The Stanford Data Warehousing Project

Joachim Hammer, Hector Garcia-Molina, Jennifer Widom, Wilburt Labio, and Yue Zhuge
Computer Science Department
Stanford University
Stanford, CA 94305
E-mail: joachim@cs.stanford.edu

Abstract

The goal of the data warehousing project at Stanford (the WHIPS project) is to develop algorithms and tools for the efficient collection and integration of information from heterogeneous and autonomous sources, including legacy sources. In this paper we give a brief overview of the WHIPS project, and we describe some of the research problems being addressed in the initial phase of the project.

1 Introduction

A *data warehouse* is a repository of integrated information, available for querying and analysis [13, 14]. As relevant information becomes available or is modified, the information is extracted from its source, translated into a common model (e.g., the relational model), and integrated with existing data at the warehouse. At the warehouse, queries can be answered and data analysis can be performed quickly and efficiently since the information is directly available, with model and semantic differences already resolved. Furthermore, warehouse data can be accessed without tying up the information sources (e.g., holding locks or slowing down processing), accessing data at the warehouse does not incur costs that may be associated with accessing data at the information sources, and warehouse data is available even when the original information source(s) are inaccessible.

The key idea behind the data warehousing approach is to extract, filter, and integrate relevant information in *advance* of queries. When a user query arrives, the query does not have to be translated and shipped to the original sources for execution (as would be done in, e.g., a *mediated* approach to information integration [19].) Not only can such translation and shipping be a complex operation, but it also can be time consuming, especially if the sources are many and remote. Thus, warehousing may be considered an "active" or "eager" approach to information integration, as compared to the more traditional "passive" approaches where processing and integration starts when a query arrives. Warehousing also simplifies metadata management, and it provides a trusted and long-term repository for critical data that is under the control of the end-user.

One potential drawback of the warehousing approach is that data is physically copied from the original sources, thus consuming extra storage space. However, given dropping storage prices and the fact that data can be filtered or summarized before it is warehoused, we do not believe this is a serious problem. A more significant problem is that copying data introduces potential inconsistencies with the sources—warehouse data may become out of date. Another potential drawback is that the "warehouse administrator" must specify in advance what sites data should be extracted from, and which data should be copied and integrated. For these reasons, the data warehousing approach may not be appropriate when absolutely current data is required, or when clients have unpredictable needs.

In reality, we believe that data warehousing should be seen as a complement, not a replacement, to passive query processing schemes, or to ad-hoc exploration and discovery mechanisms [2, 11]. For example, ad-hoc discovery mechanisms can identify information of interest, which can can be collected at the warehouse, improving and simplifying access to the information.

The following examples suggest some of the application areas or characteristics for which the warehousing approach to information integration seems well suited.

1. *Collecting scientific data.* In these applications, large amounts of heterogeneous data are created so rapidly that real-time query processing becomes impossible. Furthermore, the sources may be sporadic and unreliable, so that warehousing the data in a safe and convenient location for later processing is appropriate.

2. *Maintaining historical enterprise data.* Processing and mining enterprise data (e.g., computing the sales history of all stores of a large supermarket chain over a certain period of time) is a resource-intensive job that is better performed off-line so as to not affect the day-to-day operations of the business. Thus it is desirable to move historic enterprise data away from the mainstream transaction processing systems where the data is created into an enterprise-owned data warehouse.

3. *Caching frequently requested information.* By storing in the data warehouse previously fetched and integrated answers to frequently asked queries, the inherent disadvantages of federated database systems (e.g., inefficiency, delay in query processing, etc.) can be overcome, resulting in improved performance and efficiency.

In this paper we present a brief overview of the data warehousing project we are undertaking at Stanford, called WHIPS (for "WareHouse Information Project at Stanford"). The goal of the WHIPS project is to develop algorithms and tools for the efficient collection and integration of information from heterogeneous and autonomous sources. Because the project is quite new—it began officially in January 1995—our focus in this paper is on the overall architecture of the system (Section 3), and on some of the specific research issues that we have investigated to date (Section 4).

