Multi-Agent Framework for Immediate Incremental View Maintenance in Data Warehousing

by

Gary Yeung

B.A.Sc., University of British Columbia, 1999

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF APPLIED SCIENCE
in the
School of Engineering Science

SIMON FRASER UNIVERSITY
September 2001

APPROVAL

Name: Gary Yeung

Degree: Master of Applied Science

Title of Thesis: Multi-Agent Framework for Immediate Incremental View Maintenance in Data Warehousing

Examining Committee:

Dr. John Dill
Professor, Engineering Science, SFU
Chair

_____

Dr. William A. Gruver
Professor, Engineering Science, SFU
Senior Supervisor

_____

Dr. Ke Wang
Associate Professor, Computing Science, SFU
Supervisor

_____

Mr. Dilip Kotak
Group Leader, Innovation Centre, NRC
Supervisor

_____

Dr. John Jones
Director, Engineering Science, SFU
External Examiner

_____

Date Approved: _____

Abstract

A data warehouse organizes and stores consolidated data derived from distributed, autonomous data sources for OLAP (Online Analytical Processing) and data mining analyses. The data warehouse views must be kept up-to-date as changes are made at the sources. View maintenance involves the process of propagating the changes to the data warehouse. Commercial systems typically designate view maintenance down time when the data warehouse is disabled for data updates. The maintenance time window could range from several hours to several days, depending on the size of the data warehouse, the data sources, and the volume of updates. The duration may be too long for some systems.

In this research, we applied a multi-agent approach to enable continuous updating of data warehouse views as transactions are executed at the sources. This technique, called immediate incremental view maintenance (IIVM), eliminates view maintenance down time for the data warehouse - a crucial requirement for Internet-based applications. Through the use of cooperation between agents, the data consistency problem usually associated with IIVM is solved. In addition, a fuzzy agent scheduling system was developed to prioritize tasks for the agents. The results from this research showed that the proposed multi-agent system drastically increases the availability of the data warehouse while preserving a stringent requirement of data consistency. The process parallelism inherent in the agent system improves overall processing time as compared to other existing IIVM systems.

Acknowledgments

I would like to give thanks to Dr. Bill Gruver who has been supervising my study and research for the past two years. Without his patience and flexibility, this thesis would not have been possible. I would also like to thank Dr. Ke Wang who helped me find the general direction of the research. I wish to thank Dr. John Jones and Mr. Dilip Kotak for reviewing this thesis. Finally, I want to give tribute to my family for their encouragement and support.

Dedication

To Mom, Dad, Christina, and Bonny

Contents

List of Tables

List of Figures

1 Introduction

1.1 Introduction

This chapter presents an overview of this thesis. View maintenance is the process of updating data warehouse data as transactions are performed at the underlying data sources. This research is inspired by the following three motivations. The first motivation is that as data warehouses are developed for global Internet access, there is a need to eliminate the down time for view maintenance. This thesis investigated and implemented a system for performing continuous view maintenance based on a technique called immediate incremental view maintenance (IIVM). The second motivation is that there is a growing need for higher scalability and maintainability for data warehousing systems. This research applied multi-agent technology to significantly improve the scalability using the plug-and-play concept. The third motivation is the need to improve view update speed and reduce data storage. The proposed multi-agent system performs the sub-processes for view maintenance in parallel to improve speed and to eliminate the need for storing copies of source data tables as required by the conventional deferred or batched view maintenance methods.

1.2 Objective

This thesis project concerns the design, development, and implementation of a multi-agent software system for performing immediate incremental view maintenance for a data warehousing system.

Data warehousing technology has been growing rapidly as many companies adopt OLAP and data mining tools to perform information discovery from the vast amount of

corporation data. As data warehouses are being developed for globally distributed data sources and OLAP end users, the down time for view maintenance must be considerably reduced or eliminated. In addition to the relatively large view maintenance time window, the conventional view maintenance method requires storing copies of data source tables.

The objective of this thesis is to develop a multi-agent system that enables continuous view maintenance during the operation of the data warehouse, thereby eliminating down time. Since it is crucial that the data warehouse maintains data consistency, a thorough investigation of the data consistency requirement for a data warehousing system is provided.

There are two existing systems developed for similar objectives. The earliest work on this subject is the development of the Strobe algorithms by Zhuge, et al. [35]. The SWEEP algorithm developed by Agrawal, et al., [7] made fundamental improvements to the Strobe algorithms by reducing the number of queries. These two systems have been investigated and they were used in the performance comparison testing in this thesis project.

A multi-agent system prototype was developed using the Java programming language. Data sources used were databases managed by Microsoft SQL Server 2000. Multi-threaded programming was utilized to simulate a distributed multi-agent environment.

1.3 Research Overview

This thesis project involves the design, implementation, and testing of an immediate incremental view maintenance system for a data warehousing system. The research project is divided into the following two stages. The first stage is the design of the multi-agent view maintenance system and its implementation. The second stage is a series of experiments that include a performance comparison of our system with the commonly

applied deferred view maintenance method and algorithms developed by other researchers, the Strobe algorithms and the SWEEP algorithm.

The main task of this thesis project is to convert and adapt a sequential view maintenance process into sub-processes and assign them to different agents. Such concurrent sub-processes could be managed by the agents to achieve gain in performance and scalability. Since transactions continue to execute at the data sources while the data warehouse is performing view maintenance, the data warehouse needs to manage the updates intelligently and treat interfering updates by compensation. Under all circumstances, the data consistency requirement must be satisfied in the data warehouse. Therefore, the task for this part of the project is to design the breakdown of tasks and the communication mechanism between agents such that the data consistency requirement is satisfied.

The experimental results of this thesis project have two objectives. The first objective is to demonstrate that the immediate incremental view maintenance method provides higher warehouse availability than the conventional deferred view maintenance method currently used in most commercial data warehouse products. It is important to justify the need for adopting immediate incremental view maintenance before trying to develop system based on this method. The second objective is to demonstrate that the proposed multi-agent system has better performance than the existing algorithms designed for the same purpose. These previous algorithms were analyzed and emulated in the same software environment. Tests were performed to compare these systems in various operational scenarios such as various data warehouse sizes and frequencies of update.

1.4 Organization of this Thesis

Chapter 2 presents an overview of database systems, data warehousing, and view maintenance. In particular, the two common view maintenance methods, deferred incremental view maintenance (DIVM) and immediate incremental view maintenance (IIVM), are presented. Chapter 3 is the core of this thesis. It explains each of the

modules that form the multi-agent system. The functionalities of each agent are explained and the mechanisms of agent interaction, such as blackboard coordination, cooperation domain, and failure recovery, are discussed. It investigates the IIVM process in more detail and provides a detailed comparison of our approach with the existing approaches. Previous research and its impact on our project are described throughout these two chapters.

In Chapter 4, the notion of data consistency is discussed. We present a formal proof to show that the multi-agent view maintenance system satisfies the specified data consistency requirement. Chapter 5 gives an overview of the software technologies used in the project, including SQL, JDBC, ODBC, and Java multi-threaded programming. Chapter 6 presents the experimental results from a series of tests. Finally, Chapter 7 provides the conclusions for this thesis study and ideas for further research.

1.5 Summary

In this chapter, the thesis project objective, a research overview, and the organization of this thesis were discussed. The main contribution of this thesis research project is a multi-agent architecture for performing immediate incremental view maintenance for a data warehouse and its verification.

2 Background

2.1 Introduction

The purpose of this chapter is to provide an understanding of database systems, data warehousing, view maintenance and how they are related to the subject of this thesis project. Then, the available methods for view maintenance in a data warehouse are discussed. The relevant literature is quoted throughout the development.

2.2 Database Systems

The most popular types of database designs include relational database design, object-oriented database design, and object-relational design. Among the different designs, relational database design has established itself as the primary data model for commercial data-processing applications. In this section, we shall briefly discuss the relational database design and proceed to discuss the Select-Project-Join (SPJ) Schema. Relational database design and SPJ schema are used for the data warehouse and the data sources in this thesis project.

2.2.1 Relational Database Design

A relational database consists of a collection of tables, each of which is assigned a unique name. A row in a table represents a relation among a set of values. For example, an "account" relation in a bank would consist of columns such as customer ID, account number, and balance. Each record or row in a table is called a *tuple*.

A *primary key* of a relational table is a column that contains only unique values, used to uniquely identify a tuple. An example of a primary key is customer ID if the relational table contains one and only one record for each customer. A *foreign key* of a relational table is a column that is the primary key of another table.

2.2.2 Select-Project-Join Schema

Relational algebra consists of a set of mathematical operators used to specify a subset of data in relational tables. The fundamental operations are *select, project, union, set difference*, and *Cartesian product*. In addition, there are other operations such as *set intersection, natural join, and division*. SQL (Sequel) is a computer language that implements these operations and we shall only discuss the *select*, *project* and *join* operations. The *select operator* (σ) extracts data that satisfy a condition. For example, $\sigma_{x<10}$ gives the tuples whose column x is less than 10. The *project operator* (Π) gives only the specified column of the selected tuples. For example, Π_x only gives column x of the data set. The *join operator* ($\bowtie$) combines two tables using a common column in the two tables. For example, if relation $R_1(X, Y)$ contains columns X and Y and $R_2(Y, Z)$ contains columns Y and Z, $R = R_1 \bowtie R_2$ presents a new relation R(X, Y, Z) with column Y as the link.

A *view* is a specification for a subset of data in relational tables using a set of relational operators. The purpose of a view is to provide a specific perspective of a relational table, focusing only on the data concerned by the view's purpose. For example, a manager who is concerned about customers whose balance is less than $500 would create a view to select those tuples that satisfies "balance < 500".

A *materialized view* is an executed view, that is, the relational tables are queried and data are organized in a result table. Therefore, a materialized view is basically another relational table that contains the subset of data from one or more relational tables.

A common method for creating a view is to use the *select*, *project*, and *join* operators. Materialized views that were created using only these three operators are said to have the Select-Project-Join (SPJ) schema. Such materialized view can be specified by an SPJ expression. For example, executing $\Pi_{W,X}\sigma_{W<500}$ ($R_1 \bowtie R_2 \bowtie R_3$) would build a *join* of relations R_1, R_2, and R_3, then select those tuples whose W attribute is less than 500, and then display only the W and X attributes to the result table.

