A System Prototype for Warehouse View Maintenance

Janet L. Wiener, Himanshu Gupta, Wilburt J. Labio, Yue Zhuge,
Hector Garcia-Molina, Jennifer Widom *

Department of Computer Science

Stanford University

Stanford, CA 94305-2140, USA

wiener@cs.stanford.edu

http://www-db.stanford.edu/warehousing/warehouse.html

Abstract

A data warehouse collects and integrates data from multiple, autonomous, heterogeneous, sources. The warehouse effectively maintains one or more materialized views over the source data. In this paper we describe the architecture of the Whips prototype system, which collects, transforms, and integrates data for the warehouse. We show how the required functionality can be divided among cooperating distributed CORBA objects, providing both scalability and the flexibility needed for supporting different application needs and heterogeneous sources. The Whips prototype is a functioning system implemented at Stanford University and we provide preliminary performance results.

1 Introduction

A data warehouse is a repository of integrated information from distributed, autonomous, and possibly heterogeneous, sources. In effect, the warehouse stores one or more materialized views of the source data. The data is then readily available to user applications for querying and analysis. Figure 1 shows the basic architecture of a warehouse: data is collected from each source, integrated with data from other sources, and stored at the warehouse. Users then access the data directly from the warehouse.

As suggested by Figure 1, there are two major components in a warehouse system: the *integration component*, responsible for collecting and maintaining the materialized views, and the *query and analysis component*, responsible for fulfilling the information needs of specific end users. Note that the two components are not independent. For example, which views the integration component materializes depends on the expected needs of end users.

Most current commercial warehousing systems (e.g., Redbrick, Sybase, Arbor) focus on the query and analysis component, providing specialized index structures at the warehouse and extensive querying facilities for the end user. In this paper, on the other hand, we focus

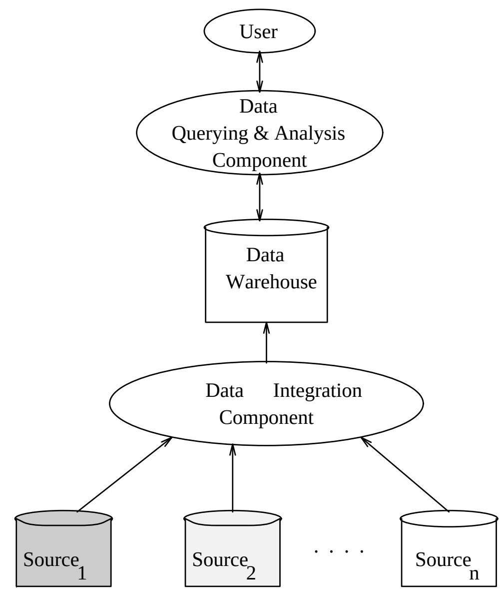

Figure 1: The basic architecture of a data warehousing system.

on the integration component. Specifically, we describe the architecture of a prototype system that collects data from heterogeneous sources, transforms and summarizes it according to warehouse specifications, and integrates it into the warehouse. This architecture has been implemented in the WHIPS (WareHouse Information Prototype at Stanford) System at Stanford. The Whips system is currently being used as a testbed for evaluating various integration schemes (as described briefly in Section 3).

We designed the Whips architecture to fulfill several important goals, all interrelated, as follows:

- *Plug-and-Play Modularity.* We clearly do not wish to have a system that only works with a specific warehouse or with particular types of sources, or that can only manage views in a specific way. On the contrary, the integration component should be composed of interchangeable *modules*, each providing some of the required functionality. For example,

*This work was supported by Rome Laboratories under Air Force Contract F30602-94-C-0237 and by an equipment grant from Digital Equipment Corporation.

a *warehouse wrapper* module is responsible for storing information into the warehouse, which could be any database system. If the target database system changes, we only need to change the warehouse wrapper module.

- *Scalability.* The integration component must deal with large amounts of data, coming from many sources. As the load grows, the system should scale gracefully by distributing its work among more machines and among more modules. For example, in our architecture, each materialized view is handled by a separate module. As the number of views grows, each view module can be run on a separate machine. Similarly, the system should support high degrees of concurrency, so that large numbers of updates can be processed simultaneously.

