list at `http://pwp.starnetinc.com/larryg` contains hundreds of warehouse-related products. However, there are numerous issues that these products have yet to address. In particular, after the warehouse has been initialized with the source data, the sources may continue to change. These changes need to be propagated to the warehouse as well. We now enumerate some of the research challenges in creating and maintaining a data warehouse that we are addressing at Stanford. For a less Stanford-centric overview of data warehousing, see [16, 5, 17].

- A relational database source may provide triggers or a replication program that will notify the data extractor of source changes. Other sources are not as capable. Detecting changes in flat file (text) sources requires examining both old and new copies of the files. Efficient algorithms are needed to detect changes in flat file sources [12] and legacy sources.

- Sources may continue to change their data after the warehouse is created. Warehouse maintenance algorithms are then needed. Both recomputation and incremental view maintenance are well understood for centralized relational databases [3, 11, 10, 9]. However, more complex algorithms are required when updates originate from multiple sources and affect multiple views. Otherwise, the warehouse may not contain accurate data. Current commercial systems assume that the sources are quiescent during maintenance. We describe our approach to on-line warehouse view maintenance in Section 4.

- Summary tables, such as daily total sales or average January sales, are created from large underlying tables, e.g., all sales. They require special maintenance algorithms to avoid rescanning the entire underlying table when there are incremental changes [14].

- Warehouse creation and maintenance loads typically take hours to run. Most of the work occurs in transformations after the data is extracted from the sources and before it is stored in the warehouse. If the load is interrupted by failures, traditional recovery algorithms *undo* the incomplete load. The administrator must then restart the load, wait the full load time, and hope it does not fail again. A better approach is to resume the incomplete load. We discuss our load resumption algorithms in Section 5.

- In current warehousing systems, maintenance operations usually are isolated from client read activity, limiting the availability and size of the warehouse. A more effective approach is to maintain multiple logical versions of updated warehouse data, so that maintenance transactions can run currently with readers. Furthermore, multiple versions permit the readers' data to remain stable until well-publicized times. Efficient multi-version algorithms are needed [15].

- Most work in view maintenance for data warehousing only considers non-temporal views. Automatic maintenance of temporal views over nontemporal source relations is necessary to allow users to ask temporal (historical) queries using these views. Their maintenance is further complicated because, due to the dimension of time, a materialized temporal view may need to be updated not only when source relations change, but also as time advances [18].

- Given a data item in a materialized warehouse view, analysts often want to identify the set of source data items that produced the view item. Algorithms to trace the lineage of an item from the view back to source data items from multiple sources are needed [6].

So far as we know, none of these areas have been tackled by the current generation of commercial products. While our ongoing work addresses some of the challenges outlined above, future research into efficient policies and algorithms is needed in all of these areas. In the next sections, we illustrate some of our work, focusing on problems that involve some form of distributed or parallel computing.

4 Maintaining warehouse data consistency

The data at the warehouse is a derived *copy* of data at the sources, and hence consistency is an issue. Because of source autononomy or performance issues, traditional solutions for maintaining consistency may not be applicable. Instead, we need to develop specialized solutions that exploit the semantics of warehouse updates to avoid inconsistencies without requiring sources to lock data or to modify their procedures. In this section we illustrate these issues and our solutions. We start by showing that even with a single source and a single warehouse view there can be problems. Then in the following two subsections we consider multiple sources and multiple views.

4.1 Autonomous sources

We start in Example 1 by showing a correct view maintenance scenario for a view defined over a remote autonomous source. Then we will illustrate the potential problems that may arise. Let the maintenance for each view V be handled by a *view manager* $\mathcal{V}$ in the integration component.

Example 1: Correct view maintenance. Consider a view V_1 defined over two relations *accounts(client-id, stock, num-shares, price-paid)* and *inquiries(client-id, date, topic)* at a source *Portfolios*. Let the current (simple) contents of the relations be as follows:

accounts:			
client-id	stock	num-shares	price-paid
101	IBM	2000	18.5

inquiries:		
client-id	date	topic
101	4/2/98	"price of SGI"

View V_1 is defined as $V_1 = accounts \bowtie inquiries$. (The $\bowtie$ operator performs a natural join, combining tuples that match on the attributes that have the same names.) Thus, V_1 initially contains one tuple, [101, IBM, 2000, 18.5, 4/2/98, "price of SGI"]. Now suppose an update U_1 occurs. A typical relational view maintenance algorithm (such as [3]) will handle U_1 as follows.