2.3 Data Warehousing

Database applications can be broadly classified into transaction processing and decision support. Transaction processing systems are concerned with data storage and data retrieval. Such systems formed the core development of database technology. On the other hand, the more recently developed decision support systems are for organizing the transaction data in such a way that they can be helpful for making business decisions. A decision support system consolidates and summarizes the historical and current data from the transaction processing systems to facilitate decision-making or reporting. Data warehousing is a popular form of decision support system. In essence, a data warehouse is a collection of data from multiple data sources and its data are organized to facilitate information retrieval, instead of data retrieval. The purpose of a data warehouse is to combine related data into one location (e.g., a view table), such that analyses can be performed readily on such data. For instance, the sales data from different branches of a retail enterprise may be combined and stored in a data warehouse data table.

In general, a data warehouse can be modeled as three process components: Acquisition, Storage, and Access. Figure 2-1 shows the structure of a typical data warehouse.

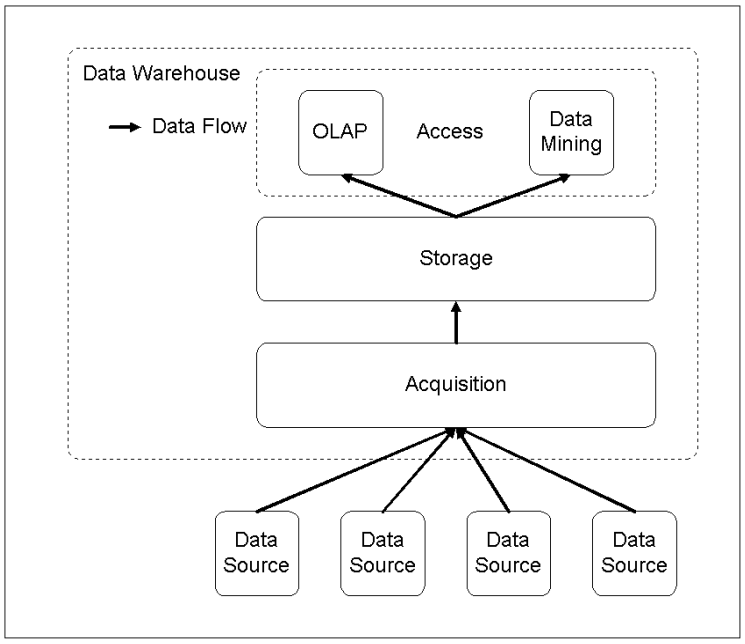

Figure 2-1 Data warehouse architecture

- **Acquisition** - The acquisition component is responsible for the extraction of data from various data sources. This component also has the responsibility of organizing the incoming data for storage. The source data are cleaned, validated, and formatted by this component. Schema integration might be needed if the warehouse data and the source data have different schemas, or even different data models. The acquisition component is the data warehouse's interface to the data sources. In organizing data for storage, this component must ensure the data being stored in the storage component are consistent.

- **Storage** - The storage component keeps the summarized data acquired from the data sources in a certain format. Common storage formats include star schema, data cube, and linked materialized views. A star (or snowflake) schema has a central fact table with summaries of the fact table (dimension tables) branching out from it. Data cubes store data as a 3D structure and each dimension of the cube represents an attribute. Data are distributed throughout the cube and this

cubic structure facilitates analysis technique such as slicing. We shall only be concerned with linked materialized views since this is the storage format used in this thesis project. Linked materialized views are also relational data tables. According to Mattison [25], this is the most popular format storage used by data warehouses in the industry. These materialized views contain consolidated data from various data sources. They are usually the aggregations of the source data (e.g., totals, averages, maximum values, minimum values) or groupings of data elements that are commonly used together (e.g., *join*). A common method to define the materialized view in the storage is the SPJ schema. The materialized views are defined by *project*, *select*, and *join* operations performed on source tables. Virtual views are sometimes used in a data warehouse in addition to materialized views. Virtual views are view definitions without the extracted data. Hull and Zhou [24] applied a hybrid system of virtual and materialized views in storage.

- **Access** - The access component of a data warehouse consists of a collection of tools for the end users to perform query, analysis, visualization, and manipulation. The access components include various OLAP and data mining tools. It is the data warehouse's interface to the end users. The simplest access tools would be the graphing tools and sophisticated data mining tools are available to perform detailed analysis using the data stored in the data warehouse. Simulation can be used to perform forecast and risk analysis. Other mathematical techniques, such as Genetic Algorithms and Neural Networks, can be adopted for modeling and characterization of systems using the data in the data warehouse.

In this study, we are concerned only with the acquisition and storage components. We aim to apply multi-agent technology to these two layers to make the update process of the storage more efficient.

2.4 View Maintenance

A data warehouse gathers and stores data from distributed autonomous data sources. When data updates (e.g., due to transactions) happen at the sources, the views at the data warehouse must be updated accordingly. This operation is called the view maintenance operation of a data warehouse. View maintenance is the updating of views when the source data from which the view derived are changed. This includes transactions such as update, insertion, and deletion in the source data tables. To update the warehouse views, the views can either be recomputed from the beginning or by using incremental maintenance techniques.

View maintenance can be implemented by re-computation or by incremental maintenance (Figure 2-2). The re-computation method regenerates all the views in the data warehouse at a predefined time interval. In the early years of data warehousing development, this view maintenance approach was the standard method because of its simplicity. On the other hand, the incremental view maintenance method updates the views by applying only the changes made at the data sources to update the views in the data warehouse. Many software developers now realized that incremental updates could significantly reduce the view maintenance time for large source databases and they are now providing this option in their data warehouse products (Table 2.1). According to Zhuge [35], incremental view maintenance typically out-performs re-computation in cases where the volume of source data is large, source data are not constantly available, or no down time for the data warehouse is permitted for re-computation.

Incremental view maintenance is typically based on "incremental queries" that avoid the need to re-compute a materialized view. For a view that spans multiple sources, an incremental query is executed at each source to compute the new view tuple(s) that updates the materialized view.

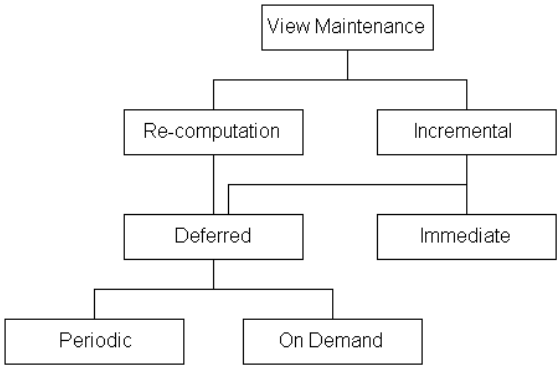

Figure 2-2 Classification of view maintenance methods

View maintenance can also be further categorized into two types of update scheduling:

- Immediate - In the immediate update approach, the view at the data warehouse is updated immediately when its base relation at the source is changed.

- Deferred - In the deferred update approach, the warehouse views are updated collectively at a designated time when changes are applied as a batch.

In general, the available view maintenance strategies are immediate incremental, deferred incremental, and deferred re-computation. It is not feasible to have immediate re-computation due to the unrealistic computing power required. Furthermore, deferred update can be done periodically or only when the client makes a query against the view (on-demand).

2.4.1 Deferred Incremental View Maintenance (DIVM)

Deferred Incremental View Maintenance (DIVM) is a common type of view maintenance. The data warehouse is locked at a designated time for performing view updates. In this method, changes to the source tables are recorded in a log file. During the view maintenance process, the changes are applied in a batch to the materialized

views in the data warehouse according to the sequence of transaction at the source tables. There are many alternative ways to record the changes, primarily to improve the efficiency of the process and to reduce the view maintenance time window.

One of the earliest studies was done by Hanson [8], where differential tables are maintained on base tables that contain the suspended updates that have not actually been applied to the database state. Colby [18] applied base logs and differential tables for periodic update of views, which is similar to taking snap-shots from every state of change in the base tables. Mumick [15] improved this method by storing changes of base tables in a so-called summary-delta table. The information in the summary-delta table is updated to the summary tables during off-hours. Gupta [1] proposed a counting algorithm to keep track of the order of updates for each tuple in a view, so that updates can be applied at the correct data warehouse states.

DIVM typically requires a nightly batch window for updating the materialized views stored at the warehouse. The drawback is that the data warehouse is typically unavailable to the users during view maintenance because of the large number of updates that need to be applied. Since the data warehouse must be made available to readers again by the next morning, the time required for view maintenance is often a limiting factor in the number of materialized views that can be made available in the warehouse. The number of materialized views available has a significant impact on OLAP query performance. Therefore, using DIVM limits the size of the data warehouse.

2.4.2 Immediate Incremental View Maintenance (IIVM)

An update made at the data source could be applied to the data warehouse view immediately after the transaction committed at the source table. This is called immediate incremental view maintenance and is achieved by sending update notifications to the data warehouse from the data sources. Typically, a monitor is installed in each data source to

detect transaction commits (insertion, deletion, or update). Once a data change is detected, a message is sent to the data warehouse delivering information such as the modified relation, the pre-update tuple, and the post-update tuple. The data warehouse receives and processes the update notification and makes modification to the warehouse views affected. Since this is a continuous process, IIVM removes the need to store changes for update and to designate off-hours for view maintenance.

Some systems have been developed for IIVM. ECA (Eager Compensation Algorithm) [35] was designed for a data warehouse system with a single data source. The Strobe algorithms (Strobe, C-Strobe, and G-Strobe) [37] perform IIVM for multiple, distributed data sources and avoid data inconsistency by using compensation queries. The SWEEP [7] algorithm improves the Strobe algorithms by eliminating compensation queries. The algorithm by Ling & Liu [28] adopts most parts of the SWEEP algorithm and made an improvement of incorporating a message counter at each data source to remove the message transmission disorder problem due to network delay.

