

RL-TR-97-72
Final Technical Report
August 1997



A WAREHOUSING APPROACH TO DATA AND KNOWLEDGE INTEGRATION

Stanford University

Jennifer Widom

PHOTO QUALITY REPRODUCED

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

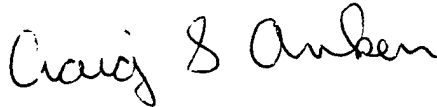
19970919 030

Rome Laboratory
Air Force Materiel Command
Rome, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-72 has been reviewed and is approved for publication.

APPROVED:



CRAIG S. ANKEN
Project Engineer

FOR THE DIRECTOR:



JOHN A. GRANIERO, Chief Scientist
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL/C3CA, 525 Brooks Rd, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE Aug 97	3. REPORT TYPE AND DATES COVERED Final Sep 94 - Aug 96		
4. TITLE AND SUBTITLE A WAREHOUSING APPROACH TO DATA AND KNOWLEDGE INTEGRATION			5. FUNDING NUMBERS C - F30602-94-c-0237 PE- 62702F PR- 5581 TA- 27 WU-79	
6. AUTHOR(S) Jennifer Widom				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Sponsored Projects Office Stanford University Stanford, CA 94305			8. PERFORMING ORGANIZATION REPORT NUMBER N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3CA 525 Brooks Rd. Rome, NY 13441-4505			10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-97-72	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Craig Anken/C3CA/(315) 330-4833				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) The topic of data warehousing encompasses architectures, algorithms, and tools for bringing together selected data from multiple databases or other information sources into a repository, called a data warehouse, suitable for direct querying or analysis. The objective of this effort was to provide a mechanism for rapidly and effectively integrating data and knowledge from multiple, diverse, information sources, and to provide a simple way for clients to locate, retrieve, and exploit the relevant integration information quickly and efficiently. Within this effort the goals were to develop the specification languages, algorithms, and techniques necessary to realize a warehousing approach to data and knowledge integration, to demonstrate the techniques in a software prototype, and to validate the utility of the approach with a suite of sample applications.				
14. SUBJECT TERMS Data Warehousing, Distributed Information, Data Integration, Artificial Intelligence			15. NUMBER OF PAGES 36	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

FINAL REPORT

A Warehousing Approach to Data and Knowledge Integration Stanford University

Principal Investigator: Prof. Jennifer Widom

Contents

1 Overall Report	3
1.1 Project Objective	3
1.2 Technical Accomplishments	3
1.2.1 Data and Knowledge Model	3
1.2.2 Source Specification Language	3
1.2.3 Monitoring and Change Propagation	4
1.2.4 Integration Specification Language	4
1.2.5 Information Mediator	4
1.2.6 Information Importer	5
1.2.7 System Evaluation	5
1.2.8 Additional tasks beyond the scope of the proposed project	5
1.3 Shortcomings	6
1.4 Web Pages, Demo, and Publications	6
1.4.1 Relevant publications	6
1.5 Future Work	7
2 Research Problems in Data Warehousing	8
2.1 Abstract	8
2.2 Introduction	8
2.3 Industrial Perspective	10
2.4 Architecture of a Data Warehousing System	10
2.5 Research Problems	11
2.5.1 Wrapper/Monitors	11
2.5.2 Integrator	13
2.5.3 Warehouse Specification	14
2.5.4 Optimizations	15
2.5.5 Miscellaneous	16
2.6 Conclusions	17
3 A System Prototype for Warehouse View Maintenance	17
3.1 Abstract	17

3.2	Introduction	18
3.3	Whips architecture	20
3.3.1	System initialization and source startup	21
3.3.2	View definition and initialization	21
3.3.3	View maintenance	22
3.3.4	Communication and message ordering	22
3.4	Whips modules	23
3.4.1	View specifier	23
3.4.2	Meta-data store	23
3.4.3	Integrator	23
3.4.4	View manager(s)	24
3.4.5	Query processor(s)	24
3.4.6	Wrappers	25
3.4.7	Sources and monitors	25
3.4.8	Warehouse and warehouse wrapper	25
3.5	Whips implementation	26
3.6	Performance	26
3.6.1	System latency	27
3.6.2	System throughput	27
3.7	Conclusions and future work	28

1 Overall Report

1.1 Project Objective

The objective of the effort was to provide a mechanism for rapidly and effectively integrating data and knowledge from multiple, diverse, information sources, and to provide a simple way for clients to locate, retrieve, and exploit the relevant integrated information quickly and efficiently. Within this effort the goals were to:

1. Develop the specification languages, algorithms, and techniques necessary to realize a warehousing approach to data and knowledge integration.
2. Demonstrate the techniques in a software prototype.
3. Validate the utility of the approach with a suite of sample applications.

1.2 Technical Accomplishments

The goals of the effort as stated above have largely been achieved. In many areas we went well beyond the tasks originally specified (see Section 1.2.8 below), although in some areas we did fall slightly short (see Section 1.3 below).