2 Related Research

An introduction to data and knowledge warehousing is presented in reference [15]. In this book, the advantages of the warehousing approach are put forth and a general architecture and suggested functionality are presented. There has been a significant amount of research in the database and knowledge-base community related to the problem of integrating heterogeneous database and knowledge bases; representative literature is [4, 6, 10, 16, 17, 18, 19, 20]. Much of this work uses the "passive" approach discussed in the introduction. Some of the approaches rely on modifying the individual databases to conform to a "global schema." Despite these differences with our project, there are still a number of similarities, e.g., translating heterogeneous data into a common model, merging data from multiple sources, propagating data from source to target databases, etc. Hence, we are adapting methods from the heterogeneous database literature to the mediation and integration aspects of our warehousing approach.

If one considers the data residing in the warehouse as a *materialized view* over the data in the individual information sources, then it may be possible to adapt previously devised algorithms for *view maintenance* to the problem of change propagation and warehouse updating in the data warehousing environment. Two important differences with the traditional view maintenance problem are (1) the heterogeneity of the sources, and (2) the autonomy of the sources. The second problem implies that sources may not fully cooperate in view management, which leads to problems discussed in Section 4.3

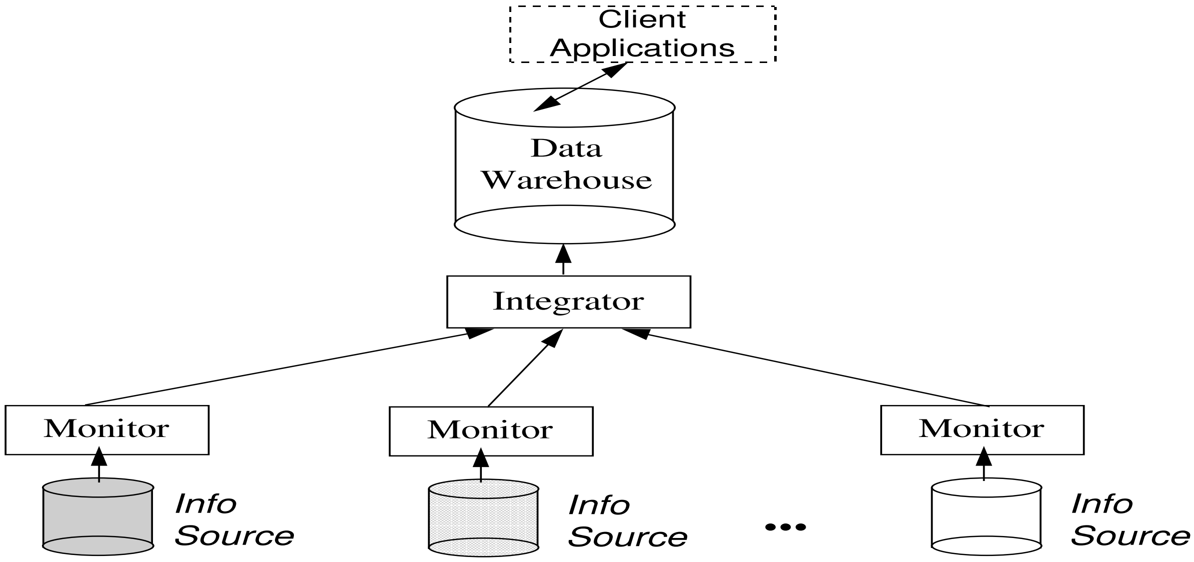

Figure 1: Architecture

below. Previously devised algorithms for view maintenance can be found in [1, 3, 5], and we have built upon these in our solutions.

The problem of monitoring individual databases to detect relevant changes is central to the area of *active databases*, as represented in, e.g., [7, 9, 12]. We are exploiting appropriate parts of this technology in the WHIPS project.