We shall present an architecture for performing IIVM and compare its performance with the C-Strobe (Complete Strobe) and the SWEEP algorithms in the experimental studies (Chapter 6).

2.4.3 Motivation for Adopting IIVM

Most commercially available products adopt the DIVM approach while the IIVM approach still remains in the research stage (Table 2.1).

Product Name	Incremental (I) / Re-computation (R)	Deferred (D) / Immediate (IM)
Microsoft SQL Server 2000 [19]	Options of I and R	D
IBM Data Warehouse Manager [13, 14]	Options of I and R	D
IBM Data Propagator	Options of I and R	D
IBM Visual Warehouse	R	D
Oracle Warehouse Builder [20]	Options of I and R	D
Sybase Adaptive Server IQ [5, 21, 26]	Options of I and R	IM (batched)

Table 2.1 Commercially available data warehouse products

As the system scales up (e.g., data warehouse size, data source size, or update frequency), the time needed for performing view maintenance increases drastically. If DIVM is used, there is a limit that restricts the size of the system because of the required view maintenance time. For data warehouses that provide global access over the Internet, no down time for view maintenance is possible.

Many online or soft real-time applications, especially in banking and communications, are now trying to take advantage of the data warehouse for high-speed global access of current data gathered from distributed sources. Such application requires more current data than is possible by DIVM, which keeps the data warehouse as current as the previous view maintenance. Furthermore, if the source data are not constantly available, re-computation is unfeasible. Therefore, IIVM is valuable for data warehouses and it is the subject of this study.

2.5 Summary

The concepts of data warehousing and view maintenance were discussed in this chapter. Among the three main components of a data warehouse, namely, Acquisition, Storage, and Access, we shall focus on the Acquisition and the Storage components in this study. It was advocated that Immediate Incremental View Maintenance (IIVM) possesses some

important advantages over the Deferred Incremental View Maintenance (DIVM), such as view freshness and data warehouse availability. Therefore, IIVM is the technique investigated in this thesis and its use will be justified by tests presented in Chapter 6 - Experimental Studies.

3 Multi-Agent IIVM System Architecture

3.1 Introduction

In this chapter, we shall investigate the IIVM mechanism in detail and the potential data inconsistency that may occur. Section 3.2 investigates IIVM and the methods that previous researches applied as compared to the approach used in this thesis project. Section 3.3 discusses the general architecture and assumptions for our data warehousing system. In Section 3.4, the functionalities of each agent in the system shall be explained in detail. Section 3.5 analyzes the different processes involved in IIVM and the advantages offered by the multi-agent system over the existing systems. In Section 3.6, the features of the multi-agent IIVM system shall be highlighted and their significance to the performance improvement and consistency enforcement shall be explained. The fuzzy system used for agent coordination shall be discussed in Section 3.7. In Section 3.8, the failure recovery procedure shall be explained in detail. Section 3.9 provides a summary for this chapter.

3.2 Immediate Incremental View Computations

In this section, we shall discuss the processes involved in IIVM. Furthermore, the view consistency issues associated with IIVM computations will be discussed.

Using IIVM, the data warehouse is updated incrementally in response to the update notification messages arriving from the data sources. If any two updates are sufficiently far apart in terms of time such that the incremental re-computation of the view is not

interfered with these updates, the view maintenance can be carried out in a straightforward manner. For a typical warehouse view V defined by

$$V = \Pi_{ProjAttr}\sigma_{SelectCond}(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n) \quad (1)$$

where ProjAttr specifies the attributes for the *project* operation, and SelectCond specifies the conditions for the *select* operation.

In this case, when an update ΔR_i, i.e., an update to the base relation R_i, is received at the data warehouse, the incremental change to the view is computed by first querying the data sources to compute the *join*, then applying the select and project operations. The query to be evaluated for the *join* is expressed as

$$Q = R_1 \bowtie \ldots \bowtie \Delta R_i \bowtie \ldots \bowtie R_n \quad (2)$$

That is, the view must be re-assembled by making queries for each relation in the SPJ expression. This entails making queries to each of the sources that hold the relations R_1, R_2, … R_n.

Using relational algebra to illustrate, assume two relations R_1 and R_2 and a view defined as a natural *join* over the two relations, i.e., $V = R_1 \bowtie R_2$. As a result of an update ΔR_1 the new view should be as follows:

$$V_{new} = (R_1 + \Delta R_1) \bowtie R_2 = R_1 \bowtie R_2 + \Delta R_1 \bowtie R_2 \quad (3)$$

Since the data warehouse already has $R_1 \bowtie R_2$, it only needs to compute a query $Q_1 = \Delta R_1 \bowtie R_2$ at the data source containing R_2 and incorporate the results into the materialized view. In our model, an update is in the form of either an insertion and a deletion of a tuple. Using the above notation, ΔR for an insertion will carry a positive

sign and hence will have the effect of adding tuples to the materialized view. Deletion will result in ΔR having a negative sign and therefore will result in removing the appropriate tuples from the materialized view. In this way, the views can be maintained incrementally as long as R_2 did not change since ΔR_1 occurred. Figure 3-1 illustrates this operation.

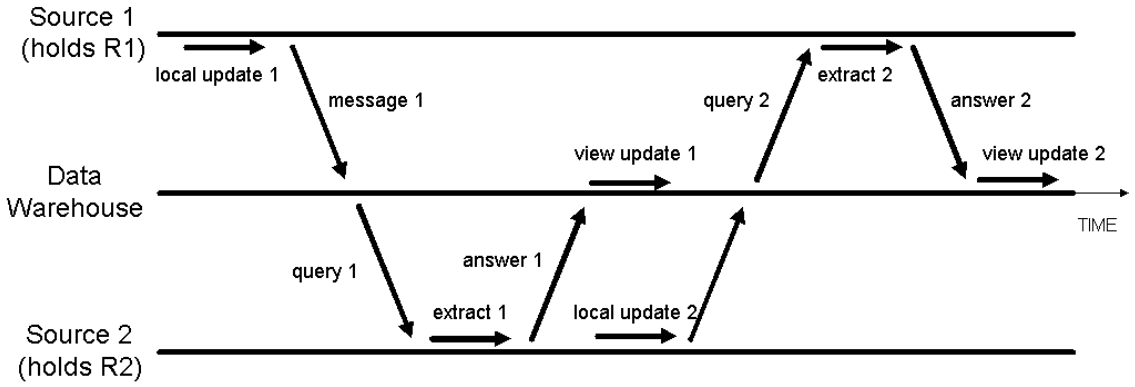

Figure 3-1 Immediate incremental view update with isolation

Example 3-1

Assume a view V_1 is defined as a *join* of relations $R_1(X,Y)$ and $R_2(Y, Z)$. That is, $V_1 = R_1 \bowtie R_2$. Table 3.1 illustrates the IIVM process. Initially, at time t_0, V_1 contains one tuple (1, 2, 3). At time t_1, tuple (2, 2) is inserted into R_1 and the update notification is sent to the data warehouse. To evaluate the incremental view, the data warehouse queried R_2 and found $R_2 = (2, 3)$ that joins with $R_1 = (2, 2)$. At t_2, V_1 is updated to contain the tuple (2, 2, 3). At t_3, tuple (1, 2) is deleted from R_1 and notification is sent. Finally, at t_4, the data warehouse queries R_2 again and deletes (1, 2, 3) from view V_1. The result is correct.

Time	Action	R_1 (X, Y)	R_2 (Y, Z)	$V_1 = R_1 \bowtie R_2$
t_0		(1, 2)	(2, 3)	(1, 2, 3)
t_1	**insert**(R_2, (2, 2))	(1, 2), (2, 2)	(2, 3)	(1, 2, 3)
t_2		(1, 2), (2, 2)	(2, 3)	(1, 2, 3), (2, 2, 3)
t_3	**delete**(R_1, (1, 2))	(2, 2)	(2, 3)	(1, 2, 3), (2, 2, 3)
t_4		(2, 2)	(2, 3)	(2, 2, 3)

Table 3.1 IIVM process with isolation property preserved

The updates are not always clearly separated or isolated since the data sources are assumed to be distributed and autonomous. Transactions at the sources take place concurrently with the view maintenance process. A transaction does not wait for the warehouse view update to finish before it proceeds.

We shall consider the case in which the isolation condition is violated. In the above example, if another update ΔR_2 occurred before R_2 was queried for the first update. The queried answer would have been wrong. This scenario is illustrated in Figure 3-2.

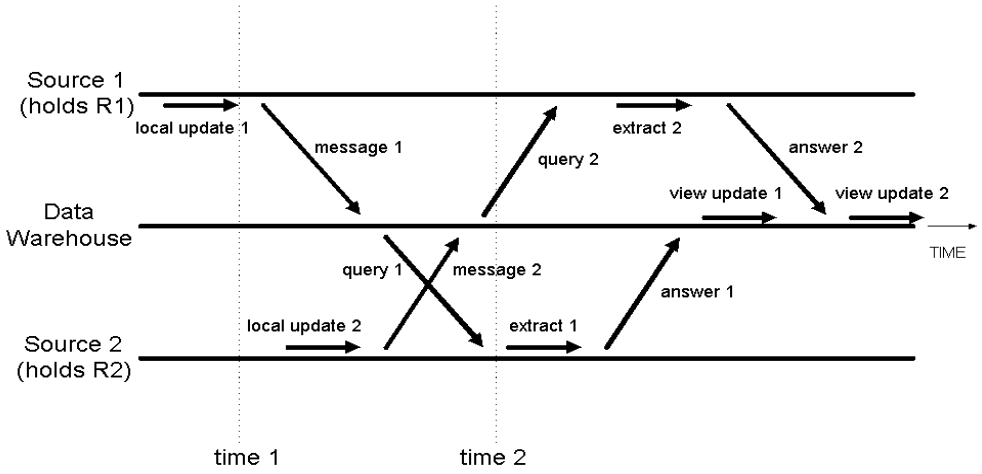

Figure 3-2 Concurrent view update

Using relational algebra, as a result of two updates, the view should change to the following:

$$V_{new} = (R_1 + \Delta R_1) \bowtie (R_2 + \Delta R_2) = \{ (R_1 \bowtie R_2) + (\Delta R_1 \bowtie R_2) + (R_1 \bowtie \Delta R_2) + (\Delta R_1 \bowtie \Delta R_2) \} \quad (4)$$

The answer of the first incremental query will include the effect ΔR_2 and hence will be $A_1 = \Delta R_1 \bowtie (R_2 + \Delta R_2) = (\Delta R_1 \bowtie R_2) + (\Delta R_1 \bowtie \Delta R_2)$. We refer to $\Delta R_1 \bowtie \Delta R_2$ as the error term in the incremental answer due to concurrent update ΔR_2. Only incorporating A_1 into the materialized view will not reflect all the changes that should have occurred after the two updates, i.e., $R_1 \bowtie \Delta R_2$ is missing. In essence, the missing state is caused by the concurrent update modifying the transaction source state. As illustrated in Figure 3-2, the local update 2 causes the state of source 2 to be different in time 1 than in time 2.