1. The *Portfolio* data extractor detects update U_1 = Insert(*inquiries*, [101, 4/7/98, "price of ITT"]) and forwards it to the view manager $\mathcal{V}_1$.

2. Manager $\mathcal{V}_1$ receives U_1. It needs to know which tuples in *accounts* join with U_1, so it sends query $Q_1 = accounts \bowtie$ [101, 4/7/98, "price of ITT"] to the source extractor.

3. The *Portfolio* extractor evaluates Q_1 and returns the answer A_1 = [101, IBM, 2000, 18.5, 4/7/98, "price of ITT"].

4. Manager $\mathcal{V}_1$ receives A_1 and adds A_1 to the warehouse view V_1.

The final view V_1 correctly contains two tuples. □

In our next example we will show how the simple maintenance algorithm above can lead to an incorrect view when there are concurrent updates. However, before proceeding it is important to note that duplicate tuples must be stored in the materialized warehouse views. (Conventional relations typically do not store duplicates.) As a very simple example, suppose that a view is defined over *inquiries*, but only retains the attributes *client-id* and *topic* (i.e., attribute *date* is projected out). Suppose that the source relation has two tuples, [101, 4/2/98, "price of SGI"] and [101, 4/5/98, "price of SGI"]. The warehouse view must store two copies of [101, "price of SGI"]. To see why two copies are necessary, consider what happens when the source deletes tuple [101, 4/5/98, "price of SGI"]. If there is only one copy, the view manager will delete the one copy, leaving the view inconsistent. If there are two copies, on the other hand, one copy can be removed when the deletion is reported, leaving the other copy. Copies can be tracked in views either by explicitly storing copies, or by keeping a "count" attribute on a single copy.

Example 2: View maintenance anomaly. Consider the same relations and view V_1 from Example 1, with the same contents for *accounts*. However, let the initial contents of *inquiries* be empty. The view V_1 is initially empty as well. Suppose there are two updates U_1 and U_2 as follows.

1. The *Portfolio* data extractor detects update U_1 = Insert(*inquiries*, [101, 4/7/98, "price of ITT"]) and forwards it to the view manager $\mathcal{V}_1$.

2. Manager $\mathcal{V}_1$ receives U_1 and sends query Q_1 = *accounts* $\bowtie$ [101, 4/7/98, "price of ITT"] to the source extractor.

3. The extractor detects update U_2 = Insert(*accounts*, [101, APP, 3000, 12.2]) and forwards it to the view manager $\mathcal{V}_1$.

4. Manager $\mathcal{V}_1$ receives U_2 and sends query Q_2 = [101, APP, 3000, 12.2] $\bowtie$ *inquiries* to the *Portfolio* extractor.

5. The extractor receives Q_1 and the source evaluates it on the current base relations. The resulting answer is A_1 = ([101, IBM, 2000, 18.5, 4/7/98, "price of ITT"], [101, APP, 3000, 12.2, 4/7/98, "price of ITT"]), which is sent to $\mathcal{V}_1$.

6. Manager $\mathcal{V}_1$ receives A_1 and updates the view to $V_1 \cup A_1 = A_1$.

7. The extractor receives Q_2 and evaluates it to A_2 = ([101, APP, 3000, 12.2, 4/7/98, "price of ITT"]). It sends A_2 to $\mathcal{V}_1$.

8. Manager $\mathcal{V}_1$ receives A_2 and adds it to the warehouse view V_1. The view now contains the tuple [101, APP, 3000, 12.2, 4/7/98, "price of ITT"] twice, which in this case is incorrect. □

The problem in Example 2 is that Q_1 is evaluated on a different source state than existed at the time that U_1 occurred and caused Q_1 to be issued. Such view maintenance *anomalies* occur when the view manager tries to update a view while the base data at the source is changing. These anomalies arise in warehousing because the view maintenance is decoupled from the source updates. Both insertion and deletion updates can cause anomalies.

Previous view maintenance algorithms assume that sources know about the view definitions, and can include all relevant information with an update. In the warehouse environment, however, sources can be legacy or unsophisticated systems that do not understand views. When information about an update arrives at the integration component, it may discover that additional information is needed to update the view. Thus, it may issue queries back to the sources, as illustrated in our example. As we saw, these queries are evaluated at the source later than the corresponding update, so the source may have changed. This decoupling between the source data and the view maintenance machinery can lead to incorrect views.