3 Architecture

Figure 1 illustrates the basic architecture of our system. The bottom of the figure depicts the multiple *information sources* of interest. These sources could include imagery, video, and text data, along with more traditional data sources such as relational tables, object-oriented structures, or files. Data that is of interest to the client(s) is copied and integrated into the *data warehouse*, depicted near the top of the figure.[1] Between the sources and the warehouse lie the (*source*) *monitors* and the *integrator*. The monitors are responsible for automatically detecting changes in the source data. Whenever the source content changes, the new or updated information that is relevant to the warehouse is propagated to the integrator. More details on how monitors operate and how they can be implemented are provided in Section 4.1.

The integrator is responsible for bringing source data into the warehouse. Integrating heterogeneous data into an existing schema is a difficult problem requiring several steps. First, the data must be made to conform to the conceptual schema used by the warehouse. Then the data must be merged with existing data already present, in the process resolving inconsistencies that might exist between the source and warehouse data. In essence, the integrator is where the differences across information sources are specified, where relationships among data at multiple sources are defined, where duplicates and inconsistencies are detected, and where it is determined how information will be integrated into the warehouse. More details on our approach to integration are provided in Section 4.2.

[1] If feasible, the client(s) may initially choose to replicate *all* of the data in order to get an overview of what is available before making a choice.

At the top of the figure are the *client applications* that let users interact with the warehouse. Currently we plan to provide only the basic query capabilities offered by any "off-the-shelf" database system, but additional capabilities (e.g., decision support, data mining, etc.) can be added at a later point if necessary.

A number of features of our planned architecture are not depicted explicitly in Figure 1. For example, although in the figure it appears that information flow is exclusively "upwards" (from information sources through monitors to the integrator and finally to the warehouse), in reality it often becomes necessary for the integrator to submit queries to one or more information sources in order to correctly integrate new or updated information (see Sections 4.2 and 4.3). Also note that the figure depicts *run-time* aspects of the system only. It is our ultimate goal to provide, in addition, a powerful *compile-time* component. Using this component, the desired contents of the warehouse are specified by the warehouse administrator. From this specification, appropriate monitors and an appropriate integrator are generated in an automatic or semi-automatic fashion, leading to easy and fast system configuration and reconfiguration for a variety of client needs.

4 Currently Under Investigation

There are numerous challenges associated with realizing the architecture in Figure 1. In this section we briefly summarize the main issues we have been addressing in the initial phase of the project.

4.1 Monitors

Monitors detect changes to an information source that are of interest to the warehouse and propagate the changes in a generic format to the integrator. If the source is a full-functionality database system, it may be possible for the monitor to simply define an appropriate set of triggers and wait to be notified of changes. Another option may be for the monitor to examine the update log file and extract the changes of interest. In other cases, such as legacy systems, the monitoring task may be much harder. Typically, legacy systems do not have trigger or logging capabilities (or, if such capabilities are present, they are not available to an external component like the monitor). In these cases, there are two options:

- Every application that changes the source data is modified so it emits appropriate notification messages to the monitor. For example, the application program that creates a new patient record for a hospital database will send a message to the monitor giving the new patient's name, status, and so on. Clearly, application level notifications are not particularly desirable, but they may need to be used when the underlying storage system will not perform notifications itself.

- A utility program is written that periodically dumps the source data into a file, and the monitor compares successive versions of the file. Many legacy systems already have dump utilities, typically used for backup or copying of data. Although the dump file format is usually specific to the legacy system, it often is possible to describe its "schema" in a generic way. We refer to the problem of detecting modifications by comparing successive dumps as the *snapshot differential* problem. Again, this solution is not particularly desirable, but it may often need to be used in practice.