Example 3-2

In this example, we shall illustrate a case where a wrong final view is caused by a concurrent update. Using Example 3-1 with $V_1 = R_1 \bowtie R_2$ again, Table 3.2 illustrates the IIVM process with an insertion interfering with an insertion. Initially at time t_0, V_1 contains tuple (1, 2, 3). At time t_1, an update *insert* (R_1, (2, 2)) occurs. At time t_2, another insertion *insert* (R_2, (2, 4)) occurs and the query answer of the first insertion incorporates this effect, giving tuples (2, 2, 3) and (2, 2, 4). At time t_3, the query answer for the second insertion returns tuples (1, 2, 4) and (2, 2, 4) again. The final view contains an extra (2, 2, 4) and that is incorrect.

Time	Action	R_1 (X, Y)	R_2 (Y, Z)	$V_1 = R_1 \bowtie R_2$
t_0		(1, 2)	(2, 3)	(1, 2, 3)
t_1	**insert** (R_1, (2, 2))	(1, 2), (2, 2)	(2, 3)	(1, 2, 3)
t_2	**insert** (R_2, (2, 4))	(1, 2), (2, 2)	(2, 3), (2, 4)	(1, 2, 3), (2, 2, 3), (2, 2, 4)
t_3		(1, 2), (2, 2)	(2, 3), (2, 4)	(1, 2, 3), (2, 2, 3), (2, 2, 4), (1, 2, 4), (2, 2, 4)

Table 3.2 IIVM process resulting in duplicates

3.2.1 The Strobe Algorithms

A mechanism must be in place to alleviate the problems associated with concurrent updates. The ECA [35] protocol is based on the idea of using compensation to formulate the query to offset the error term introduced in the interfered update. For Example 3-2 above, the compensated query Q_1 would have been formulated as:

$$Q_1 = (R_1 - \Delta R_1) \bowtie \Delta R_2 = (R_1 \bowtie \Delta R_2) - (\Delta R_1 \bowtie \Delta R_2) \quad (5)$$

The answer of this query removes the first (2, 2, 4) that was incorrectly inserted into the view V_1. However, the ECA has a rather significant limitation. When the system consists of distributed data sources, the compensation part can no longer be patched onto the subsequent queries since SQL does not allow convenient references to relations in distributed databases. The Strobe algorithm [37] proposed performing compensation through compensation queries. Consider a view definition involving multiple relations ($V_1 = R_1 \bowtie R_2 \bowtie R_3$) and an update ΔR_1. In response, the data warehouse by default dispatches a query $Q_1 = \Delta R_1 \bowtie R_2 \bowtie R_3$ to the data sources corresponding to the base relations R_2, R_3. If no concurrent updates arise, then the resulting answer of Q_1 can be merged into the materialized view, which will reflect the correct state after ΔR_1. However, if a concurrent update ΔR_2 occurs during the computation of the answer to Q_1, then the answer will have an error term $\Delta R_1 \bowtie \Delta R_2 \bowtie R_3$. The data warehouse now has to send a query, which is $Q_{11} = \Delta R_1 \bowtie \Delta R_2 \bowtie R_3$, to R_3 to subtract out the effects of the error term. During this evaluation, if a concurrent update ΔR_3 occurs, then a second further compensation query needs to be sent to the remaining base relations. A potential problem is a nested compensation loop that would take infinite amount of time to finish if concurrent update never terminates.

In the Strobe algorithm, the sub-queries for a view update are completely finished before compensation is processed. If a given view requires k sub-queries for a view update, the compensation for a concurrent update will take k-1 queries. If each source has a concurrent update, the whole compensation process potentially requires (k-1)! queries.

3.2.2 The SWEEP Algorithm

The SWEEP algorithm [7] eliminates the need for compensation queries in distributed sources by a local compensation process. After the sub-query at each source, the data warehouse checks for concurrent update at that source in the data warehouse's update queue. If a concurrent update occurred at the source between the time of transaction and the time of query (e.g., between time 1 and time 2 in Figure 3-2), compensation will be applied immediately to the sub-query answer to offset its effect using the update notification in the update queue. Local compensation is possible since the checking for concurrent update or interference is done at the sub-query of each data source. The concurrent update notification is used for compensation and it will still remain in the queue to be processed according to the order of arrival. This process is illustrated in Figure 3-3:

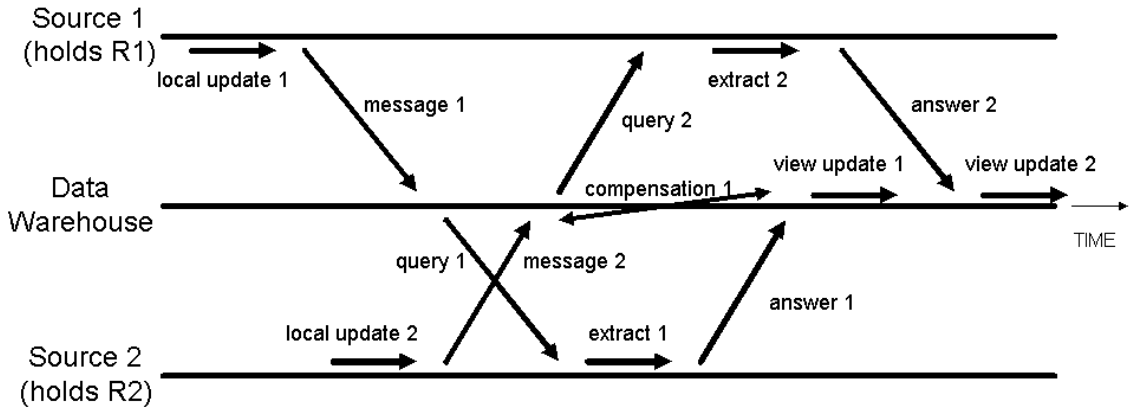

Figure 3-3 Concurrent view update with local compensation

Example 3-3

We shall use Example 3-2 and implement local compensation. Assume $V_1 = R_1 \bowtie R_2$. Table 3.3 illustrates the IIVM process with local compensation. Initially at time t_0, V_1 contains tuple (1, 2, 3). At time t_1, a tuple (2, 2) is inserted to R_1 and an update notification is sent to the data warehouse. At time t_2, a concurrent update **insert** (R_2, (2,

4)) occurs at the source. As a result, the query answer of the first insertion includes the tuple (2, 2, 4) due to the concurrent update. The system detects the interfering update at R_2 and compensates for it by deleting $\Delta R_1 \bowtie \Delta R_2$ (i.e., (2, 2, 4)) from the result. So, only the tuple (2, 2, 3) is inserted. Finally at time t_3, the query answer for the second insertion returns and correctly updates the warehouse view V_1. Note that not only the final result is correct, all the states for the sources are correctly reflected in V_1.

Time	Action	R_1 (X, Y)	R_2 (Y, Z)	$V_1 = R_1 \bowtie R_2$
t_0		(1, 2)	(2, 3)	(1, 2, 3)
t_1	**insert** (R_1, (2, 2))	(1, 2), (2, 2)	(2, 3)	(1, 2, 3)
t_2	**insert** (R_2, (2, 4))	(1, 2), (2, 2)	(2, 3), (2, 4)	(1, 2, 3), (2, 2, 3)
t_3		(1, 2), (2, 2)	(2, 3), (2, 4)	(1, 2, 3), (2, 2, 3), (1, 2, 4), (2, 2, 4)

Table 3.3 IIVM process with local compensation

In Appendix A, all possible anomalies associated with concurrent updates in IIVM are investigated in detail and more examples are given.

3.2.3 Multi-Agent System

In our multi-agent approach, we create certain types of agents to accomplish the tasks involved in IIVM described above. The multi-agent system resembles an assembly line in that tasks in a strict sequence are passed through the system while suitable and available agents work on the tasks concurrently. The finished product would be the complete, properly compensated view tuple(s) ready for updating the warehouse views. The system has specific agents for performing sub-queries. Some agents are for managing views and some are for managing sources. Compensation is done through cooperation between agents. This architecture improves the processing speed of the system since maximum concurrency is used. For instance, the sub-tasks for a later update could start before the current update is completed. It provides better scalability as the existence of a source or a view is completely determined by the existence of the specific

agent. The Multi-Agent System, SWEEP, and Strobe algorithms will be directly compared in performance in Chapter 6 - Experimental Studies.

3.3 System Architecture

Even though all previously developed systems have slightly different implementation methods for IIVM, a number of processes are essential for performing IIVM. In this section, we shall discuss the data warehouse model and the assumptions made regarding the data sources and the network.

3.3.1 Assumptions of Data Source

We adopt the data warehouse model developed in references [24, 35, 37]. In this model, updates occurring at the data sources are classified into three categories:

1. Single update transactions where each update is executed at a single data source.
2. Source local transactions where a sequence of updates is performed as a single transaction. However, all of the updates are directed to a single data source.
3. Global transactions where the updates involve multiple data sources.

In this study, we assume that the updates being handled at the data warehouse are of types 1 and 2.

The underlying database model for each data source is assumed to be a relational data model. Each data source may store any number of base relations, but conceptually we assume a single base relation R_i at data source i.