We first report on our technical accomplishments based on the task items in the statement of work for the effort. We then list additional accomplishments that went beyond the scope of the original proposal. Excerpts from the original statement of work appear indented in italics.

1.2.1 Data and Knowledge Model

Develop an appropriate data and knowledge model for the information warehouse.

After evaluation of several models we decided to use the relational model for our initial warehousing algorithms, tools, and prototype development. The relational model allowed us to exploit some existing tools and techniques and move directly to more intricate and interesting problems in warehouse creation and maintenance.

1.2.2 Source Specification Language

Develop a specification language for describing which data and knowledge from an information source should be replicated at the warehouse, and how that data and knowledge should be translated to the model developed in 1.2.1.

The specification language we chose is similar to the SQL view definition language. View definitions are augmented with the specification of which of the information sources data are to be extracted from, and which consistency algorithms should be used (see below). Our first prototype concentrated on simple select-project-join views. Our current prototype also handles aggregate functions and external predicates and functions.

1.2.3 Monitoring and Change Propagation

Design and implement techniques for monitoring and propagating changes of interest at an information source. Generic techniques as well as source-specific techniques will be developed.

A significant amount of effort went into this important area. We designed a very efficient change monitor for information sources that are files of tagged records. This change monitor is general-purpose in that it is useful for any information source that can provide its evolving information as a sequence of file snapshots. A large number of information sources fall under this category.

We also developed an efficient and expressive change monitor for information sources that export hierarchically structured data. This change monitor takes as input two tree-structured databases. It produces as output both an "edit script" and a "delta tree" that represent the differences between the two input trees. These change monitoring algorithms apply to information sources ranging from nested object databases to structured documents.

1.2.4 Integration Specification Language

Develop a specification language for describing how data and knowledge from multiple information sources should be integrated.

We decided to combine this specification language with the specification language developed for 1.2.2.

1.2.5 Information Mediator

Design and implement a prototype warehouse information mediator that integrates data and knowledge into a warehouse according to specifications in the language of 1.2.4. Generic techniques as well as techniques specific to certain sets of sources will be developed.

A significant amount of effort went into this important area. We focused primarily on generic techniques, since we found that we were able to develop general tools and rule-based techniques that applied directly to most sources and warehouse scenarios. We discovered interesting anomalies that can occur in warehouse integration, and we developed a family of algorithms to handle them (depending on the desired level of consistency). Correctness proofs are provided and we have performed both analytical and empirical performance analyses of our algorithms.

Our mediator has a fairly complex process architecture in order to decouple the different activities (change receipt and processing, query evaluation, anomaly compensation, etc.). We have found the architecture to be very flexible and efficient for our purposes so far.

1.2.6 Information Importer

Design and implement an information importer module that can be used to incorporate a new database or knowledge-base into the warehousing system.

Our information importer is based on the view specification language in 1.2.2 above and uses the multi-source query processor we developed as part of 1.2.3 and 1.2.5 above.

1.2.7 System Evaluation

Evaluate the complete system prototype with a suite of sample applications.

We have built a complete robust prototype and have done a preliminary set of performance evaluations using a diverse financial information scenario. In the initial stages of our prototype development we also experimented with university accounts data. Our prototype includes a Graphical User Interface built using HTML and accessible through the world-wide web. We performed a number of tests on the performance of our warehouse architecture: We measured the update rate the warehousing architecture can handle in accepting and processing information source updates, and determined the average delay incurred when propagating information source updates through the warehousing architecture to the warehouse.

1.2.8 Additional tasks beyond the scope of the proposed project

- Developed techniques for “self-maintainability” of warehouse views. A set of views is self-maintainable if changes to the base data from which the view is derived can be integrated into the warehouse without obtaining additional information from the sources. When the warehouse views are self-maintainable, the anomalies discussed above are avoided and the performance of warehouse maintenance increases dramatically.
- Addressed the design problem of deciding which views to materialize in the warehouse based on update performance requirements and expected query load. The problem is inherently exponential but we have developed effective polynomial heuristics.
- Developed techniques for graceful and correct crash recovery in a warehousing environment. The techniques are used to bring the warehouse to a consistent and correct state following the failure of the warehouse, an information source, or a process in the mediator architecture.
- Developed an algorithm for incrementally maintaining the data at a warehouse in the presence of large maintenance transactions. The algorithm operates without locking (which would slow down both update and query processing), and without bringing the warehouse off-line (which would limit availability).
- Developed a suite of algorithms for maintaining view-specific temporal information in the warehouse when the underlying information sources are non-temporal.

1.3 Shortcomings

While we intended to experiment with our warehousing approach on a wide variety of sample applications, to date we have focused primarily on a financial scenario and, earlier, on university accounts data. In the course of the effort we decided that it was more important to experiment with our approach on a wide variety of types and capabilities of information sources – which we have done – rather than a wide variety of application domains. We still intend to implement several application domains using our warehousing prototype, but we feel confident that our financial and accounts scenarios have exercised and demonstrated most of the functionality that will be needed for most domains.