- 24×7 *Operation.* Many customers have international operations in multiple time zones, so there is no convenient down time, no "night" or "weekend" when new sources or views can be added and all of the recent updates can be batched and processed together to (re)compute materialized views. Thus, we should be able to add new sources and views to the system dynamically, and the integration component should be able to incrementally maintain the materialized views, without halting queries by end-users.

- *Data Consistency.* When data is collected from autonomous sources, the resulting materialized views may be inconsistent, e.g., they may reflect a state that never existed at a source [ZGMW95]. We would like a system that can avoid these problems, if it is important to the application. Thus, it should be possible to specify the desired level of consistency, and the system should support the necessary algorithms to achieve the different levels.

- *Support for Different Source Types.* Not all data sources are cooperative and willing to notify the warehouse when their data has changed. On the other hand, some sources do provide notification, e.g., by using trigger facilities. The integration component should be able to handle many different types of sources, and extract data from them in the most effective fashion. For example, to incrementally maintain a view based on data from an uncooperative source, the system should be capable of comparing database snapshots and extracting the differences.

The contribution of this paper is to show how the functionality required for integration can be decomposed into modules to achieve our desired goals, and to show how these modules then efficiently interact. Our solution is based on the notion of *distributed objects*, as in the CORBA model [Obj95, YD96]. Each module is implemented as a CORBA object that can run on any machine. Each object has a set of methods that can be called from other objects. In essence, our architecture and prototype system may be viewed as an experiment of CORBA's suitability for building information processing systems such as a data warehouse. Our experience indicates that distributed object technology, with the right architecture, is indeed very useful for providing the modularity and scalability required.

The remainder of paper is organized as follows. In Section 2, we overview the Whips architecture by showing the flow of messages that occurs among the modules during system startup, view creation, and view maintenance. In Section 3, we describe the modules and explain the design trade-offs we faced. We then go into more specific implementation details in Section 4. We present some preliminary performance results from our prototype in Section 5 and conclude in Section 6.

For an additional discussion of data warehouses and their research challenges, we refer the reader to [Wid95, HGMW+95]. These papers provide references to work upon which our system builds, for instance, in incremental view maintenance, data consistency for materialized views, and snapshot difference algorithms for identifying updates to legacy sources. Due to space limitations, we do not survey that work here.

2 Whips architecture

In Figure 2 we expand the integration component of Figure 1 to depict the Whips system architecture. As shown in the figure, the system is composed of many distinct modules that communicate with each other although they potentially reside on different machines. We implemented each module as a CORBA object, using the ILU implementation of CORBA [Xer95]. The commmunication between objects is then performed within the CORBA distributed object framework, where each object O has a unique identifer used by other objects to identify and communicate with O.

Using CORBA provides several benefits. First, CORBA hides the low-level communication so that the modules themselves are written independently of the communication; contacting another module is simply a method call. Second, CORBA guides all communication by the destination module's identifier rather than by its location. Therefore, it is easy to redistribute modules as the system scales.

In the current prototype, we use the relational model to represent the warehouse data: views are defined in the relational model and the warehouse stores relations. The underlying source data is converted to the relational model by the source's monitor and wrapper before it is sent to any other module. To simplify the presentation, we will discuss each source as if it contained only