Traditional distributed database solutions require that the source lock its data (prevent updates) during view maintenance, or use timestamps to detect concurrent updates or "stale" queries. Since we cannot impose such restrictions on the sources, we developed the *Eager Compensating Algorithm (ECA)* for view maintenance. ECA modifies each query sent to the source by adding *compensating queries* to offset the effect of concurrent updates. In Example 2, when $\mathcal{V}_1$ receives U_2, ECA will realize that the previously sent query Q_1 will be answered in a state after U_2. (Otherwise, $\mathcal{V}_1$ would have received A_1 before U_2.) Therefore, query Q_2 is modified to compensate as follows: Q_2 = ([101, APP, 3000, 12.2] $\bowtie$ *inquiries*) minus ([101, APP, 3000, 12.2] $\bowtie$ [101, 4/7/98, "price of ITT"]). The first part of Q_2 is unchanged; the second part compensates for the extra tuple that Q_1 sees. Due to the compensation, the answer to Q_2 is empty and the final view is correct.

In [19], we present ECA in detail, define warehouse view consistency formally, and prove that ECA guarantees consistency for views over one source. We also discuss how view keys (attributes that can uniquely identify source tuples that contributed to a view tuple) can simplify processing. Note that the same consistency problems arise whether the views are traditional select-project-join views, or contain more complex transformations. Any maintenance transformation that requests information from a source is subject to the same anomalies.

We note that an alternate solution is to copy all base data at the warehouse. Since the anomalies only arise when maintaining views with joins over base data, traditional algorithms can be used to maintain both the copies (which do not have joins) and the new join views (over warehouse data). However, copies impose high overhead both in storage cost and in maintenance, especially if only a small fraction of the data participates in the view.

4.2 Multiple sources

Views defined over multiple sources pose further maintenance challenges, since it may not be obvious when new updates from a source impact the processing of previous updates. We illustrate the new potential anomalies in Example 3.

Example 3: Multiple source anomaly. Consider the earlier relation *accounts(client-id, stock, num-shares, price-paid)* at a source *Portfolios*. Let two additional sources *Price-Earnings* and *Stocks* contain the relations *PE(stock, pe)* and *daily(date, stock, closing)*. Let view V_2 compute a join over all three relations. Suppose the current (simple) contents of the relations are as follows:

accounts:			
client-id	stock	num-shares	price-paid
101	IBM	2000	18.5

daily:		
date	stock	closing

PE:	
stock	pe
IBM	5

View V_2 is defined as V_2 = *accounts* $\bowtie$ *daily* $\bowtie$ *pe*. View V_2 is initially empty. Now suppose updates U_1 and U_2 occur. Using ECA or a conventional view maintenance algorithm, the following scenario may occur.

1. The *Stocks* data extractor detects update U_1 = Insert(*daily*, [6/7/98, IBM, 15]) and forwards it to the view manager $\mathcal{V}_2$.

2. Manager $\mathcal{V}_2$ receives U_1. It needs to know which tuples in *accounts* and *PE* join with U_1, so it first sends query $Q_1 = \textit{accounts} \bowtie$ [6/7/98, IBM, 15] to the *Portfolio* source extractor.

3. The *Portfolio* extractor evaluates Q_1 and returns the answer A_1 = [101, IBM, 2000, 18.5, 6/7/98, 15].

4. Manager $\mathcal{V}_2$ receives A_1 and sends Q_2 = [101, IBM, 2000, 18.5, 6/7/98, 15] $\bowtie$ *PE* to the *Price-Earnings* data extractor.

5. The *Portfolio* extractor detects update U_2 = Delete(*accounts*, [101, IBM, 2000, 18.5] and sends it to $\mathcal{V}_2$.

6. Manager $\mathcal{V}_2$ receives U_2. Since the view is empty, no action is taken for this deletion. (Since U_2 includes a key for *accounts*, there is no need to join the tuple with the other relations before performing the delete.)

7. The *Price-Earnings* extractor evaluates Q_2 and returns A_2 = [101, IBM, 2000, 18.5, 6/7/98, 15, 5] to $\mathcal{V}_2$.

8. Manager $\mathcal{V}_2$ receives A_2, the final answer regarding update U_1. Since there are no pending queries or updates, A_2 is inserted into V_2. This final view is incorrect. □

In the above example, the interleaving of the queries for U_1 with updates arriving from the sources causes the incorrect view. In ECA, we compensated for the updates that occurred at the source before the query was processed. With multiple sources, however, we may have to compensate for updates that occur *after* the query, if they overlap even the processing of a previous update.