1.4 Web Pages, Demo, and Publications

A set of web page describing our warehousing project (dubbed “WHIPS” for *Warehouse Information Processing at Stanford*) is located at the URL:

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

In addition to general technical information, the pages include personnel and relevant talks and publications.

A demo of our prototype warehousing system is available at the URL:

<http://halibut.stanford.edu:8000/start.html>

The user name is `dbgroup` and the password is `whips-demo`. It is suggested to contact the PI before trying the demo to ensure that all necessary Stanford servers are operational. Currently the demo lacks some user-friendly features, but we intend to add them soon.

A list of publications related to the warehousing project is available from the warehousing web page described above. A selected set of publications funded by this project is included below. All publications also are available by navigating from the Stanford Database Group's WWW home page: <http://www-db.stanford.edu>.

1.4.1 Relevant publications

Please note that publications 5 and 6 are included as subsequent sections ino this report.

1. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom. Change Detection in Hierarchically Structured Information. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 493–504, Montreal, Canada, June 1996.
2. J. Hammer, H. Garcia-Molina, W. Labio, J. Widom, and Y. Zhuge. The Stanford Data Warehousing Project. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):41–48, June 1995.

3. D. Quass, A. Gupta, I.S. Mumick, and J. Widom. Making Views Self-Maintainable for Data Warehousing. *Proceedings of the Fourth International Conference on Parallel and Distributed Information Systems (PDIS '96)*, Miami Beach, Florida, December 1996.
4. D. Quass and J. Widom. On-Line Warehouse View Maintenance for Batch Updates. To appear in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, May 1997
5. J. Widom. Research Problems in Data Warehousing. *Proceedings of the Fourth International Conference on Information and Knowledge Management (CIKM '95)*, pages 25–30, Baltimore, Maryland, November 1995.
6. J.L. Wiener, H. Gupta, W.J. Labio, Y. Zhuge, H. Garcia-Molina, and J. Widom. A System Prototype for Warehouse View Maintenance. *Proceedings of the 1996 Workshop on Materialized Views: Techniques and Applications*, pages 26–33, Montreal, Canada, June 1996.
7. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 316–327, San Jose, CA.

1.5 Future Work

As parts of follow-on projects we intend to pursue the following research directions related to the completed project:

- Evaluate our warehousing prototype with a suite of widely representative sample applications.
- Improve the usability for naive users of our web-based prototype demonstration.
- Implement and experiment with techniques we have developed for graceful and correct crash recovery in a warehousing environment.
- Incorporate into our prototype the warehouse self-maintainability algorithms described above.
- Incorporate into our prototype a technique we have developed for incrementally maintaining data in the warehouse in the presence of large maintenance transactions, without locking.
- Implement the algorithms we have developed for maintaining view-specific temporal information in the warehouse when the underlying information sources are non-temporal.

- Incorporate into our prototype update filtering algorithms at monitors and mediator to optimize performance of the warehousing infrastructure by minimizing data shipping and update processing.
- Extend the warehousing algorithms and prototype to operate in an object-oriented database environment, as well as in the relational database environment in which it now operates.

2 Research Problems in Data Warehousing

The material in this section appeared as an invited paper in the Fourth International Conference on Information and Knowledge Management (CIKM '95).

2.1 Abstract

The topic of data warehousing encompasses architectures, algorithms, and tools for bringing together selected data from multiple databases or other information sources into a single repository, called a *data warehouse*, suitable for direct querying or analysis. In recent years data warehousing has become a prominent buzzword in the database industry, but attention from the database research community has been limited. In this paper we motivate the concept of a data warehouse, we outline a general data warehousing architecture, and we propose a number of technical issues arising from the architecture that we believe are suitable topics for exploratory research.

2.2 Introduction

Providing integrated access to multiple, distributed, heterogeneous databases and other information sources has become one of the leading issues in database research and industry. In the research community, most approaches to the data integration problem are based on the following very general two-step process:

1. Accept a query, determine the appropriate set of information sources to answer the query, and generate the appropriate subqueries or commands for each information source.
2. Obtain results from the information sources, perform appropriate translation, filtering, and merging of the information, and return the final answer to the user or application (hereafter called the *client*).

We refer to this process as a *lazy* or *on-demand* approach to data integration, since information is extracted from the sources only when queries are posed. (This process also may be referred to as a *mediated* approach, since the module that decomposes queries and combines results often is referred to as a *mediator*.)

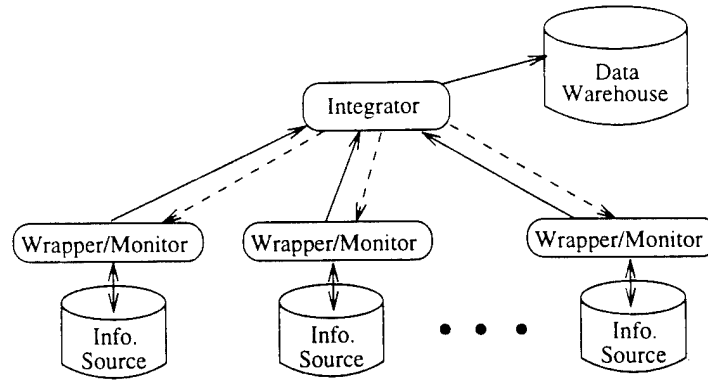


Figure 1: Basic architecture of a data warehousing system

The natural alternative to a lazy approach is an *eager* or *in-advance* approach to data integration. In an eager approach:

1. Information from each source that may be of interest is extracted in advance, translated and filtered as appropriate, merged with relevant information from other sources, and stored in a (logically) centralized repository.
2. When a query is posed, the query is evaluated directly at the repository, without accessing the original information sources.