We are currently focusing on solutions to the snapshot differential problem. One simple version of this problem is stated as follows: Each snapshot is a semistructured file F of the form $\{R_1, R_2, \ldots, R_n\}$, where R_i denotes a row or record in the file. Each row is (logically) of the form $\langle K, B \rangle$, where K is a key value and B is an arbitrary value representing all non-key data in the record. Given two files F_1

(the old file) and F_2 (the new file), produce a stream of *change notifications*. Each change notification is of one of the following forms:

1. *updated*(K_i,B_1,B_2), indicating that the row with key K_i has value B_1 in file F_1 and value B_2 in file F_2.
2. *deleted*(K_i,B_1), indicating that there exists a row with key K_i and value B_1 in file F_1, but no row with key K_i in file F_2.
3. *inserted*(K_i,B_2), indicating that there exists a row with key K_i and value B_2 in file F_2, but no row with key K_i in file F_1.

Note that rows with the same key may appear in different places in files F_1 and F_2. The snapshot differential problem is similar to the problem of computing *joins* in relational databases (*outerjoins*, specifically), because we are trying to match rows with the same key from different files. However, there are some important differences, among them:

- For snapshots, we may be able to tolerate some rows that are not "properly" matched. For example, suppose files F_1 and F_2 are identical except F_1 contains the row with key K_1 at the beginning, while F_2 contains K_1 towards the end. A simple "sliding window" snapshot algorithm might see K_1 in F_1 and try to find it in the first N (say) rows of F_2. Not seeing it there, it will report a deleted row with key K_1. Later, the algorithm will find K_1 in the second file and will report it as an inserted row. The superfluous delete and insert may create unnecessary work at the warehouse, but will not cause an inconsistency and therefore may be acceptable. In exchange, we may be able to construct a much more efficient snapshot differential algorithm, using an approach that cannot be used for relational (outer-)join.
- For the snapshot problem, we may be able to tolerate efficient probabilistic algorithms. For instance, suppose the B fields are very large. Instead of comparing the full B fields to discover differences, we may hash them into relatively small *signatures*. Consequently, there will be a small probability that we will declare rows $\langle K_i, B_1\rangle$ and $\langle K_i, B_2\rangle$ to be identical because the signatures matched, even though $B_1 \neq B_2$. However, a few hundred bits for the signature makes this probability insignificant, and this approach lets us dramatically reduce the size of the structures used in matching and differentiating rows.
- The snapshot problem is a "continuous" problem, where we have a sequence of files, each to be compared with its predecessor. Thus, in comparing F_2 to F_1, we can construct auxiliary structures for F_2 that will be useful when we later compare F_3 to F_2, and so on.

We are currently experimenting with several solutions to the snapshot differential problem. Some of the algorithms are similar to traditional join algorithms such as *sort-merge join* and *hash join*. Others are similar to UNIX *diff* and may not produce the minimal set of change notifications. The performance of these algorithms depends largely on the characteristics of the snapshots, e.g., whether the files are ordered or almost ordered, whether the keys are unique, whether the modifications are restricted to inserts only, etc. We have implemented an initial, simple differential monitor. It takes as input a schema specification for the files to be compared. It then analyzes the snapshot files using a sort-merge algorithm, sending change notifications to the integrator. Based on this framework, we plan to implement and evaluate several of the other schemes discussed above.

4.2 Integrator

The integrator component receives change notifications from the monitors and must integrate the changes into the warehouse. We are developing an approach where the integrator is implemented as a rule-based engine. Each rule is responsible for handling one kind of change notification, and is implemented as an object-oriented method. The method is called (or "triggered") whenever a monitor generates a change notification of the appropriate type. The method body then performs the necessary processing to integrate the change into the warehouse. During this processing, the method may need to obtain additional data from the warehouse, or from the same or other sources. For example, suppose the warehouse keeps the average salary of employees at a company. When an update to an employee salary is reported, the update rule will have to obtain the current average from the warehouse in order to compute the new value. Scenarios where the integrator must obtain additional data from the sources are discussed in Section 4.3.