The architecture of the data warehouse was shown in Figure 2-1. The data warehousing system consists of n sites for data sources and another site for storing and maintaining the

materialized views of the data warehouse. The communication between each data source and the data warehouse site is assumed to be reliable and FIFO, i.e., messages are not lost and are delivered in the order in which they were sent. No assumption is made about the communication between the different sites maintaining the data sources. In fact, they may be completely independent and may not be able to communicate with each other. The sources are assumed to be "non-cooperative sources" since no assumption is made about the sources' knowledge of the existence of a data warehouse. The sources only perform queries and log updates and these are very standard functions of any SQL database.

There are two types of messages from the sources to the data warehouse: reporting an update and returning the answer to a query. There is only one type of message in the other direction: the data warehouse may send queries to the sources. We assume that each single update transaction and source-local transaction is reported in one message, at the time when the transaction commits. For example, a relational database source may trigger a command to produce a message on any transaction commit.

Each data source is assumed to be completely autonomous in that the updates on different data sources are not related, and therefore not synchronized. However, we assume that for a given data source, updates are executed atomically. By atomic update, we mean that each source action, plus the resulting message sent to the warehouse, is considered one event. For example, evaluating a query at a source and sending the answer back to the warehouse is considered one event. Processing a transaction and sending the update message to the warehouse is considered another event. It is not allowed, for example, for the source to process a transaction during the time a query is in process (the time after a query has been started and before the query answer has been sent).

In this model, as updates occur, they are conceptually seen as being transmitted asynchronously from the sources to the data warehouse. All the updates performed atomically at a data source are sent as a single unit from the source to the data warehouse.

3.3.2 Assumptions of Data Warehouse

We assume that the view used at the data warehouse to form the materialized view is defined by the SPJ expression. That is, if $\{R_1,\ldots, R_i,\ldots, R_n\}$ are the n base relations at the corresponding data sources, the view denoted by V is as follows:

$$V = \Pi_{ProjAttr}\sigma_{SelectCond}(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n) \quad (6)$$

It is possible to model the data warehouse using more complex view functions such as aggregations (e.g., average, maximum, minimum), but for simplicity we restrict the model in this study to SPJ expressions. The updates to the base relations are assumed to be insertions and deletions of tuples. Furthermore, the operation "update" is modeled as a deletion followed by an insertion. Although some earlier data warehouse models (e.g., [37]) assume that the view definition includes the key attributes of each base relations, no such assumption is made here.

The updates at the data sources can be handled at the data warehouse in different ways. Depending on how the updates are incorporated into the view at the data warehouse, the different notions of view consistency will be identified in Chapter 4.

3.4 Multi-Agent IIVM System

Multi-agent systems (MAS) can be viewed as a software technology that is able to model and implement individual and social behavior in distributed systems. Agents possess several common characteristics, such as their ability to communicate, cooperate, and

coordinate with other agents in a multiple agent environment. Agents generally possess distinctive skills to perform specific tasks. Each agent is capable of acting autonomously, cooperatively, and collectively to achieve the collective goal of a system. The coordination capability helps manage problem solving so that cooperating agents work together as a coherent team. The coordination is achieved, for example, by exchanging data, providing partial solution plans and constraints among agents.

The reasons for applying a multi-agent framework in IIVM are to provide: (i) Scalability (ii) Efficiency through Parallel Processing. A data warehouse is a dynamic system. Sources or views that are obsolete should be removed to optimize performance. A multi-agent system is the most scalable system since agents are independent and they can be added or removed quickly. Parallel processing is easily feasible in such a system since agents are autonomous and can run processes in parallel.

Our multi-agent system was designed with the above two objectives in mind. The view maintenance system can be seen as the interface between data sources and warehouse views. Both the sources and views are added or removed depending on the changing needs. In the multi-agent system, there is an agent (Update Agent) responsible for each source and there is another agent (View Agent) responsible for each view. With the meta-data (information about views) completely distributed, the system can be expanded or shrunk by adding or removing agents. The third agent, Extract Agent, is responsible for querying the source upon request. Furthermore, the Blackboard is a common area for the agents to post and receive tasks. We shall discuss the functionality of each type of agent in the following sections.

The multi-agent framework exploits the opportunities for executing the IIVM sub-processes in parallel. For instance, querying tasks for different sources could be carried out in parallel. Other tasks could be carried out in parallel as long as the receiving order of the updates equals the updating order of the warehouse views.

3.4.1 Update Agent

The update agent stores the information regarding the source (source meta-data), such as the complete locator (URL) for the data table (network address, database name, connection protocol) and the relation identities. The functions of the update agent are: (i) polling a data source for updates (ii) converting the update message into an update task (iii) posting the update task onto the blackboard. The algorithm for update agent can be described by the following pseudo-code:

```
1. PROCESS UpdateAgent
2.      LOOP
3.              RECEIVE ΔR_i from Data Source i;
4.              IF (ΔR_i NOT 0)
5.                      UpdateTask = CONVERT (ΔR_i);
6.                      POST (UpdateTask) TO Blackboard;
```

where
ΔR_i denotes update of relation R_i from source i;
UpdateTask denotes the view update request;
CONVERT denotes the function for converting an update message to an UpdateTask;
POST denotes the function for posting UpdateTask to Blackboard.

3.4.2 View Agent

A view agent holds the meta-data regarding a view, such as the SPJ view definition and the origins of the view relations. The view agent holds the mapping of source relations to view relations, as a view may organize relations in a way different from the source tables. The functions of the view agent are: (i) checking if an update is associated with the view it represents (ii) monitoring for concurrent update (iii) modifying warehouse view. If a view agent detects that the incoming updates interferes with another update in process, it passes the incoming update to the cooperating extract agent to perform compensation. The operation of the view agent can be described by the following pseudo-code:

```
1. PROCESS ViewAgent
2.      LOOP
3.              RECEIVE ΔRi FROM Blackboard
4.              IF (∃ΔRi ∈ V) THEN
5.                      POST ΔV TO Blackboard;
6.              IF (ΔRi INTERFERES ΔV) THEN
7.                      SEND ΔRi TO ExtractAgent;

8.              RECEIVE ΔV FROM Blackboard
9.                      V = V + ΔV;
```

where
ΔR_i denotes update of relation R_i from source i;
ΔV denotes new incremental view, $\Delta V = R_1 \bowtie ... \bowtie \Delta R_i \bowtie \ldots R_n$;
INTERFERES denotes a test of interference for a concurrent update.

3.4.3 Extract Agent

The extract agents take requests to complete the sub-queries required to reconstruct a view. The number of extract agents is limited, depending on the computing resources of the system. The following pseudo-code describes the functionality of the extract agent:

```
1. PROCESS ExtractAgent
2.      LOOP
3.              RECEIVE ΔV FROM Blackboard
4.              FOR (i = Position + 1; i ≤ NumberOfRelations; i++) DO
5.                      SEND ΔV TO Source i;
6.                      RECEIVE ΔV FROM Source i;
7.                      IF (∃ΔR FROM ViewAgent) THEN
8.                              ΔV = ΔV - ΔR ⋈ TempView
9.              ENDFOR
10.             POST  ΔV TO BlackBoard
```

where

ΔV denotes new incremental view, $\Delta V = R_1 \bowtie \ldots \bowtie \Delta R_i \bowtie \ldots R_n$;
Position denotes the position of the updated relation in the SPJ expression;
NumberOfRelations denotes the number of relations in the view definition;

3.4.4 Blackboard

The blackboard is a buffer that is accessible by all agents in the system. It acts as a common knowledge area where requests are posted and received by the agents in the system. The blackboard does not perform any decision making function and it has no control over how the system should run. The idea of a multi-agent system is the non-existence of a central controller. The blackboard keeps the update task requests in a queue based on the arriving order of update notifications. The blackboard would multi-cast view modification requests to the view agents only for the first update task in the queue in order to preserve the order of transaction at the sources. The blackboard is described by the following pseudo-code:

```
1. PROCESS Blackboard

2. IF (∃UpdateTask ∈ TaskBuffer[]) THEN
3.       NOTIFY (∀ViewAgent);

4. IF (∃ExtractTask ∈ TaskBuffer[]) THEN
5.       NOTIFY (∀ExtractAgent);

6. IF (∃WriteTask ∈ TaskBuffer[1]) THEN
7.       NOTIFY (∀ViewAgent);
```

where
TaskBuffer[] denotes an array containing the pending task requests;
UpdateTask denotes an update notification from the source;
ExtractTask denotes a request to perform sub-queries;
WriteTask denotes a request to perform view modification.

3.4.5 Multicast Messaging

Multicasting is similar to broadcasting except that the receivers are restricted to a group of individuals, instead of all individuals. It is equivalent to the subscribe/unsubscribe system. Multicasting is very suitable in this system since there are three distinct types of agents. Notifications are usually made to the relevant group only.

3.4.6 A Multi-Agent IIVM System Example

The following sequence diagram illustrates the processes in a scenario where two updates messages are delivered to the data warehouse. Note that the agents could perform other tasks during the time of inactivity. Only two simple tasks are used here to illustrate the operation. The system can provide more parallelism in actual operation with more frequent updates.