This approach is commonly referred to as *data warehousing*, since the repository serves as a warehouse storing the data of interest.

A lazy approach to integration is appropriate for information that changes rapidly, for clients with unpredictable needs, and for queries that operate over vast amounts of data from very large numbers of information sources. However, the lazy approach may incur inefficiency and delay in query processing, especially when queries are issued multiple times, when information sources are slow, expensive, or periodically unavailable, and when significant processing is required for the translation, filtering, and merging steps. In cases where information sources do not permit ad-hoc queries, the lazy approach is simply not feasible.

In the warehousing approach, the integrated information is available for immediate querying and analysis by clients. Thus, the warehousing approach is appropriate for:

- clients requiring specific, predictable portions of the available information
- clients requiring high query performance (the data is available locally at the warehouse), but not necessarily requiring the most recent state of the information
- environments in which native applications at the information sources require high performance (large multi-source queries are executed at the warehouse instead)
- clients wanting access to private copies of the information so that it can be modified, annotated, summarized, and so on, or clients wanting to save information that is not maintained at the sources (such as historical information)

The lazy and warehousing approaches are each viable solutions to the data integration problem, and each is appropriate for certain scenarios.¹ The database research community has focused primarily on lazy approaches to integration. In this paper we consider research problems associated with the warehousing approach.

2.3 Industrial Perspective

Before considering the research problems associated with data warehousing, we note that there has been great interest in the topic within the database industry over the last several years. Most leading vendors claim to provide at least some “data warehousing tools,” while several small companies are devoted exclusively to data warehousing products. Despite rapid advances in commercial data warehousing tools and products, most of the available systems are relatively inflexible and limited in their features. We believe that a truly general, efficient, flexible, and scalable data warehousing architecture requires a number of technical advances, outlined below.

The importance of data warehousing in the commercial segment appears to be due to a need for enterprises to gather all of their information into a single place for in-depth analysis, and the desire to decouple such analysis from on-line transaction processing systems. Analytical processing that involves very complex queries (often with aggregates) and few or no updates—usually termed *decision support*—is one of the primary uses of data warehouses, hence the terms data warehousing and decision support often are found together, sometimes interchanged.² Since decision support often is the goal of data warehousing, clearly warehouses may be tuned for decision support, and perhaps vice-versa. Nevertheless, decision support is a very broad area, so we focus this paper specifically on research issues associated with the warehousing approach to integration.

2.4 Architecture of a Data Warehousing System

Figure 1 illustrates the basic architecture of a data warehousing system. The bottom of the diagram shows the information sources. Although the traditional disk shapes connote conventional database systems, in the general case these sources may include non-traditional data such as flat files, news wires, HTML and SGML documents, knowledge bases, legacy systems, and so on. Connected to each information source is a *wrapper/monitor*. The *wrapper* component of this module is responsible for translating information from the native format of the source into the format and data model used by the warehousing system, while the *monitor* component is responsible for automatically detecting changes of interest in the source data and reporting them to the integrator.

¹Another promising and relatively unexplored approach to information integration is a *hybrid* approach, in which some information is stored in a centralized repository while other information is fetched on demand.

²Other relevant terms include *data mining*, *on-line analytical processing (OLAP)*, and *multidimensional analysis*, which we view as refinements or subclasses of decision support.

When a new information source is attached to the warehousing system, or when relevant information at a source changes, the new or modified data is propagated to the *integrator*. The integrator is responsible for installing the information in the warehouse, which may include filtering the information, summarizing it, or merging it with information from other sources. In order to properly integrate new information into the warehouse, it may be necessary for the integrator to obtain further information from the same or different information sources. This behavior is illustrated by the downward dashed arrows in Figure 1.

The data warehouse itself can use an off-the-shelf or special purpose database management system. Although in Figure 1 we illustrate a single, centralized warehouse, the warehouse certainly may be implemented as a distributed database system, and in fact data parallelism or distribution may be necessary to provide the desired performance.

The architecture and basic functionality we have described is more general than that provided by most commercial data warehousing systems. In particular, current systems usually assume that the sources and the warehouse subscribe to a single data model (normally relational), that propagation of information from the sources to the warehouse is performed as a batch process (perhaps off-line), and that queries from the integrator to the information sources are never needed.

2.5 Research Problems

Based on the general architecture for data warehousing described in Section 2.4, we now outline a number of research problems that arise from the warehousing approach.