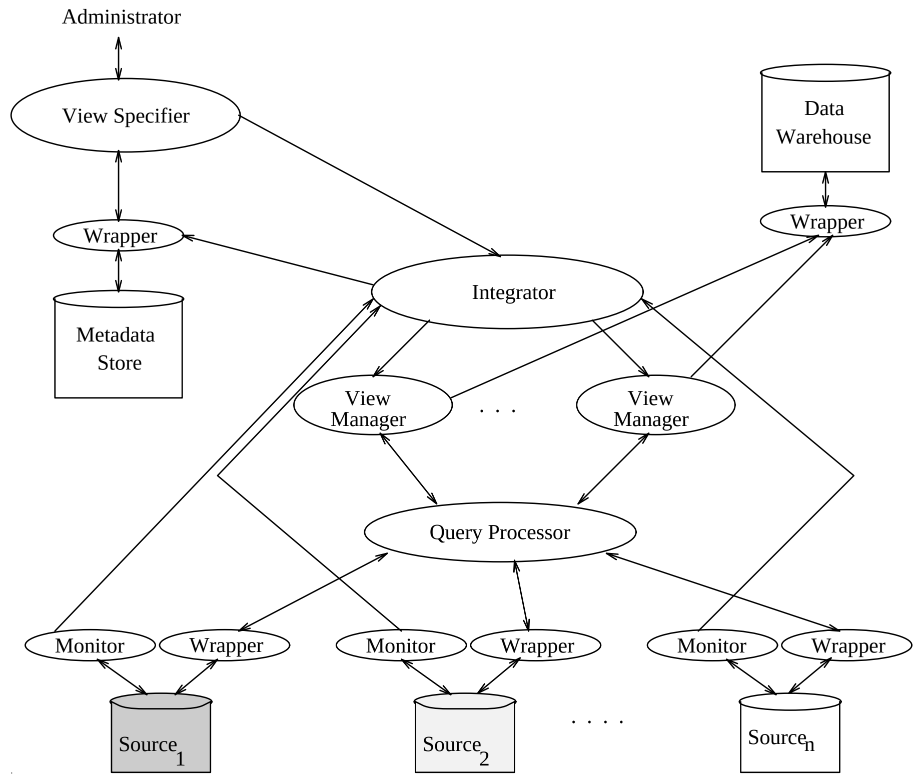

Figure 2: The Whips system architecture for warehouse maintenance.

a single "relation." In actuality, each source may contain multiple relations (or anything else, converted to relations), and modifications are detected separately for each of them.

We overview the modules of the architecture first by tracing the flow of messages in the Whips system. There are three distinct operations that each have their own flow of messages. First, at startup, the modules must identify themselves to each other. Similar actions also occur whenever a new source becomes available. Second, whenever a view is defined, the view is initialized and the system is primed to maintain the view. Third, each defined view is maintained (updated) in response to modifications that affect the view. Figure 2 shows the communication patterns during view definition and maintenance.

2.1 System initialization and source startup

At system startup, the integrator publishes its identifier and creates the query processor(s). All other starting modules contact the integrator. More specifically, the warehouse, meta-data store, and view specifier contact the integrator and identify themselves. Each source monitor and wrapper also contact the integrator to register the source meta-data, which is passed to the persistent meta-data store and query processor(s). Currently, there is one monitor and one wrapper per relation, implemented according to the source data type (see Sections 3.6 and 3.7). While we expect most sources to be reported at startup, sources may be added to the system at any time, by following the same procedure.

2.2 View definition and initialization

Views are defined at the view specifier by a system administrator. The view specifier type-checks each view definition with the meta-data store and then passes the view definition to the integrator, which spawns a view manager for that view. The integrator also notifies the monitors for all of the sources involved in the view to begin sending relevant modifications (if they were not already). The view manager is then responsible for initializing and maintaining the view. First, the view manager generates a (global) query corresponding to the view definition. It passes the query to a query processor, which contacts the query wrapper for each source involved in the view. The query processor joins the results returned to it by the query wrappers, and passes the (global) query answer back to the view manager. The view manager then sends the query answer to the warehouse wrapper to initialize the view.

2.3 View maintenance

Each monitor of a relation R detects the modifications to R that occur at its source (see Section 3.7) and forwards these modifications to the integrator. The integrator then forwards the modifications to all interested view managers (see Section 3.3). Each view manager then uses one of the *Strobe* algorithms for view consistency [ZGMW95] to compute the corresponding changes to the view at the warehouse. This computation may involve generating a (global) query, which is sent to the query processor and evaluated as at view initialization time. The returned query result is then adjusted as necessary by the view consistency algorithm and possibly held and combined with other query results. When the combined modifications will leave the view in a consistent state, the view manager sends the set of adjusted query results to the warehouse wrapper, which applies them to the warehouse view as a single transaction, bringing the view to a new consistent state.