Example 3-4

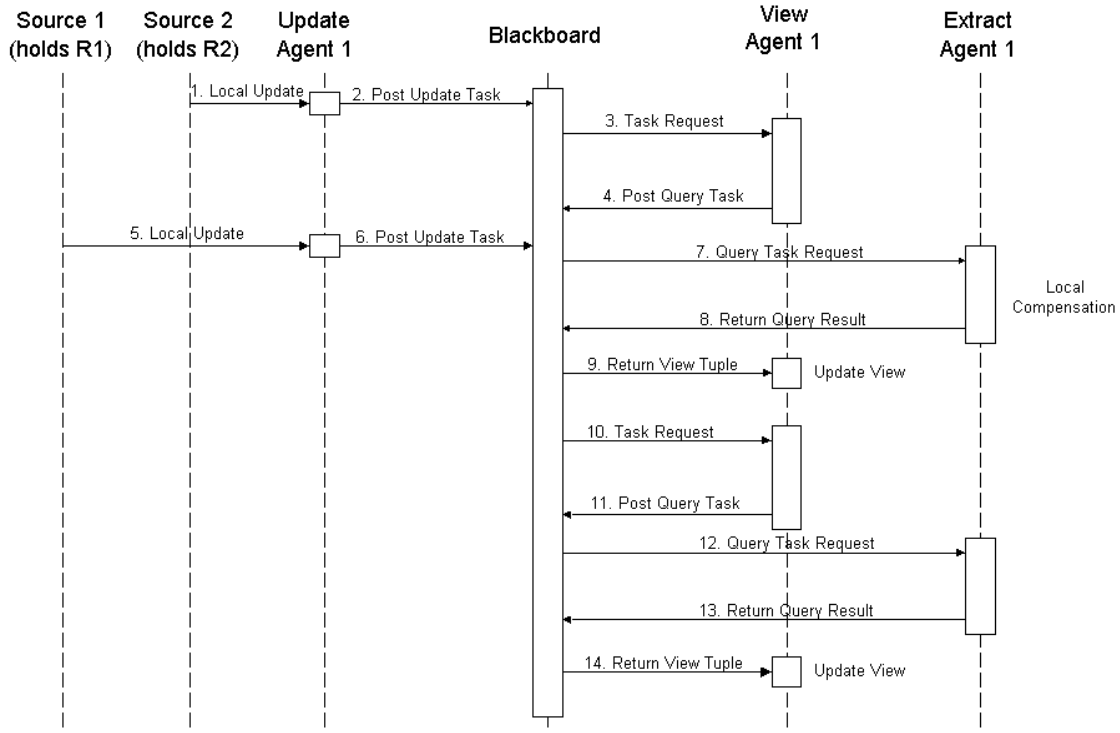

Figure 3-4 Sequence diagram for multi-agent IIVM system example

3.5 Comparison of IIVM Systems

IIVM involves re-assembly of a view as defined by the SPJ view definition. In this section, we shall divide the general IIVM process in the data warehouse into general steps and discuss each of them in each sub-section. These general steps are common for

all previous researches studied. We shall investigate these general steps and provide comparison with our multi-agent IIVM system.

3.5.1 Associating Updates with Views

Sources dispatch update messages (notifications) to the data warehouse when an update transaction occurs. The message would as a minimum include the identity of the changed relation, the pre-update tuple, and the post-update tuple. Since we consider non-cooperative sources, the data sources would never have any information on the data warehouse views. It is completely up to the data warehouse to interpret the update message received from the data source and to perform the appropriate view update.

The data warehouse needs to associate an update with a view through a searching process to see which view(s) should be updated. This is basically a matching of the relation specified in the update message to the relations in the data warehouse view. In all the IIVM systems reviewed, the data warehouse keeps track of the meta-data (e.g., relation identities) for all of its views. A sequential search is used to check which view(s) need to be updated. The following pseudo-code describes this process:

```
REMOVE (ΔRi) FROM UpdateMessageQueue[]
        LOOP
                IF (∃Ri ∈ ∀V)
                        ViewUpdate();
```

where
ΔR_i denotes an update at source i;
UpdateMessageQueue[] denotes an array containing the pending update messages;
ViewUpdate() denotes the function for performing view update.

The C-Strobe and SWEEP systems execute a sequential scan that includes all the views in the system. For our multi-agent system, the update notification is distributed to the

view agents to check in parallel. A view agent can proceed to check the following update in queue without waiting for the current update to complete.

3.5.2 Performing Sub-Queries

The SPJ views in the data warehouse contain relations that are distributed among the data sources. Consider a view definition involving relations R_1, R_2, R_3, … and each one resides in a different source. Source S_1 may be a data table that holds R_1 etc. In response to an update ΔR_1, the data warehouse dispatches a query $Q_1 = \Delta R_1 \bowtie R_2 \bowtie R_3 \bowtie \ldots$ to the data sources corresponding to the base relations R_2, R_3, … . It is the responsibility of the data warehouse to issue a query to each source that holds the required relation. Each query on a relation that is required to re-construct a view is called a sub-query.

In our example, a sub-query is required to get R_2 from S_2. Another sub-query to get R_3 from S_3 and so on. Therefore, if a view definition contains a *join* of k relations, the maximum number of sub-queries required to re-construct the view would be k - 1. The order of sub-queries has to be started from the updated relation and propagates according to the SPJ expression. This is required to provide sufficient query criteria, i.e., the link of the *join*. For example, if a query $Q_1 = R_1 \bowtie R_2 \bowtie \Delta R_3 \bowtie R_4 \bowtie R_5$ is required, the order of sub-queries would be R_2 (or R_4), then R_1 (or R_5). The direction, however, does not matter.

Let a general SPJ view be defined by $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n$ where S_1 contains R_1, S_2 contains R_2, S_n contains R_n, etc. The following pseudo-code describes this operation for right side of the sub-queries:

```
FOR (j = Position + 1; j ≤ NumberOfRelations; j++) DO
        SEND ΔV to Data Source j;
        RECEIVE ΔV from Data Source j;
END FOR
```

where

ΔV denotes new incremental view, $\Delta V = R_1 \bowtie .. \bowtie \Delta R_i \bowtie \ldots R_n$;
Position denotes the position of the updated relation in the SPJ expression;
NumberOfRelations denotes the number of relations in the view definition.

For instance, if an update ΔR_i occurs, Position = i, and NumberOfRelations = n

This process is required for any incremental view maintenance process. The procedure is common in C-Strobe, SWEEP and our multi-agent systems. However, there can be multiple extract agents in our multi-agent system to perform the sub-queries for different updates concurrently. This provides significant saving in process time and was tested in the Experimental Studies - Chapter 6.

3.5.3 Probing for Concurrent Updates

As discussed in the previous section, the data warehouse system must intelligently compensate for the changes in the source made by a concurrent update. Concurrent updates that occur at a relevant source could result in incorrect query answer, hence, incorrect final view.

If a concurrent update occurs at the source before the sub-query is made, compensation is necessary to revert to the pre-update source state. Local compensation proposed in the SWEEP system [7] eliminates the need for compensation queries. This compensation process involves the checking for concurrent update after each sub-query at a source. If a concurrent update were detected, the data warehouse would perform an operation opposite to the concurrent update to offset the effect. For example, if a concurrent insertion were detected, then, a deletion of the tuple would be applied to the query answer. If a concurrent deletion were detected, then, an insertion of the tuple would be applied to the query answer. The following pseudo-code illustrates the concurrent update detection and compensation process.

IF ($\exists \Delta R_j \in$ UpdateMessageQueue[])
THEN

$\Delta V = \Delta V - \Delta R_j \bowtie$ TempView

ENDIF

where
ΔV denotes new incremental view, $\Delta V = R_1 \bowtie .. \bowtie \Delta R_i \bowtie \ldots R_n$;
UpdateMessageQueue[] denotes an array containing the pending update messages;
TempView denotes the portion of view already constructed.

The SWEEP system made a significant improvement over the C-Strobe by eliminating compensation queries. Our multi-agent system adopts the local compensation method in SWEEP. In the SWEEP system, the data warehouse checks for concurrent update in the system update queue. In multi-agent system, updates are initially assigned to view agents and checking is performed on a view agent's own update queue. This is substantially smaller than the system update queue.

3.5.4 Updating of Warehouse Views

Once an incremental view is completely constructed by performing queries to all the relevant sources, the incremental view is inserted to or deleted from the data warehouse.
In SWEEP [7] and C-Strobe [37], IIVM is performed by the above steps running in a sequence. When an update notification message is received by the data warehouse, the system would search for the affected view, then perform the sub-queries one at a time with local compensation, and finally modify the warehouse view with the new view tuple. After that, the system would then proceed to the next update in the queue and follow the same steps. If an update affects multiple views, these systems would process these views in a queue.
In our multi-agent system, the update modification of warehouse views is performed by the view agents according to the arrival of update notifications from the sources. If an

update affects multiple views, the view agents cooperate to perform update modification simultaneously. This action improves data consistency in the data warehouse.

3.6 Special Features of Multi-Agent IIVM System

In this section, we shall highlight the features of the multi-agent system that help to minimize latency time and maximize scalability while maintaining data consistency. Chapter 4 analyzes in greater detail on how the multi-agent IIVM system could enforce the required consistency level.

3.6.1 Autonomy

An important feature of the multi-agent IIVM system is that views, queries, and sources are represented and managed autonomously by agents. Using the steps of IIVM discussed in the previous section, parallel processing can be used in:

- Associating Updates with Views

View agents check each update in separate, parallel processes. A view agent can perform the checking, leave the job to the blackboard and make itself available for the next update task or view writing task. A single view agent is responsible for a single view, therefore, conflicts due to write-lock on the view from another update would not occur.

- Performing Sub-Queries

Sub-queries are performed by agents in separate, parallel processes. Sub-queries for different update tasks can take place concurrently. The number of extract agents can be adjusted according to the computing and network resources.

3.6.2 Cooperation

The requirement of consistency requires the agents to cooperate since the IIVM steps are carried out by separate processes. Cooperation of agents is required for performing compensation and modifying warehouse views synchronously.

- Compensating for Concurrent Updates

Local compensation requires a linkage between the query and the view definition. This linkage requires cooperation between the view agents and the extract agents. Therefore, the extract agent must check with the view agent after each sub-query to compensate for any concurrent update. In this system, the link between the extract agent and view agent is called a Cooperation Domain. Once all the sub-queries are completed, the agents involved become free and the cooperation domain vanishes.

- Updating of Warehouse Views

Updating of warehouse views are performed by view agents. If an update affects multiple views, the view agents must synchronize their view writing processes such that multiple view consistency is maintained. This also requires the view agents to be in a cooperation domain for the synchronization. The cooperation domain vanishes once the view updates are completed. Multi-view consistency is discussed in Chapter 4.

3.7 Fuzzy Inference System

Since the possible number of connections to the source tables is controlled, the number of available extract agents is limited. In order to intelligently allocate the extract agents to more urgent tasks, a fuzzy system is applied in the blackboard.

One important application of a fuzzy rule set is to minimize the chance of interfering updates by minimizing the delay in sending sub-queries to the sources. Fewer interfering updates would improve the process time as compensation incurs extra process time.