2.5.1 Wrapper/Monitors

The *wrapper/monitor* components illustrated in Figure 1 have two interrelated responsibilities:

1. **Translation:** Making the underlying information source appear as if it subscribes to the data model used by the warehousing system. For example, if the information source consists of a set of flat files but the warehouse model is relational, then the wrapper/monitor must support an interface that presents the data from the information source as if it were relational. The translation problem is inherent in almost all approaches to data integration—both lazy and eager—and is not specific to data warehousing. Typically, a component that translates an information source into a common integrating model is called a *translator* or *wrapper*.³
2. **Change detection:** Monitoring the information source for changes to the data that are relevant to the warehouse and propagating those changes to the integrator. Note

³Most commercial data warehousing systems assume that both the information sources and the warehouse are relational, so translation is not an issue. However, some vendors do provide wrappers for other common types of information sources.

that this functionality relies on translation since, like the data itself, changes to the data must be translated from the format and model of the information source into the format and model used by the warehousing system.

One approach is to ignore the change detection issue altogether and simply propagate entire copies of relevant data from the information source to the warehouse periodically. The integrator can combine this data with existing warehouse data from other sources, or it can request complete information from all sources and recompute the warehouse data from scratch. Ignoring change detection may be acceptable in certain scenarios, for example when it is not important for the warehouse data to be current and it is acceptable for the warehouse to be off-line occasionally. However, if currency, efficiency, and continuous access are required, then we believe that detecting and propagating changes and incrementally folding the changes into the warehouse will be the preferred solution.

In considering the change detection problem, we have identified several relevant types of information sources:

- **Cooperative sources:** Sources that provide triggers or other *active database* capabilities, so that notifications of changes of interest can be programmed to occur automatically.
- **Logged sources:** Sources maintaining a log that can be queried or inspected, so changes of interest can be extracted from the log.
- **Queryable sources:** Sources that allow the wrapper/monitor to query the information at the source, so that periodic polling can be used to detect changes of interest.
- **Snapshot sources:** Sources that do not provide triggers, logs, or queries. Instead, periodic dumps, or *snapshots*, of the data are provided off-line, and changes are detected by comparing successive snapshots.

Each type of information source capability provides interesting research problems for change detection. For example, in cooperative sources, although triggers and active databases have been explored in depth, putting such capabilities to use in the warehousing context still requires addressing the translation aspect; similarly for logged sources. In queryable sources, in addition to translation, one must consider performance and semantic issues associated with polling frequency: If the frequency is too high, performance will degrade, while if the frequency is too low, changes of interest may not be detected in a timely way. In snapshot sources, the challenge is to compare very large database dumps, detecting the changes of interest in an efficient and scalable ways. is to develop appropriate representations for the changes to the data, especially if a non-relational model is used.

Finally, we note that a different wrapper/monitor component is needed for each information source, since the functionality of the wrapper/monitor is dependent on the type of the source (database system, legacy system, news wire, etc.) as well as on the data provided by

that source. Clearly it is undesirable to hard-code a wrapper/monitor for each information source participating in a warehousing system, especially if new information sources become available frequently. Hence, a significant research issue is to develop techniques and tools that automate or semi-automate the process of implementing wrapper/monitors, through a toolkit or specification-based approach.

2.5.2 Integrator

Assume that the warehouse has been loaded with its initial set of data obtained from the information sources. (The task of setting up and loading the data warehouse is discussed in Section 2.5.5 below.) The ongoing job of the integrator is to receive change notifications from the wrapper/monitors for the information sources and reflect these changes in the data warehouse: see Figure 1.

At a sufficiently abstract level, the data in the warehouse can be seen as a *materialized view* (or set of views), where the *base data* resides at the information sources. Viewing the problem in this way, the job of the integrator is essentially to perform *materialized view maintenance*. Indeed, there is a close connection between the view maintenance problem and data warehousing. However, there are a number of reasons that conventional view maintenance techniques cannot be used, and each of these reasons highlights a research problem associated with data warehousing:

- In most data warehousing scenarios, the views stored at the warehouse tend to be more complicated than conventional views. For example, even if the warehouse and the information sources are relational, the views stored in the warehouse may not be expressible using a standard relational view definition language (such as SQL) over the base data. Typically, data warehouses may contain a significant amount of historical information (e.g., the history of stock prices or retail transactions), while the underlying sources may not maintain this information. Hence, warehouse views may not be functions of the underlying base data as traditional views are, but rather functions of the history of the underlying data. Relevant areas of research here certainly include temporal databases, as well as work on efficient monitoring of historical information.
- Data warehouses also tend to contain highly aggregated and summarized information. Although in some cases aggregations may be describable in a conventional view definition language, the expressiveness of aggregates and summary operators in such languages are limited, so more expressive view definition languages may be needed. Furthermore, efficient view maintenance in the presence of aggregation and summary information appears to be an open problem.
- The information sources updating the base data generally operate independently from the warehouse where the view is stored, and the base data may come from legacy systems that are unable or unwilling to participate in view maintenance. Most materialized view

maintenance techniques rely on the fact that base data updates are closely tied to the view maintenance machinery, and view modification occurs within the same transaction as the updates. In the warehousing environment it is generally the case that:

- The system maintaining the view (the integrator) is only loosely coupled to the systems handling the base data (the information sources).
- The underlying information sources do not participate in view maintenance but simply report changes.
- Some sources may not provide locking capabilities, and there are almost certainly no global transactions.