2.4 Communication and message ordering

Communication messages are sent asynchronously during view maintenance, which means that delays in communication should not hold up the processing at any module. Note that in our architecture, messages sent from a source may arrive at a view manager by two paths. Modifications are sent from the monitor to the integrator to the view manager. Query results are sent from the wrapper to the query processor to the view manager. The architecture cannot guarantee that two messages sent by different paths will arrive in order, yet the view consistency algorithms require the view manager to know about all previous modifications when it receives a query result.

One possible solution is to also send query results via the integrator and to send modification synchronously from the integrator to the view manager. However, this would require both more messages and more expensive (synchronous) messages.

Our solution is to use sequence numbers, instead. Each monitor has its own sequence counter (per relation) and each modification is tagged with a sequence number when it is sent to the integrator. In addition, each wrapper tags its query results with the sequence number of the last modification sent by the corresponding monitor. The query processor builds an array of sequence numbers returned by the wrappers, one per relation, as part of each query result. The view manager also keeps an array of sequence numbers, one per relation, corresponding to the last modification it has received. When a query result arrives, the view manager then compares the query result array with its own array. If any query result sequence number is higher than the view manager's corresponding sequence number, the view manager waits for the modification before continuing. Note that this solution requires each single source query to receive a sequence number at least as high as any modification that may be reflected in the query result, and communication between the monitor and wrapper is involved. However, no special concurrency control is needed.

3 Whips modules

In this section, we describe the modules of the Whips architecture in more detail. For each module, we discuss the current implementation, design alternatives we considered, advantages of the current design, and extensions we would like to make. The modules are described below in roughly the order in which they are encountered during view definition and materialization.

3.1 View specifier

Views are defined in a subset of SQL that includes select-project-join views over all of the source data, without nesting. Optionally, the view definition may also specify which Strobe algorithm to use for view consistency. When a view is defined, the view specifier parses it into an internal structure we call the *view tree*, adds relevant information from the meta-data store (e.g., key attributes, attribute types), and sends the view tree to the integrator.

We are currently adding simple SQL aggregate operators (min, max, count, sum, and average) to the view language. We plan to add index specification capabilites for each view. We also plan to include the option of specifying that the view should include historical information (although the source data does not).

3.2 Meta-data store

The meta-data store keeps catalog information about the sources and how to contact them, the relations stored at each source, and the schema of each relation. The meta-data store also keeps track of all view definitions.

3.3 Integrator

The integrator coordinates both system startup, including new source additions, and view initialization. However, the main role of the integrator is to facilitate view maintenance, by figuring out which modifications need to be propagated to which views. To do so, the integrator uses a set of rules that specify which view managers are interested in which modifications. These rules are generated automatically from the view tree when each view is defined. In the simplest case, the rules dictate that all modifications to a relation over which a view is defined are forwarded to the corresponding view manager. Currently, the integrator is implemented as an index over the view managers, keyed by the relations. We would like to extend the integrator to filter the modifications for each view. For example, a selection condition in a view definition might render some modifications irrelevant to that view (although relevant to other views).

Although we have initially built the system with one integrator, an advantage of our design is that the integrator only depends on the view definitions. Therefore, the integrator can be replicated to scale the system. One integrator would be designated to spawn the view managers for each view definition, and to register the view managers with the other integrators.

3.4 View manager(s)

There is one view manager module responsible for maintaining each view, using one of the Strobe algorithms (as specified in the view definition) to maintain view consistency. The different Strobe algorithms yield different levels of consistency depending on the modification frequency and clustering; all of the algorithms require keeping track of the sequence of modifications and compensating query results for modifications that may have been missed. A full discussion of the algorithms may be found elsewhere [ZGMW95].

There are two advantages to using one view manager per view. First, the work of maintaining each view can be done in parallel on different machines. Second, each view may employ a different Strobe algorithm, to enforce a different level of consistency for its view.

3.5 Query processor(s)

The query processor is responsible for distributed query processing, using standard techniques such as sideways information passing and filtering of selection conditions [OV91] to prune the queries it poses to the wrappers. It tracks the state of each global query while waiting for local query results from the wrappers.