We propose a new algorithm *Strobe* in [20] that extends ECA for multiple sources. The Strobe algorithm keeps track of all updates that occurred at any source while processing query Q for update U. These updates are then applied to Q's answer A, before installing A in the warehouse view.

In the above example, Strobe remembers the update U_2 until after all processing for U_1 is completed. When the final answer for U_1, A_2, arrives, $\mathcal{V}_2$ applies the deletion U_2 to A_2, and then correctly inserts nothing into the view V_2. In [20] we also redefine consistency for multiple sources and prove that Strobe provides this consistency.

4.3 Multiple views

Finally, we extend the earlier work to ensure that multiple views are consistent with each other, which we call the multiple view consistency (MVC) problem. With MVC, the maintenance algorithms presented above are still used to maintain each view. However, some coordination among views is necessary before updates are propagated to the warehouse, as Example 4 demonstrates.

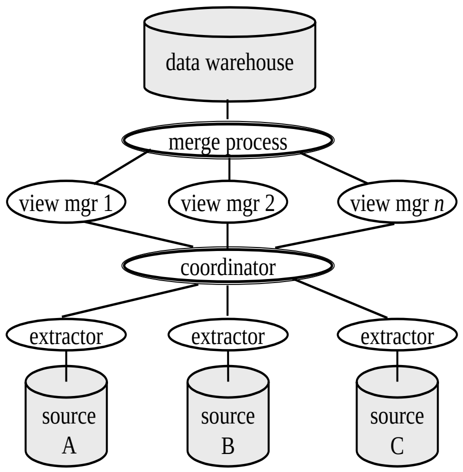

Figure 2: Maintaining consistency of multiple views

Example 4: Consistency across multiple views. Let V_1 be defined as in Example 1. Let V_3 be a copy of the *inquiries* relation. Consider the same contents of the relations and the same updates as in Example 2. When the view manager $\mathcal{V}_1$ for view V_1 receives update U_1 (an insertion to *inquiries*), it sends a query to the Portfolio extractor. Manager $\mathcal{V}_1$ does not update V_1 until it receives the answer to its query. In the meantime, when manager $\mathcal{V}_3$ for view V_3 receives update U_1 (which also affects this view), it can immediately update V_3 since no query needs to be issued. When $\mathcal{V}_3$ updates the warehouse, V_3 reflects update U_1 but V_1 does not. ($\mathcal{V}_1$ is still waiting for its query answer.) □

Although both V_1 and V_3 are consistent with source states, V_3 is consistent with a later state than V_1. The two views are not consistent with each other and any client analysis at the warehouse that uses both views may have incorrect results. The problem could be avoided by processing updates strictly sequentially, using a variant of the Strobe algorithm. However, sequential handling does not permit concurrency and limits parallelism in view maintenance. In a high update environment, sequential handling is unacceptably slow.

Instead, we propose adding another component to the warehouse maintenance architecture, the *merge process*, shown in Figure 2. (The merge process, the view managers and the coordinator are all part of the data integration component shown in Figure 1.) As updates arrive from the extractors, they are dispatched by the coordinator to the appropriate view manager(s) (whose views are impacted by the update). The view updates generated by the managers are forwarded to the merge process, which collects all updates and holds them until all affected views can be updated together. It then forwards a "batch" of the view updates to the warehouse in a single transaction. When a merge process is used in Example 4, both $\mathcal{V}_1$ and $\mathcal{V}_3$ can process their updates concurrently, and any delays incurred while waiting for queries do not interfere with the other managers. When the merge process receives the V_3 update first, it holds it (because it knows that the source update that generated this view update also affects V_1). When the merge process receives the corresponding updates to V_1, it then forwards both sets of updates to the warehouse as one transaction.

The full MVC algorithm is more complicated than this example shows. Suppose updates U_1 and U_2 both impact views V_1 and V_3. Suppose that $\mathcal{V}_1$ processes U_1 first. Then the merge process will hold it until it receives the corresponding updates to V_3. However, assume further