In this scenario, certain "anomalies" arise when attempting to keep views consistent with base data, and algorithms must be used that are considerably more complicated than conventional view maintenance algorithms.

- In a data warehouse, the views may not need to be refreshed after every modification or set of modifications to the base data. Rather, large batch updates to the base data may be considered, in which case efficient view maintenance techniques may involve different algorithms than are used for conventional view maintenance.
- In a data warehousing environment it may be necessary to transform the base data (sometimes referred to as *data scrubbing*) before it is integrated into the warehouse. Transformations might include, for example, aggregating or summarizing the data, sampling the data to reduce the size of the warehouse, discarding or correcting data suspected of being erroneous, inserting default values, or eliminating duplicates and inconsistencies.

Finally, we note that although integrators can be based purely on the data model used by the warehousing system, a different integrator still will be needed for each data warehouse, since a different set of views over different base data will be stored. As with wrapper/monitors, it is desirable not to require that each integrator be hard-coded from scratch, but rather to provide techniques and tools for generating integrators from high-level, non-procedural specifications. This general approach is standard practice in conventional view maintenance, however there are a number of interesting problems in adapting it to data warehousing, discussed in the next section.

2.5.3 Warehouse Specification

In the previous section we drew an analogy between maintenance of a data warehouse and materialized view maintenance. We also indicated that it is useful to provide capabilities for specifying integrators in a high-level fashion, rather than implementing each integrator from scratch. Hence, in an ideal architecture, the contents of the data warehouse are specified as a

set of view definitions, from which the warehouse updating tasks performed by the integrator and the change detection tasks required of the wrapper/monitors are deduced automatically.

For conventional view maintenance, algorithms have been developed to automatically generate active database rules for maintaining SQL-defined views. Each rule is “triggered” by the notification of an update that may affect the view, and the rule modifies the view appropriately. A similar approach may be applied to data warehousing if a rule-driven integrator is used. Each integrator rule is triggered by a change notification (possibly of a specific type) from a wrapper/monitor. Similar to the view maintenance rules, integrator rules must update the warehouse to reflect the base data updates. However, in the warehousing scenario, rules may need to perform more complicated functions, such as fetching additional data from sources using remote queries and “scrubbing” the data (as described in Section 3.4.3). Despite the additional complexity of rules in the warehousing environment, it still should be possible to automatically or semi-automatically generate appropriate rules from the warehouse (view) specification.

Thus, the research challenge in realizing the ideal architecture is to devise a warehouse specification language, rule capabilities, wrapper/monitor interfaces, and appropriate algorithms to permit developers of a data warehousing system to generate the integrator and the relevant change detection mechanisms automatically. We self-servingly note that this approach is being pursued by the *WHIPS* data warehousing project at Stanford.

2.5.4 Optimizations

In this section we outline three optimizations that can improve the performance of the architecture described Section 2.4: filtering irrelevant modifications at the sources, storing additional data at the warehouse for “self-maintainability,” and efficiently managing multiple materialized views.

Update Filtering We have said that all data modifications at a source that may be relevant to the warehouse are propagated to the integrator by the wrapper/monitor. Returning to our view maintenance analogy and considering the relational case as an example, we would propagate all inserts, deletes, and updates on any relation that participates in a view at the warehouse. A number of papers have been devoted to the topic of determining when certain modifications are guaranteed to leave a view unchanged. Related techniques allow distributed integrity constraints to be checked at a single site when certain types of modifications occur. We believe that these classes of techniques can be adapted to data warehousing, whereby as many changes as possible are filtered at the source rather than propagated to the integrator.

Self-maintainability When the integrator receives a change notification, in order to integrate that change into the warehouse the integrator may need to fetch additional data from the same or different sources. (As a simple example, if the warehouse joins two relations R

and S , and there is a notification of an insert to relation R , then the inserted tuple must be joined with the contents of relation S .) Issuing queries to sources can incur a processing delay, the queries may be expensive, and such queries are the basis of the warehouse maintenance “anomalies” alluded to in Section 3.4.3. Even worse, when information sources are highly secure or when they are legacy systems, ad-hoc queries may not be permitted at all. Consequently, it may be desirable to ensure that, as much as possible, queries to the sources are not required in order to keep the warehouse data consistent.

In view maintenance, when additional queries over base data are never required to maintain a given view, then the view is said to be *self-maintainable*. Most views are not fully self-maintainable. However, self-maintainability can be ensured by storing additional data at the warehouse. For example, in the extreme case, all relevant data from the sources is copied to the warehouse, and views can be recomputed in their entirety if necessary. It appears to be an open research problem to determine the minimum amount of extra information needed for self-maintainability of a given view. Also interesting is to balance the cost of maintaining extra data at the warehouse against the cost of issuing queries to the sources.