The primary advantages of separating the query processing from the view manager are that the view manager can generate global queries, unaware of the distributed sources; the query evaluation code, which is commmon to all of the Strobe algorithms, is only written once; and a single query processor can handle queries for many view managers. Because the wrappers hide the source-specific query syntax, the query processor generates single source queries as if the sources were relational databases.

Currently, the query processor waits for each single source query result from the wrapper before continuing. We are extending the query processor to work concurrently on evaluating multiple queries; while waiting for a query result from a given source, the query processor can then generate another single source query or apply a single source query result to a global query.

The architecture provides for multiple query processors as needed to handle the number of queries in the system. One design issue is then how each view manager chooses a query processor for each query. One option lets the view manager choose a query processor, either at random or with a hint from the integrator. However, a better alternative provides an additional module that exists purely to schedule queries to query processors. This scheme is most likely to scale to large numbers of view managers and queries. Note that multiple query schedulers could be added if needed, where each scheduler handles N query processors, and each view manager always sends its queries to a given query scheduler.

3.6 Wrappers

Each wrapper is responsible for translating single source queries from the internal relational representation used in the view tree (which resembles relational algebra) to queries in the native language of its source. For example, a relational database wrapper would merely translate the relational algebra expression into SQL. A wrapper for a flat file Unix source might translate the algebra expression into a Unix *grep* for one selection condition, use postprocessing to apply further selection conditions and projections, and then transform the result into a relation. As stated above, using one wrapper per source hides the source-specific querying details from the query processor and all other modules: all wrappers support the same method interface although their internal code depends on the source.

3.7 Sources and monitors

Each source may be completely autonomous of the warehouse and of the Whips system. However, we do take advantage of sources that are willing to cooperate (notify the system of changes) when we build monitors for them. Like the wrappers, the monitors all support a uniform method interface. However, their code differs according to the underlying source.

Each monitor detects the modifications that are performed (outside the Whips system) on its source data. These modifications are then sent to the integrator. Currently, we have implemented trigger-based monitors for cooperative (relational) sources, and snapshot monitors for flat file sources that only provide periodic snapshots of the source data. We describe algorithms for efficient change detection on snapshots elsewhere [LGM95]. We are working on adding IBM's DataCapture to the system; DataCapture is a log-based monitor which reads the log for DB2 and generates a table of source changes.

Currently, once a monitor is told that there is at least one view interested in the source, it notifies the integrator of all source modifications. However, we plan to enhance the monitors by filtering modifications based on selection conditions and projecting only relevant attributes (those involved in a selection condition, projection or join, or which are keys for the relation [GM95]) in the view definition. Note, though, that filters applied at the monitor must apply to all view definitions. View-specific filtering must be performed at the integrator.

3.8 Warehouse and warehouse wrapper

The warehouse in the Whips architecture may be any relational database. Of course, some relational databases are optimized for querying warehouse data, e.g. Redbrick [Red95], and may be more appropriate.

The warehouse wrapper receives all view definitions and all modifications to the view data in a canonical (internal) format, and translates them to the specific syntax of the warehouse database. The wrapper thus shields all other modules in the Whips system from the particulars of the warehouse, allowing any database to be used as the warehouse. All modifications received by the warehouse wrapper in a single message are applied to the warehouse in one transaction, as needed by the Strobe view consistency algorithms.

4 Whips implementation

All of the code is written in C++ and C, except the view parser portion of the view specifier, which is written in Lex and Yacc. We currently use a Sybase database [Syb92] for the warehouse. We have also experimented with a Sybase source with a monitor that uses triggers and a flat file source whose monitor uses the Windowing Snapshot algorithm [LGM95] to detect modifications. The Whips system currently runs on DEC Alphas and IBM RS/6000s. In the tests below, we used five separate machines for the modules: one for the integrator, view managers, and query processor, and one each for the warehouse wrapper, view specifier, Sybase source, and monitors and wrappers.

5 Performance

In this section, we present the results of preliminary performance experiments on the Whips prototype system. We performed two experiments, one to measure the system latency in propagating a single modification from a source to the warehouse, and one to measure the system throughput in propagating modifications.