Every data extract task is assigned an announcement priority that is generated by a fuzzy inference system (FIS) based on the fuzzy attributes supplied by the view agents and the update agents. The FIS uses a fuzzy rule set to fuzzify, infer, and defuzzify the fuzzy attributes to yield an announcement priority. The extract agent selects extract tasks from the blackboard based on the announcement priority.

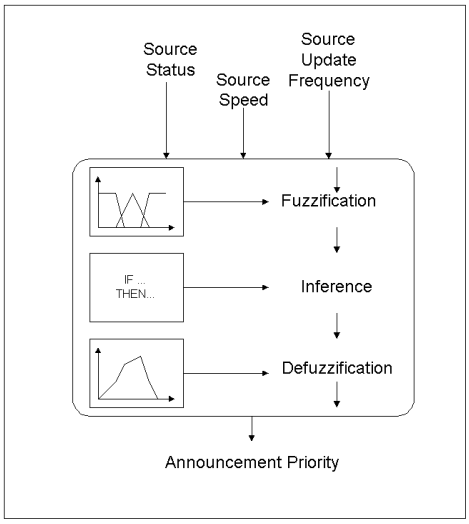

Figure 3-5 The process of fuzzy inference system

Fuzzy factors include:

(i) **Source Query Speed**

- A source may have heavy traffic, causing low query speed. An extract agent assigned to query such a source could be occupied for a long period of time.

(ii) **Source Status**

- A source may be in the process of being read or written. Exclusive locks (for writing) and shared locks (for reading) could be held at the source, which temporarily make the source unavailable.

(iii) **Source Update Frequency**

- A source with high update frequency has a greater chance of encountering interfering updates. Such sources should be queried immediately to avoid interfering updates.

(iv) **Number of Views in Cooperation Domain**

- A query request of a view update that involves a number of views in the cooperation domain should be given a higher priority. Delay of such request would not only cost delay in the current view update but also for the view updates in the cooperation domain.

An example of a fuzzy rule set is:

Fuzzy Rule 1: IF *Source Query Speed* is Low OR *Source Status* is Busy, THEN *Announcement Priority* is Low

Fuzzy Rule 2: IF *Source Update Frequency* is High OR *Number of Views in Cooperation Domain* is Many, THEN *Announcement Priority* is High

3.8 Failure Recovery

In database system theories, failure recovery deals with handling transactions when system crashes occur. The central issue is that the atomicity of every transaction must be maintained in case of a system crash: Either the whole transaction is carried out or none is carried out. The rollback operation is often required to undo the partially done transactions. In transaction systems, failing to recover an interrupted transaction could cause permanent error for the database that cannot be rectified later.

The failure recovery in warehouse view maintenance is different from that in conventional transaction processing. A failure could interrupt a view tuple assembly

process and leave it partially done. The point of interruption could be before the update notification is sent, during sub-queries, or at view modification. However, the views can always be recomputed from the source data. Therefore, the objectives of failure recovery in a data warehousing system should be failure detection and actions to maintain view consistency at any time, instead of recovering the interrupted operation.

There are many possible reasons for the failure of a source, such as network failure, hardware (disk storage), or software (database management system) problems. The type of source failure is usually unknown to the data warehouse. So, it is uncertain whether transactions are active or updates are sent from the failed source. It is not a good approach to attempt to recover the updates at the failed source during the failure. Our approach is to first maintain the warehouse views in a consistent old state. After the failure at the source or warehouse is resolved, the warehouse views are re-computed. In general, there are two types of failure possible in a data warehousing system:

Case I: Failure of Data Source

Case I (a): Source fails to send update notifications

Such failure is detected when an update agent fails to connect to or query the source, causing SQL Exception in the program. In that case, the IIVM system should stop and enter failure recovery mode (listed below) immediately. Otherwise, updates from other good sources would continue updating the warehouse views and data consistency would be violated.

Case I (b): Source fails to process sub-queries

Such failure is detected when an extract agent fails to connect to or get answer from the source, causing SQL Exception in the program. In that case, the IIVM system should stop and enter failure recovery mode immediately. The reason is the same as Case I (a).

Case II: Failure of Data Warehouse

Case II (a): Warehouse view cannot be updated

Such failure is detected when a view agent fails to connect to the warehouse or update the warehouse view, causing SQL Exception in the program. In that case, the IIVM system should stop and enter failure recovery mode immediately.

Failure Recovery Mode

The following steps are executed if the IIVM system enters the failure recovery mode:

1. The IIVM system quits all warehouse view assembly processes in progress.
2. View agents complete the started warehouse view modification processes if possible.
3. The warehouse views are kept in a consistent state, no update is allowed.
4. The IIVM system starts a full re-computation of warehouse views:
    - View agents post view definitions.
    - Extract agents compute each view completely by querying each source.
    - View agents assemble new views to replace existing views.

3.9 Summary

The IIVM process for a data warehousing system is investigated in detail in this chapter. Previous researches applied various types of compensation methods to rectify potential data inconsistency inherent in IIVM. Local compensation is the latest and the most efficient method for rectifying the problem in IIVM. Using the multi-agent architecture, the IIVM processes can be divided among various agents such that the level of parallel processing can be maximized. A blackboard architecture was adopted and three types of agents were used: (i) update agent, (ii) extract agent, and (iii) view agent. Through multicast messaging, the multi-agent IIVM system enables cooperation between the

autonomous agents when necessary. The advantages of the multi-agent system over the existing systems were discussed. A failure recovery scheme aims to maintain the warehouse views in an old state in case of system crash, while re-computing the warehouse views.

4 Data Consistency in IIVM

4.1 Introduction

In this chapter, we shall begin by discussing the general consistency requirements for a data warehousing system in Section 4.2. Section 4.3 investigates all the possible anomalies that could result from IIVM by studying the effects of insertion, deletion, and update. Section 4.4 specifies the necessary properties for an IIVM system from database theories, which are the design criteria for our multi-agent IIVM system. In Section 4.5, a proof will be used to show that our multi-agent system satisfies the completeness consistency level in every case of concurrent update. Section 4.6 provides a summary for this chapter.

4.2 Data Consistency Requirements for a Data Warehouse

In general, there are three levels of data consistency involved in data warehousing system, as illustrated in Figure 4-1.

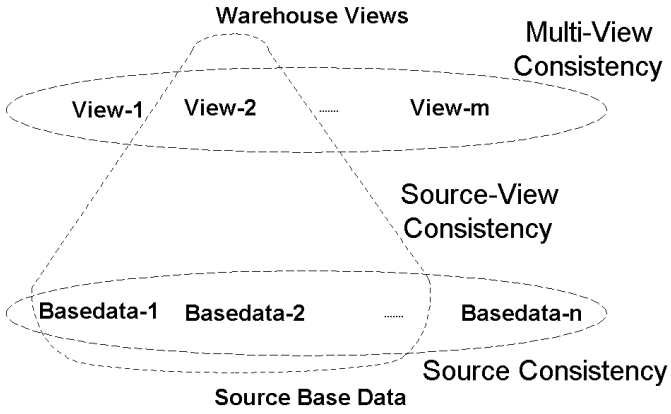

Figure 4-1 Three types of data consistency for a data warehousing system

Source consistency refers to consistency among the base data, and it is enforced by source transaction protocol (if any). There have been numerous researches on distributed database systems [32, 34] and they are not within the scope of this study. Source-View consistency refers to the consistency between a view at the warehouse and its base data at the sources and is enforced by the acquisition component of the warehouse. Multi-view consistency refers to the mutual consistency among views, which is also enforced by acquisition component of the warehouse.

In this section, we shall investigate both the source-view consistency and the multi-view consistency requirements associated with IIVM for a data warehouse.

4.2.1 Source-View Consistency

We will use an example to help illustrate the notion of source-view consistency.

Example 4-1

Using Example 3-3 with view $V_1 = R_1 \bowtie R_2$ again, Table 4.1 illustrates the IIVM process with the isolation property violated. Initially at time t_0, V_1 contains the correct tuples (1, 2, 3), (1, 2, 4). At t_1, tuple (2, 2) is inserted to R_1 and the update notification is sent to the

data warehouse. At t_2, tuple (2, 4) is removed from R_2. Since (2, 4) is not in R_2 for sub-query of the first insertion, the view tuple (2, 2, 4) is not inserted. At t_3, the system tries to delete both the view tuples (1, 2, 4) and (2, 2, 4), but (2, 2, 4) is missing. The tuple (2, 2, 4) should have existed in V_1 and this state is missing due to a concurrent deletion.

Time	Action	R_1 (X, Y)	R_2 (Y, Z)	$V_1 = R_1 \bowtie R_2$
t_0		(1, 2)	(2, 3), (2, 4)	(1, 2, 3), (1, 2, 4)
t_1	**insert** (R_1, (2, 2))	(1, 2), (2, 2)	(2, 3), (2, 4)	(1, 2, 3), (1, 2, 4)
t_2	**delete** (R_2, (2, 4))	(1, 2), (2, 2)	(2, 3)	(1, 2, 3), (1, 2, 4), (2, 2, 3)
t_3		(1, 2), (2, 2)	(2, 3)	(1, 2, 3), (2, 2, 3)

Table 4.1 IIVM process with isolation property violated

In Example 4-1, the final warehouse view is correct but one source state is lost (i.e., $(R_1 + \Delta R_1) \bowtie R_2$) and it would never be reflected in the warehouse view. The state loss may or may not be acceptable depending on the application's data consistency requirement.

Consistency level can be used to specify the consistency requirement for view maintenance. In general, four consistency levels can be considered, in ascending order of consistency level [37]: Convergence, Weak Consistency, Strong Consistency, and Completeness.

- **Convergence** requires only that the final view be consistent with the source data after all update activities have been completed.

- **Weak consistency** requires that each warehouse state reflects a valid state at each source but the warehouse state could reflect a different time state at each source.

- **Strong consistency** requires that each warehouse state reflects a set of valid source states, reflecting the global sequence of states for all sources, and that the order of the warehouse states matches the order of source actions.