Multiple View Optimization Data warehouses may contain multiple views, for example to support different types of analysis. When these views are related to each other, e.g., if they are defined over overlapping portions of the base data, then it may be more efficient not to materialize all of the views, but rather to materialize certain shared “subviews,” or portions of the base data, from which the warehouse views can be derived. When applicable, this approach can reduce storage costs at the warehouse and can reduce the effort required to integrate base data modifications into the warehouse. However, these savings must be balanced against slower query response at the warehouse, since some views may not be fully materialized.

2.5.5 Miscellaneous

We briefly note a few other important issues that arise in a data warehousing environment.

- **Warehouse management:** We have focused primarily on problems associated with the “steady state” of a data warehousing system. However, issues associated with warehouse design, loading, and metadata management are important as well. (In fact, it is these problems that have received the most attention from a large segment of the data warehousing industry to date.)
- **Source and warehouse evolution:** A warehousing architecture must gracefully handle changes to the information sources: schema changes, as well as the addition of new information sources and the removal of old ones. In addition, it is likely that clients will demand schema changes at the warehouse itself. All of these changes should be handled with as few disruptions or modifications to other components of the warehousing system as possible.

- **Duplicate and inconsistent information:** As in any environment involving multiple, heterogeneous information sources, there is the likelihood of encountering copies of the same information from multiple sources (represented in the same or different ways), or related information from multiple sources that is inconsistent. Earlier, we described the “scrubbing” of data from single sources. In addition, it is desirable for the integrator to scrub multi-source data, in order to eliminate duplicates and inconsistencies as much as possible.
- **Outdated information:** A feature of data warehouses is that they may contain historical information even when that information is not maintained in the sources. Nevertheless, in many cases it is undesirable to keep information “forever.” Techniques are needed for specifying recency requirements in a warehousing environment, and for ensuring that outdated information is automatically and efficiently purged from the warehouse.

2.6 Conclusions

In the area of integrating multiple, distributed, heterogeneous information sources, data warehousing is a viable and in some cases superior alternative to traditional research solutions. Traditional approaches request, process, and merge information from sources when queries are posed. In the data warehousing approach, information is requested, processed, and merged continuously, so the information is readily available for direct querying and analysis at the warehouse.

Although the concept of data warehousing already is prominent in the database industry, we believe there are a number of important open research problems, described above, that need to be solved to realize the flexible, powerful, and efficient data warehousing systems of the future.

3 A System Prototype for Warehouse View Maintenance

The material in this section appeared in the 1996 Workshop on Materialized Views: Techniques and Applications.

3.1 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

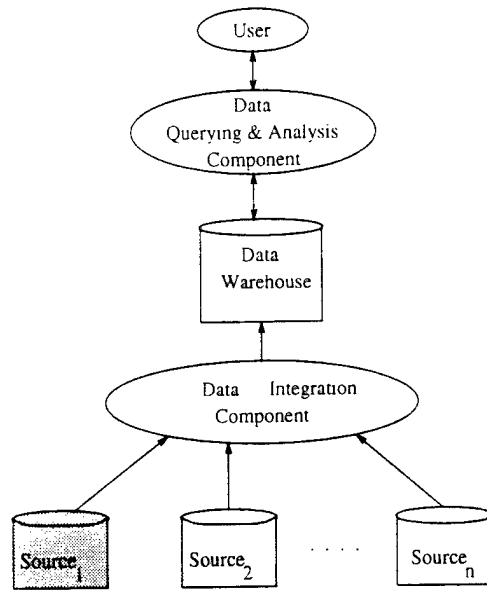


Figure 2: The basic architecture of a data warehousing system.

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.

3.2 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 2 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 2, 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 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 3.4).

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, 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. 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. 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 3.3, 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 3.4, we describe the modules and explain the design trade-offs we faced. We then go into more specific implementation details in Section 3.5. We present some preliminary performance results from our prototype in Section 3.6 and conclude in Section 3.7.

3.3 Whips architecture

In Figure 3 we expand the integration component of Figure 2 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. The communication between objects is then performed within the CORBA distributed object framework, where each object O has a unique identifier 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 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