For both experiments, the Whips system consisted of two sources containing one relation each. The `daily_stock` relation is a flat file containing a daily feed of stock prices from the NYSE and NASDAQ Stock Exchanges. The `monthly_pe` relation is a Sybase relation that provides the price-to-earnings (pe) ratio of each stock. (In the future, the pe's will be obtained from a Dialog source [Dia94] for this application.) The two relations are defined as follows, where the italicized attributes are the keys:

> `daily_stock(`*ticker*`, date, high, low, volume, close)`
> `monthly_pe(`*ticker*`, pe)`

Two views were defined for the experiments, a *Copy* view that was a copy of the `daily_stock` relation, and a *Join2* view that joined the two relations on the `ticker` attribute, as follows.

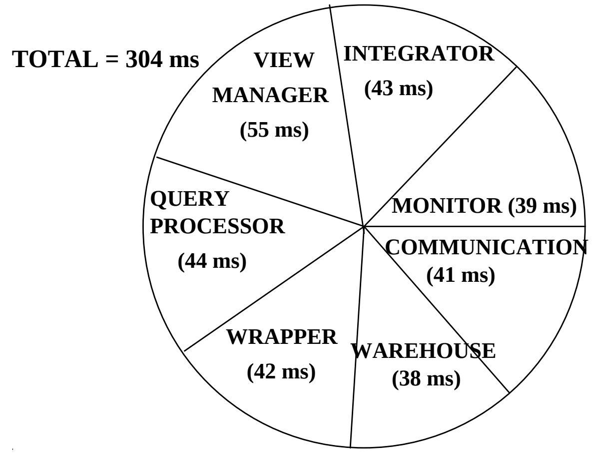

Figure 3: Time spent in each module while maintaining a view.

```
define view Copy as
select *
from daily_stock

define view Join2 as
select daily_stock.ticker, daily_stock.date,
  daily_stock.close, daily_stock.volume,
  monthly_pe.pe
from daily_stock, monthly_pe
where daily_stock.ticker=monthly_pe.ticker
  and monthly_pe.pe > 119.5
```

5.1 System latency

In the first experiment, we measured the system latency in propagating a single detected modification from the monitor to the *Join2* view at the warehouse. We simulated insertions to the `daily_stock` relation and recorded the time spent by the Whips system in each module in processing that one insert. We waited for a steady state and recorded the time for each module for 20 insertions. The average time spent in each module is shown in Figure 3, for a total time of 304 ms. The communication time is the portion of the total time not spent in any module.

As shown in the figure, a roughly equal amount of time is spent in each module. Therefore, no one module should be a bottleneck for propagating modifications in the system.

Although for these experiments, we used small versions of the relations containing 150 rows each, when we ran the experiment with larger versions of the relations (over 10,000 rows each), only the time at the monitors and wrappers increased: it takes slightly longer to detect the change and slightly longer to find join matches for it. The total time was therefore 340 ms, about 11% slower.

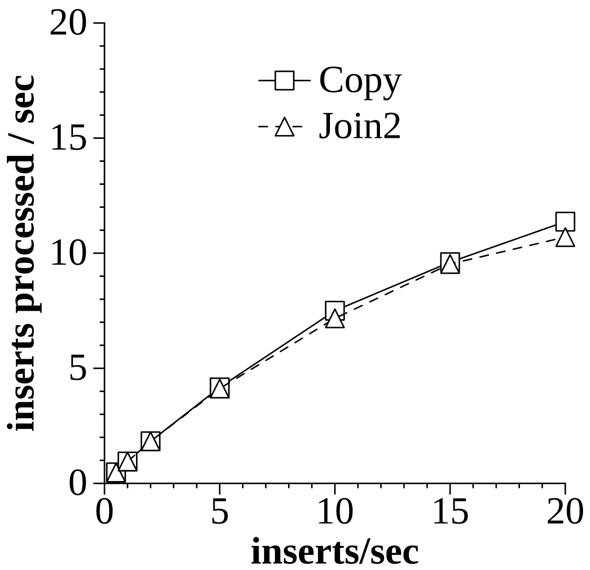

Figure 4: Arrival rate of modifications at the warehouse.