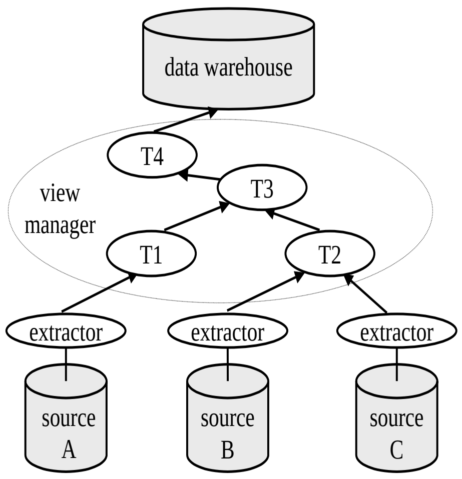

Figure 3: Transforms involved in a view manager

that $\mathcal{V}_3$ sends updates for U_1 and U_2 together. Then the merge process must continue to hold the updates until $\mathcal{V}_1$ finishes processing U_2. In general, if the merge process receives updates for view V_1 that only partially overlap the view updates received for V_3, then it cannot apply either set immediately. Instead, the merge process must hold both sets of view updates until it has a set for each of V_1 and V_3 that reflects the same sets of source updates. It can then install these view updates (although it may still be holding later sets of view updates for V_1 or V_3). Further details about the algorithm used by the merge process are in [21].

5 Resuming failed warehouse loads

Warehouse loading is often performed by a set of distributed cooperating processes. For example, in Figure 3 we show how one of our view managers could be implemented by four processes, T_1 through T_4. We call each process a *transform* since it takes one or more streams of tuples, and merges them or manipulates them into another form. Typical transforms sort data, check for errors, perform joins, filter tuples, and compute summaries.

A typical load to create or maintain a data warehouse can range from 1 to 100 Gigabytes and take up to 24 hours to execute. For example, Walmart's incremental maintenance load averages 16 Gb per day [4], and Sagent customer maintenance loads vary from 0.5 to 6 Gb nightly or weekly and up to 100 Gb to create the warehouse. If the load is interrupted by failures, traditional recovery algorithms *undo* the incomplete load, and rely on the administrator to restart it. Alternatively, persistent queues [1] or other fault-tolerant logs [2] can be used to save the data sent from one transform to another, and resend it after a failure. However, saving all the data persistently imposes a lot of overhead in both time and disk space.

Our approach is to perform the load without any logging, simply transforming data and storing it into the warehouse as fast as possible. (Typically, load utilities are used to enter data into the warehouse; these utilities do not provide any transactional guarantees.) When any failure occurs, we stop the entire load process. (We do not attempt to restart a single failed transform, as in distributed process recovery, e.g., [7].) We analyze the tuples

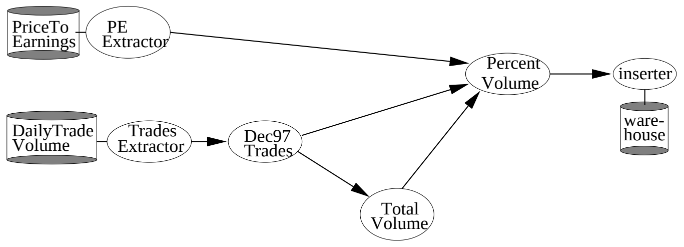

Figure 4: Components involved in warehouse load in trade volume scenario

that did make it into the warehouse, and we try to infer where the computation was at the time of the failure. We restart the entire load process, but based on what data had made it to the warehouse, we try to avoid reading unnecessary data from the sources, or redoing computations whose results are already in the warehouse.

Example 5: Resumption. Figure 4 shows the transforms used for a particular warehouse view. In this scenario, data is extracted from the *DailyTradeVolume* (*Trades*) and *PriceToEarnings* (*PE*) source relations. The stock trade data is first filtered by the *Dec97Trades* transform, which only outputs trades from December, 1997. The output of this transform is then sent to both the *TotalVolume* and *PercentVolume* transforms. The *TotalVolume* transform computes the total trade volume over its input, and sends its output to *PercentVolume*. The *PercentVolume* transform then groups the trades by company and finds the percent of the total trade volume contributed by companies whose price-to-earnings ratio is greater than 4. For those companies, it sends a tuple containing the company, price-to-earnings ratio, and percentage of the trade volume to the warehouse.