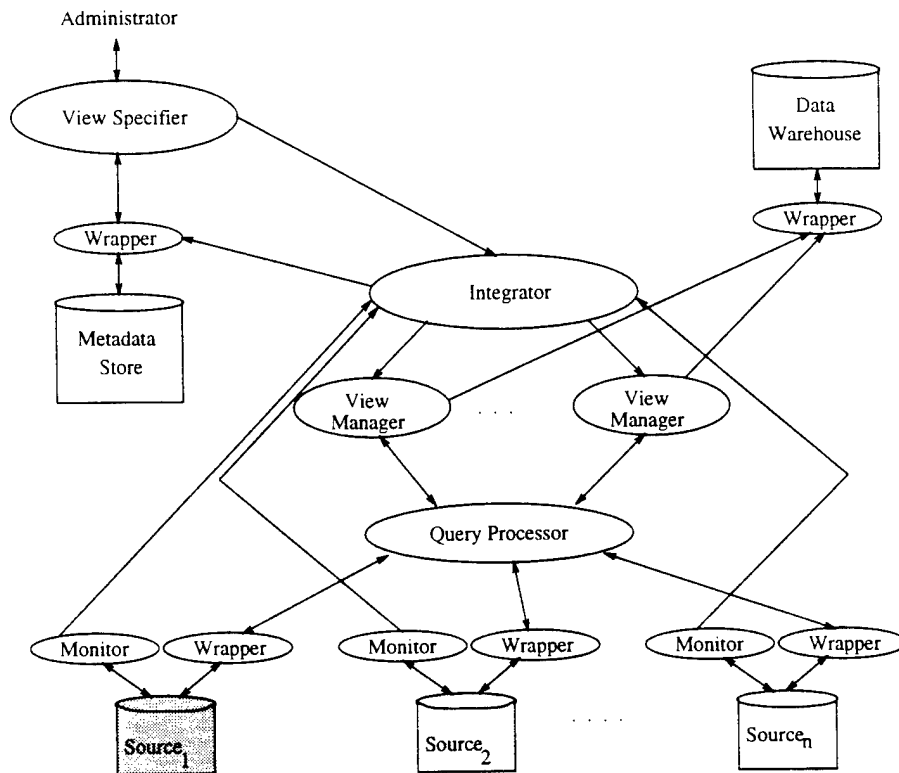


Figure 3: The Whips system architecture for warehouse maintenance.

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 3 shows the communication patterns during view definition and maintenance.

3.3.1 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 3.4.6 and 3.4.7). 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.

3.3.2 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.

3.3.3 View maintenance

Each monitor of a relation R detects the modifications to R that occur at its source (see Section 3.4.7) and forwards these modifications to the integrator. The integrator then forwards the modifications to all interested view managers (see Section 3.4.3). Each view manager then uses one of the *Strobe* algorithms for view consistency 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.

3.3.4 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.

3.4 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.

3.4.1 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 capabilities 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).

3.4.2 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.

3.4.3 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.

3.4.4 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.

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.

3.4.5 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 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 common 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.

3.4.6 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.

3.4.7 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 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) 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.

3.4.8 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, 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.

3.5 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 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 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.

3.6 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 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.

```
define view Copy as
select *
from daily_stock

define view Join2 as
select daily_stock.ticker, daily_stock.date,
```

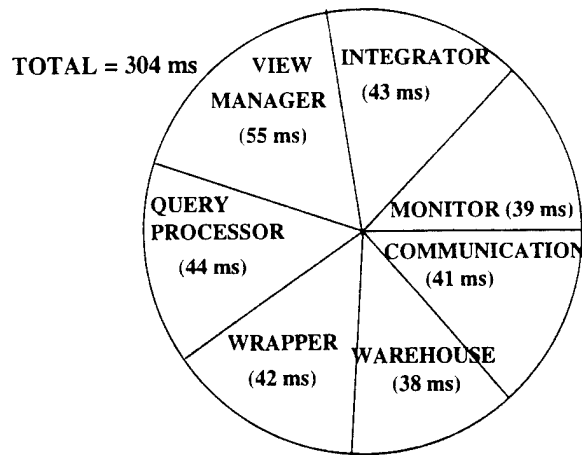


Figure 4: Time spent in each module while maintaining a view.

```

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

```

3.6.1 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 4, 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.

3.6.2 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 mod-

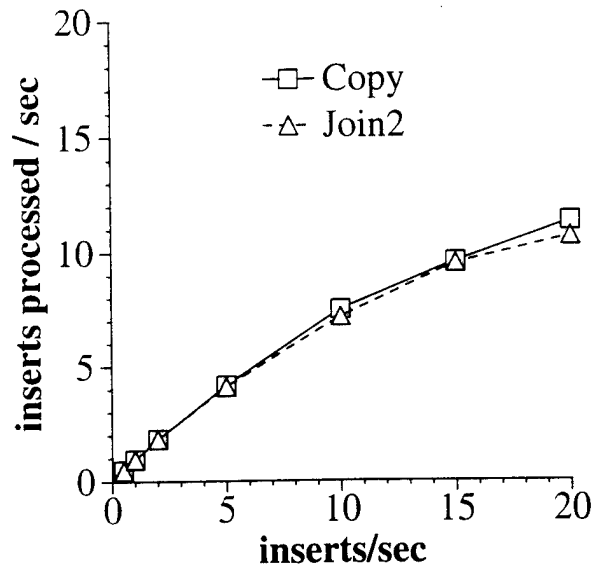


Figure 5: Arrival rate of modifications at the warehouse.

ifications 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 5 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.

3.7 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 concur-

rent 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*, possibly by querying other views stored at the warehouse rather than the sources.

**MISSION
OF
ROME LABORATORY**

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Material Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.