5.2 System throughput

In the second experiment, we measured the system throughput. We varied the number of modifications at the source per second from 1 to 20, and measured how many modifications appeared at the warehouse per second, for both the *Copy* and *Join2* views. (Twenty modifications per second is roughly 1.8 million modifications per day.) Each modification was an insert into the relation `daily_stock`. We ran the experiment for two minutes. Figure 4 shows that, as expected, as we increase the insertion rate, the Whips system processes more total modifications, but a smaller percentage of the total.

While a latency of 304 ms might predict processing only 3 inserts per second, since the modules' processing time can overlap, we expected a throughput inversely proportional to the slowest module, the view manager, of roughly 18 inserts per second (1000/55). However, in the current implementation, the query processor waits for each local query result from the wrapper before continuing. Therefore, the maximum throughput is inversely proportional to the time for the query processor and wrapper combined, or 11.6 inserts per second. The maximum we observed was 11.3; when more inserts were sent by the monitor, they generated longer and longer queues at the other modules.

This experiment shows that the throughput is as good as the slowest module. Therefore, by replicating the modules, each replica can handle as much work and the system can scale to handle larger modification rates and more defined views. For example, in the above scenario, we could add more query processor modules to handle the heavy query workload, and also extend the query processor to handle additional queries while waiting for query results from the wrappers.

6 Conclusions and future work

In this paper, we described the Whips architecture for warehouse creation and maintenance. The Whips system allows views over multiple, heterogeneous, autonomous, sources and provides incremental view maintenance in a modular and scalable fashion. The Whips system can thus grow while continuing to consistently update all defined views and to allow concurrent querying and analysis at the warehouse.

Future work on the Whips system includes adding foreign functions to the view definitions, to translate different representations of data into comparable formats (e.g., dollars to yen) and filtering modifications at the integrator so that view managers are only informed of modifications relevant to their view (not simply all modifications to relations in the view). We are also designing algorithms for crash recovery; in order to recover from a crash, not only do all source and view definitions need to be persistent (they already are), but also all modifications currently being processed must be remembered and recovered.

We also plan to do more performance testing and tuning of the prototype system. Adding system statistics could be of great benefit. For instance, usage statistics of the views defined could help decide how often the view should be updated. Query processor and integrator load statistics could help in load balancing.

Finally, we are interested in keeping track of the relationships among views and using them to make view maintenance more efficient. In the examples in this paper, it was always necessary to examine the source data to update each view. However, some views may be *self-maintainable* [QGMW95], possibly by querying other views stored at the warehouse rather than the sources.

References

[Dia94] Dialog Information Services, Inc. *Dialog Pocket Guide 1994*, 1994.

[GM95] A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin*, 18(2):4–19, June 1995.

[HGMW+95] J. Hammer, H. Garcia-Molina, J. Widom, W. Labio, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin*, 18(2):41–48, June 1995.

[LGM95] 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*, 1995. To appear.

[Obj95] Object Management Group (OMG), Framingham, MA. *The Common Object Request Broker: Architecture and Specification*, July 1995.

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

[QGMW95] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. Technical report, Stanford University, 1995.

[Red95] Red Brick Systems. *Red Brick Warehouse*, 1995.

[Syb92] Sybase, Inc. *Command Reference Manual*, release 4.9 edition, 1992.

[Wid95] J. Widom. Research Problems in Data Warehousing. In *Conference on Information and Knowledge Management*, 1995. Also http://db.stanford.edu/pub/widom/1995/warehouse-research.ps.

[Xer95] Xerox Corp. *ILU Reference Manual*, March 1995.

[YD96] Z. Yang and K. Duddy. CORBA: A Platform for Distributed Object Computing. *Operating Systems Review*, 30(2):4–31, April 1996.

[ZGMW95] Y. Zhuge, H. Garcia-Molina, and J. L. Wiener. The Strobe Algorithms for Multi-Source Warehouse Consistency. Technical report, Stanford University, 1995. Also http://db.stanford.edu/pub/zhuge/1995/consistency.ps.