For instance, suppose that the total volume of stocks traded in December 1997 was 1000. The output of the *TotalVolume* transform will be a single tuple with the value 1000. Suppose that the volume of trades for stock AAA was 14.5 in December 1997, while the volume for IBM was 20.8. If the price-to-earnings ratio of AAA was 5, while the ratio for IBM was 3, then a tuple for AAA will be output by *PercentVolume* (reporting a percent volume of 14.5 divided by 1000), while no tuple for IBM will be generated.

company	*pe*	*percentvol*
AAA	5	1.45
INTC	8	3.76
MSN	4	0.12

Figure 5: Sequence of tuples stored in the warehouse before the failure

To illustrate the intuition behind our resumption process, suppose that a failure occurs after the sequence of tuples shown in Figure 5 is stored in the warehouse. When we redo the load, we try to avoid redoing the work that lead to the tuples in Figure 5. For example, we may avoid re-extracting from source relation *PE* the price-to-earning information for stocks AAA, INTC, and MSN. If *PE* provides its data in alphabetical order, and this order was preserved by the transforms, we can avoid re-reading all *PE* information up to stock MSN. The data from *Trades* does have to be fully read, since we need to recompute the total volume. However, the output tuples coming out of *Dec97Trades* and going to *PercentVolume* can be filtered, e.g., those tuples that refer to AAA, INTC and MSN can be dropped. □

In summary, our approach is to redo the load after a failure, avoiding repeated effort. We use the warehouse contents to identify which tuples in each transform's input sequence have already been processed, and remove them from the input. To do this, we rely on knowing certain properties of the transforms, e.g., if they process tuples in order, or if they map multiple input tuples to a single output tuple. These properties are defined in [13] and can usually be inferred from the transform interface. We do *not* need to understand exactly what the transforms do, only their "data flow" properties. That is, we view transforms as opaque software modules, whose detailed functionality is unknown to the recovery system.

Our resumption algorithm proceeds in two phases. Once the graph (workflow) of transforms used in the load is known, the *design* phase of the algorithm decides where to place filters during resumption. A filter before a transform removes tuples from the transform's input tuple sequence when they (or all the tuples to which they contribute) are already stored in the warehouse. These filters remain inactive during normal load operation. After a failure, the *resume* phase instantiates the filters with the actual tuples already in the warehouse. Other than instantiating the filters and actually removing input tuples, most of the work is performed only once, in the design phase. The filters can then be used multiple times to resume different loads.

To add a filter to a transform's input, the algorithm must determine that three requirements are satisfied. (1) There must be a set of common attributes between the transform input tuples and the warehouse tuples that identifies exactly the input tuples that contribute to each warehouse tuple. These are the *identifying attributes.* (2) An input tuple is filtered only if all of the tuples to which it contributes are already in the warehouse prior to the failure. (3) If the filter removes a prefix of input tuples (i.e., a DiscardPrefix filter), then the input tuple order at load resumption time must be guaranteed to be the same as it was during the original load. There is no such requirement if the filter removes a subset of the input tuples (i.e., a DiscardSubset filter).

In [13], we define the transform properties and filter requirements formally. The algorithm computes the filter requirements from the properties and decides where filters are feasible. The resumption algorithm also determines when the data extractors can request only a subset or suffix of their original data from the sources, although we do not discuss that portion of the algorithm here.

Example 5 (Continued). We now apply the resumption algorithm to the scenario illustrated in Figure 4. The results are shown in Figure 6. During the design phase, the algorithm assigns the filter DiscardPrefix to the $PercentVolume$ input produced by the PE data extractor. The PE attribute *company* is a key for both PE and the warehouse and serves as the identifying attribute. This means that this attribute can be used to identify exactly which tuples can be ignored during the re-load. Furthermore, assume that the PE data extractor always produces the input in the same order. Therefore, all three requirements for assigning DiscardPrefix are satisfied. The filter DiscardSubset is assigned to the $PercentVolume$ input produced by transform $Dec97Trades$. Since *company* is a key at-

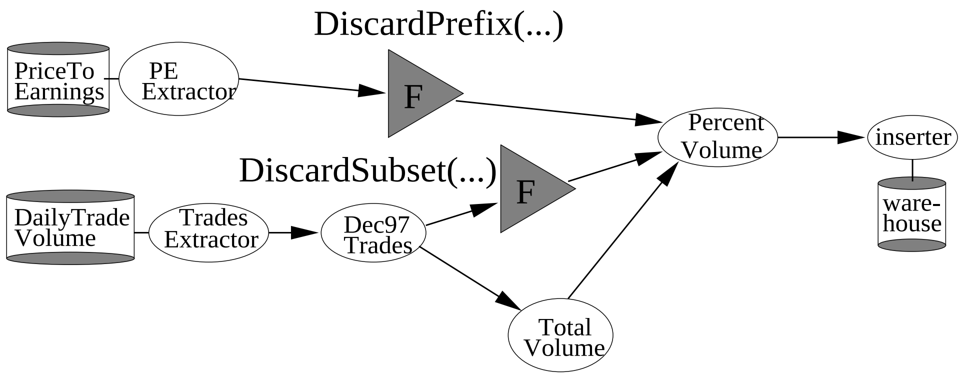

Figure 6: Re-extraction procedures and filters assigned

tribute of the warehouse, it is an identifying attribute for the filter. The algorithm further determines that each $Dec97Trades$ tuple contributes to at most one tuple in the output of $PercentVolume$, so both requirements for DiscardSubset are satisfied. However, we assume that $Dec97Trades$ does not produce the input tuples in the same order, so the filter DiscardSubset is assigned rather than DiscardPrefix.