As discussed in Section 3, our ultimate goal is to provide a non-procedural, high-level specification language for describing how the relevant source data is integrated into the warehouse. Specifications in this language are compiled into the appropriate event processing code for the rule-based integration engine. This approach makes the integration process highly flexible and configurable, and allows the system to adapt easily to metadata changes at the sources or the warehouse.

For the initial implementation of our prototype, we are using the (CORBA compliant) Xerox PARC ILU distributed object system [8]. Using ILU allows the information sources, the monitors, the integrator, and the warehouse to run on different (distributed) machines and platforms while hiding low-level communication protocols, and it allows different components to be programmed using different languages.

4.3 Warehouse Update Anomalies

As mentioned in Section 2, in developing algorithms for integration one can think of a data warehouse as defining and storing a materialized view (or views) over the information sources. Numerous methods have been developed for maintaining materialized views in conventional database systems. Unfortunately, these methods cannot be applied directly in our warehousing environment. The key difference is that existing approaches assume that the system in which changes occur is the same system in which the view resides (or at least is a tightly-coupled related system). Consequently, when a change occurs, the system knows and has available any additional data needed for modifying the view.

In our warehousing environment, sources may be legacy or unsophisticated systems that do not understand views, and they are decoupled from the system where the view is stored. Sources may inform the integrator (through the monitor) when a change occurs, but they may not be able to determine or obtain additional data needed for incorporating the change into the warehouse. Hence, when a change notification arrives at the integrator, the integrator may discover that additional source data (from the same or different sources) is necessary to modify the view. When the integrator issues queries back to the sources, the queries are evaluated later than the corresponding changes, so the source states may have changed. This decoupling between the base data on the one hand (at the sources), and the view definition and view maintenance machinery on the other (at the integrator), can lead the to incorrect views at the warehouse. We refer to this problem as the *warehouse update anomaly* problem [21].

There are a number of mechanisms for avoiding warehousing update anomalies. As argued above, we are interested only in mechanisms where the source, which may be a legacy or unsophisticated system, does not perform any "view management." The source will only notify the integrator of relevant changes, and answer queries asked by the integrator. We also are not interested in, for example, solutions where the source must lock data while the warehouse view is modified, or in solutions where the source must maintain timestamps for its data. In the following potential solutions, view maintenance is autonomous from source changes.

Recompute the view. The integrator can either recompute the view whenever a change occurs at a source, or it can recompute the view periodically. Recomputing views is usually time and resource consuming, particularly in a distributed environment where a large amount of data might need to be transferred from the source to the warehouse.

Store at the warehouse copies of all data involved in views. By keeping up-to-date copies of all relevant source data at the warehouse, queries can be evaluated locally at the warehouse and no anomalies arise. The drawbacks are wasted storage and overhead for keeping the copies current.

The Eager Compensating Algorithm. The solution we advocate avoids the overhead of recomputing the view or of storing copies of source data. The basic idea is to add to queries sent by the integrator to the sources *compensating queries* to offset the effect of concurrent updates. For details and examples of the Eager Compensating Algorithm, variations on the algorithm, and discussion of performance issues, refer to [21].

We plan to implement and experiment with the Eager Compensating Algorithm, along with the alternative approaches, within the rule-driven integrator framework described in Section 4.2.

5 Summary and Plans

Data warehousing is a valuable alternative to traditional "passive" approaches for integrating and accessing data from autonomous, heterogeneous information sources. The warehousing approach is particularly useful when high query performance is desired, or when information sources are expensive or transitory.

The initial WHIPS project testbed installed in our laboratory uses for its data warehouse a Sybase relational DBMS. We are currently using data from our University accounting system, which is supplied to us as large legacy snapshots. As described earlier, we are experimenting with simple monitors that compute changes between consecutive snapshots. We also are in the process of implementing the integrator component, using ILU objects as the communication infrastructure between the monitors and the integrator, and between the integrator and the warehouse. Since we have not yet developed our high-level integration description language, event processing code is currently "hard wired" into the integrator.