Suppose again that the load fails when the tuples shown in Figure 5 are stored in the warehouse. The resume phase begins. The DiscardPrefix filter is instantiated with the last tuple in the warehouse as DiscardPrefix($[MSN]$). It scans the tuples until it finds one with the same *company* attribute value (MSN). The filter's output starts with this tuple. The DiscardSubset tuple filter is instantiated with all of the warehouse tuples as DiscardSubset($[[AAA], [INTC], [MSN]]$), which removes any tuple in its input with a *company* attribute value of AAA, $INTC$, or MSN. In database terms, it performs an anti-semijoin between its input and the warehouse tuple sequence. All other tuples are output by the filter. □

In [13], we also discuss an extension to our incremental resumption algorithm that uses logs of a transform's output to reduce work at resumption time. We discuss both where to add logs and how to use them effectively.

6 Parallelizing view maintenance tasks

Warehouse maintenance often requires massive amounts of processing to perform joins, aggregations, index maintenance, and change installation. During maintenance, either the warehouse is unavailable for querying, or the user queries compete with view maintenance queries for warehouse resources. Hence, it is important to minimize view maintenance time. Fortunately, the process of maintaining warehouse views can be divided into "tasks" that can be done in parallel.

It is a challenging problem to schedule the tasks in parallel, in order to minimize the overall execution time. Additional complexity is introduced since there are numerous methods for maintaining warehouse views, and each method divides view maintenance into *different* sets of tasks constrained by different partial orders. In this section we briefly illustrate some of the choices [11]. Our goal is simply to identify this area as worthy of research, rather than to offer specific answers.

Example 6: Parallelization. Suppose that the warehouse maintains two views, V_1 and V_2, defined on source data. The algorithms of Section 4 are used to maintain these views, i.e., periodically a set of tuples to insert and a set of tuples to delete are computed for each view. Let $\triangle V_1$ and $\bigtriangledown V_1$ be the tuples to insert and delete from V_1, and similarly, let $\triangle V_2$ and $\bigtriangledown V_2$ be the tuples for V_2.

In addition, there is a view V_3 defined at the warehouse as $V_1 \bowtie V_2$. The changes to V_1 and V_2 in turn trigger changes to V_3, represented by $\triangle V_3$ and $\bigtriangledown V_3$. After the updates to V_1 and V_2 are computed, there are many ways in which we can do the remaining work. Here we illustrate two options:

1. *Late Evaluation.* We defer updating V_1 and V_2 until after the changes to V_3 have been computed. That is, first we compute the insertions to V_3, $\triangle V_3$, as $\triangle V_1 \bowtie V_2$ union $V_1 \bowtie \triangle V_2$ union $\triangle V_1 \bowtie \triangle V_2$. We compute $\bigtriangledown V_3$ is a similar fashion. Finally, we update all the views. The new value for V_1 is V_1 union $\triangle V_1$ minus $\bigtriangledown V_1$. The other two views are updated in a similar way.

2. *Early Evaluation.* We first compute the changes to V_3 caused by $\triangle V_1$ and $\bigtriangledown V_1$, then update V_1, and finally compute the rest of the changes. (It is also possible to update V_2 first, but we do not consider this here.) That is, we initially compute $\triangle V_3$ as $\triangle V_1 \bowtie V_2$ and $\bigtriangledown V_3$ as $\bigtriangledown V_1 \bowtie V_2$. These are *partial* sets since they do not yet reflect V_2 changes. We then update V_1 to V_1 union $\triangle V_1$ minus $\bigtriangledown V_1$. Then we add to $\triangle V_3$ the set $V_1 \bowtie \triangle V_2$, and similarly compute the deletes $\bigtriangledown V_3$. Note that in these last two computations we use the updated value of V_1. As a last step, we update V_2 and V_3.

These two schemes differ not just in the amount of work done, but also in how it can be parallelized. For example, late evaluation performs more joins, but they can all be done in parallel because they all use the non-updated states of V_1 and V_2. On the other hand, early evaluation involves fewer joins, but the ones that use the new V_1 value cannot be initiated until after V_1 is updated. $\square$

In general, it can be shown that if a view joins n tables, $3^n - 1$ compute tasks are needed by the late installation method. On the other hand, early installation requires only $2n$ tasks. However, as we observed, these tasks cannot be parallelized as easily since the method imposes ordering constraints. In addition to these choices, there are numerous ways to implement early computations, depending on which views are updated first. Performance is impacted by all these choices, and in addition by other parameters such as the number of processors available, the size of the views, and the number of tuples that match in the join operations. Overall, there are a huge number of choices for maintaining the materialized views at the warehouse. Selecting the best one is important since the differences in performance can be very significant. We are currently working on an efficient algorithm with a good set of heuristics for choosing an early versus late installation method, and then a set of tasks within the method.

7 Conclusions

Data warehousing presents many new opportunities for distributed and parallel computing. In this paper we outlined some of the distributed and parallel aspects of creating, maintaining, and querying a data warehouse. We then focused on several new challenges for warehouse maintenance. First, new algorithms are needed to maintain warehouse consistency since warehouse maintenance is performed by a set of distributed computing components that receive data from autonomous sources. Second, new approaches are needed to resume interrupted warehouse loads, because traditional approaches either have too much overhead or repeat all of the work. Third, the parallelization of maintenance tasks is an important research area, since there are many choices and the performance implications are significant. We hope that by discussing some of these problems, we stimulate more interest in data warehousing from the distributed and parallel computing community.

References

[1] P. A. Bernstein, M. Hsu, and B. Mann. Implementing Recoverable Requests Using Queues. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 112–122, 1990.

[2] P. A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan-Kaufman, Inc., San Mateo, CA, 1997.

[3] J.A. Blakeley, P.-A. Larson, and F.W. Tompa. Efficiently Updating Materialized Views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 61–71, Washington, D.C., June 1986.

[4] Felipe Carino. High-Performance, Parallel Warehouse Servers and Large-Scale Applications, October 1997. Talk about Teradata given in Stanford Database Seminar.

[5] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record*, 26(1):65–74, March 1997.

[6] Y. Cui, J. Widom, and J.L. Wiener. Tracing the Lineage of View Data in a a Data Warehousing Environment. Technical report, Stanford University, 1997. Available from http://www-db.stanford.edu/pub/papers/lineage.ps.

[7] E.N. Elnozahy and W. Zwaenepoel. Manetho: Transparent Rollback-Recovery with Low Overhead, Limited Rollback, and Fast Output Commit. *IEEE Transactions on Computers*, 41(5):526–531, May 1992.

[8] Vlad Gorelik. System Architect, Sagent Technology, Inc., May 1998. Personal communication.

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

[10] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):4–19, June 1995.

[11] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 157–166, May 1993.

[12] W. J. Labio and H. Garcia-Molina. Efficient Snapshot Differential Algorithms for Data Warehousing. In *Proceedings of the International Conference on Very Large Data Bases*, 1996.

[13] W.J. Labio, J.L. Wiener, H. Garcia-Molina, and V. Gorelik. Algorithms for Resuming Interrupted Warehouse Loads. Technical report, Stanford University, 1998.

[14] I. Mumick, D. Quass, and B. Mumick. Maintenance of Data Cubes and Summary Tables in a Warehouse. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tuscon, Arizona, May 1997.

[15] D. Quass and J. Widom. On-Line Warehouse View Maintenance for Batch Updates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tuscon, Arizona, May 1997.

[16] J. Widom. Research Problems in Data Warehousing. In *Conference on Information and Knowledge Management*, 1995. Also available at http://db.stanford.edu/pub/widom/1995/warehouse-research.ps.

[17] M. Wu and A. P. Buchmann. Research Issues in Data Warehousing. In *Proceedings of the German Database Conference (Datenbanken in Büro, Technik und Wissenschaft)*, pages 61–82, 1997.

[18] J. Yang and J. Widom. Maintaining Temporal Views Over Non-temporal Information Sources for Data Warehousing. In *Proceedings of the International Conference on Extending Database Technology*, March 1998.

[19] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, 1995.

[20] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. In *Proceedings of the Conference on Parallel and Distributed Information Systems*, 1996.

[21] Y. Zhuge, J.L. Wiener, and H. Garcia-Molina. Multiple View Consistency for Data Warehousing. In *IEEE Conference on Data Engineering*, Binghamton, UK, April 1997.