In addition to the work in progress described in Section 4, plans for the near (and not so near) future include:

- Migrate to a larger and more heterogeneous testbed application, most likely financial data from a variety of sources including subscription services, monthly transaction histories, World-Wide-Web sites, news postings, etc. The heterogeneity of the sources in this application will provide impetus for experimenting with a wide variety of monitor types and will provide additional challenges for the integrator.
- Extend our work on snapshot differential algorithms to handle dump files with nested-object structures in addition to flat record structures.

- Develop and implement algorithms that optimize the change propagation and integration process in our warehousing architecture. In particular, we would like to perform filtering of changes at the monitor level that is as sophisticated as possible, so that we can avoid overloading the integrator with change notifications that are discovered to be irrelevant to the warehouse. Conversely, we may find it useful to store certain additional data at the warehouse, so that we can eliminate the need to query the sources when a change notification occurs.

- Develop an appropriate warehouse specification language and techniques for compiling specifications into integration rules and appropriate monitoring procedures (recall Sections 3 and 4.2).

References

[1] S. Abiteboul and A. Bonner. Objects and views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 238–247, Denver, Colorado, May 1991.

[2] T. Addyman. WAIS: Strengths, weaknesses, and opportunities. In *Proceedings of Information Networking '93*, London, UK, May 1993.

[3] E. Bertino. A view mechanism for object-oriented databases. In *Advances in Database Technology—EDBT '92, Lecture Notes in Computer Science 580*, pages 136–151. Springer-Verlag, Berlin, March 1992.

[4] N. Cercone, M. Morgenstern, A. Sheth, and W. Litwin. Resolving semantic heterogeneity. In *Proceedings of the Sixth International Conference on Data Engineering*, Los Angeles, California, February 1990.

[5] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991.

[6] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queues. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, Dublin, Ireland, August 1993.

[7] S. Chakravarthy and D. Lomet, editors. *Special Issue on Active Databases,* IEEE Data Engineering Bullet in 15(4), December 1992.

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

[9] U. Dayal. Active database management systems. In *Proceedings of the Third International Conference on Data and Knowledge Bases*, pages 150–169, Jerusalem, Israel, June 1988.

[10] U. Dayal and H.-Y. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Transactions on Software Engineering*, 10(6):628–645, November 1984.

[11] J. Hammer, D. McLeod, and A. Si. Object discovery and unification in federated database systems. In *Proceedings of the Workshop on Interoperability of Database Systems and Database Applications*, pages 3–18, Swiss Information Society, Fribourg, Switzerland, October 1993.

[12] E.N. Hanson and J. Widom. An overview of production rules in database systems. *The Knowledge Engineering Review*, 8(2):121–143, June 1993.

[13] W.H. Inmon. Building the data bridge: the ten critical success factors of building a data warehouse. *Database Programming & Design*, 1992.

[14] W.H. Inmon. EIS and the data warehouse: a simple approach to building an effective foundation for EIS. *Database Programming & Design*, 5(11):70–73, November 1992.

[15] W.H. Inmon and C. Kelley. *Rdb/VMS: Developing the Data Warehouse*. QED Publishing Group, Boston, Massachusetts, 1993.

[16] W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.

[17] Y. Papakonstantinou, H. Garcia-Molina, and J. Widom. Object exchange across heterogeneous information sources. In *Proceedings of the Eleventh International Conference on Data Engineering*, Taipei, Taiwan, March 1995.

[18] A. Sheth and J. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.

[19] G. Wiederhold. Mediators in the architecture of future information systems. *IEEE Computer*, 25(3):38–49, March 1992.

[20] G. Wiederhold. Intelligent integration of information. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 434–437, Washington, DC, May 1993.

[21] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 1995.