- **Completeness** requires the views to be strongly consistent with the sources and that for each source state, there is a matching warehouse state. There is a complete order-preserving mapping between the states of the view and the states of the sources.

When a transaction happens at a source, the sub-queries due to that update should be done at the same state as the transaction source state. (query state = transaction state) If this is violated due to a concurrent update at any of the sources, the data warehouse must intelligently discard the effect of the concurrent update. Compensation is generally the method applied to remove the effect of concurrent update in the data warehouse.

For Example 4-1, the warehouse view must reflect the first state $(R_1 + \Delta R_1) \bowtie R_2$ in order to claim completeness. If weak consistency were the consistency requirement, the scenario in this example would have been acceptable.

However, as the previous examples showed, IIVM without compensation would not even satisfy the convergence requirement. If every source state has to be captured (completeness is enforced), then the effect of the concurrent update must be offset by compensation. In this thesis study, we shall aim to enforce the completeness consistency level. The local compensation method shall be applied. Previous researches such as the Strobe and SWEEP algorithms also enforce the completeness consistency level.

4.2.2 Multi-View Consistency

The discussion has so far been about the consistency of warehouse views with respect to the data sources. However, consistency among the views in the data warehouse is also important as some applications combine multiple warehouse views to generate information. If an update affects more than one view in the warehouse, all the affected

views should reflect the update at the same time, or at least with a small, controlled delay. We shall call this multi-view consistency.

All researches reviewed are concerned with consistency of warehouse views with respect to the data sources, without any regard of consistency among views. In all the algorithms reviewed, only a single view is considered in the system. The inconsistency among views is not controlled and is assumed to be small. The Painting Algorithm [36] was proposed to monitor the progress of every view update in the system by a coloring table to ensure that views are updated consistently. Our multi-agent IIVM system maintains also the multi-view consistency by cooperation between agents.

4.3 Anomalies of Concurrent Updates in IIVM

In this section, we shall investigate the possible cases of concurrent update in IIVM and summarize the consequence of every combination of concurrent updates. In essence, the concurrent update makes changes to the relation before the original update executes query on it, resulting in erroneous query answers. More detailed explanations and examples are given in Appendix A.

In general, the transactions at the source are insertion, deletion, and update. During these transactions, the possible concurrent transactions are also insertion, deletion, and update. This results in 9 possible cases of concurrent updates in IIVM. Table 4.2 summarizes the effects or anomalies.

The two possible anomalies that result in incorrect final views are "extra tuple" and "missing state". Extra tuple means that the interference by a concurrent update causes the resultant view to contain more tuple(s) than it should. Missing state means that a state that existed at the source is never reflected in the warehouse view, even though the

resultant warehouse view is correct. This is not tolerable if the completeness consistency requirement is enforced.

An update operation is basically the combination of a deletion followed by an insertion. Notice that whenever the interfering operation is an update, the effect is equivalent to the combination of an interfering insertion and an interfering deletion.

Operation	Interfering Operation	Anomalies
Insertion	Insertion	Extra Tuple
	Deletion	Missing State
	Update	Extra Tuple + Missing State
Deletion	Insertion	No Anomaly
	Deletion	Extra Tuple
	Update	Extra Tuple
Update	Insertion	Extra Tuple
	Deletion	Extra Tuple + Missing State
	Update	2 Extra Tuples + Missing State

Table 4.2 Summary of the effects of concurrent updates

4.4 Design Principles from Database Theory

In this section, we shall discuss the design requirements of data warehousing system based on the ACID design principle of a database system: Atomicity, Consistency, Isolation, and Durability.

Atomicity -

In IIVM, an update from one source may require the updated data to be integrated with data from several other sources. However, gathering the data corresponding to an update is not an atomic operation. No matter how fast the warehouse generates the appropriate query and sends it to the sources, receiving the answer is not atomic since parts of it come from different, autonomous sources. Nonetheless, the view should be updated as if all of

the sources were queried atomically. The view maintenance system must make the queries appear atomic by processing them intelligently at the warehouse.

Consistency -

In this thesis study, we shall adopt the highest consistency requirement of completeness. This implies that each warehouse view is required to capture every source state in the time order of updates. Due to this stringent consistency goal, compensation is required for every case discussed in the previous section since not even state loss is tolerated. The anomalies discussed in the previous section must be alleviated. In addition we shall aim to achieve multi-view consistency. This entails controlling the updating of warehouse views such that the warehouse views derived from the same relation must always reflect the same source state.

Isolation -

We assumed that the data sources are distributed and autonomous. As a result, updates that occur at the data sources are concurrent processes from the viewpoint of the data warehouse. In concurrent control theory for transaction databases, serialization is the key to managing concurrent processes appropriately. A concurrent schedule ensures consistency if and only if it has an equivalent serial schedule that is without conflict. In IIVM, the data warehouse must receive the concurrent updates and generate a serial schedule to update the warehouse views according to the order of update message arrival.

Durability -

We assumed that once an update is finished on a warehouse view, it is automatically committed. Once an update is committed, it is stored in permanent data storage. Any system failure after the update would not affect the updated information.

4.5 Satisfaction of Consistency Requirement

The multi-agent IIVM system aims to satisfy the completeness consistency level. This level requires that, for each global source state, there is a matching warehouse state. In IIVM, the process of updating a warehouse view involves querying the relevant sources, so it is not an atomic operation. The problem is that the transaction source state could be altered by a concurrent update before the source is queried. The effect of the concurrent update would be incorrectly captured in the original update's query answer. The proof in this section shows that under all concurrent update scenarios, the multi-agent IIVM system ensures that the effect of the concurrent update is properly compensated. The data warehouse is able to capture the original source state in every case since compensation corrects the queried answer when a concurrent update occurred at the source. Therefore, the completeness consistency level is maintained under any scenario.

Proof of Completeness

Hypothesis 4-1

The Multi-Agent System (MAS) algorithm is complete.

Assume that the list of updates at the warehouse in their receiving order is U_1, U_2, … Let R be the serial schedule where transactions are executed in this order. Then, R is a serial source schedule that is equivalent to the actual execution. Each warehouse state ws_x represents the contents of the warehouse. The warehouse state changes whenever a view is updated. We consider one view V at the warehouse, which is defined over a set of base relations at one or more sources. The view at state ws_j is $V(ws_j)$. Let the warehouse state after processing U_x be ws_x and let the source state ss_x = result (U_1, …, U_x), where ss_x is a consistent state of all sources.

Let $MV[ws_x]$ be a materialized view at state ws_x. $MV[ws_x]$ is the same as $V[ws_x]$ in the consistency definition. We assume that $MV[ws_0] = V[ss_0]$. We call the update U_i the

current update, and all the discussions below are within the context of processing U_i. Assume we have $MV[ws_{i-1}] = V[ss_{i-1}]$. That is, MV at state ws_{i-1} correctly reflects the source state ss_{i-1}. We shall prove by induction that $MV[ws_i] = V[ss_i]$ is the materialized view after processing U_i using MAS, under all possible concurrent update scenarios.

In order to prove that Hypothesis 4-1 is true, we need to prove the following statement:

Hypothesis 4-2

If $MV[ws_{i-1}] = V[ss_{i-1}]$, then $MV[ws_i] = V[ss_i]$, where $MV[ws_i]$ is the materialized view after processing U_i under all possible concurrent update scenarios using MAS.

Proof:

Assume view $V = r_1 \bowtie r_2 \bowtie \ldots \bowtie r_m \bowtie \ldots \bowtie r_n$
From basic incremental view update theory:

$$V[ss_i] = V[ss_{i-1}] \cup V\langle U_i\rangle[ss_{i-1}] \quad (7)$$

$$MV[ws_i] = MV[ws_{i-1}] \cup V\langle U_i\rangle[ws_{i-1}] \quad (8)$$

Equation (7) indicates that the view V at source state ss_i is the union of the view V at source state ss_{i-1} and the view tuple due to update U_i evaluated at source state ss_{i-1}. Equation (8) indicates that the materialized view MV at warehouse state ws_i is the union of the view MV at warehouse state ws_{i-1} and the view tuple due to update U_i evaluated at warehouse state ws_{i-1}. We shall prove that Hypothesis 4-2 is true for two cases: (i) U_i is an insertion; (ii) U_i is a deletion.

Case I: U_i is an insertion

Assume $U_i = + t_m$ where t_m is a tuple of relation r_m in view V defined above.
In MAS, update U_i is stored in a buffer according to the order of message arrival. The view agent verifies whether the update affects its view. The extract agent sends a query to the source for each relation r in the definition of the view. We define $Q_{i,j}$ as such a

query, where j denotes the sub-query source. We also define $A_{i,j}$ as the answer for query $Q_{i,j}$. We shall consider all possible occurrences of concurrent update.

Case I (a): No U_{i+1} between U_i and $A_{i,j}$ at any source

In MAS, $Q_{i,j}$ is evaluated at each source j. Since there is no concurrent update at any source, the transaction state equals the queried state for each of these sources. This results in $A_{i,j} = V_j<U_i>[ss_{i-1}]$, which is the view tuple due to update U_i after querying source j. After all required sources are queried, we have $A_i = V<U_i>[ss_{i-1}]$. That is, the final query answer equals to the view tuple due to update U_i evaluated at source state ss_{i-1}.

The subsequent updates U_{i+1}, U_{i+2}, … happen after ss_{i-1} and must arrive at the warehouse after U_i because U_i happens at state ss_{i-1} and we assumed no message transmission disorder. In MAS, U_{i+1} must be written to the warehouse view(s) after U_i is written. (Process Blackboard, Line 6) Therefore, the warehouse state must still be ws_{i-1} before U_i is written and answer A_i must be applied to the warehouse at the warehouse state ws_{i-1}. As a result, the view tuple applied to the warehouse due to U_i is $A_i = V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$.

Since we assumed that $V[ss_{i-1}] = MV[ws_{i-1}]$ and we showed that $V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$. By Equations (7) and (8), $MV[ws_i] = V[ss_i]$.

Case I (b): At least one insertion between U_i and $A_{i,j}$ at source j

In MAS, $Q_{i,j}$ is evaluated at each source j. A concurrent update $U_{i+1} = + t_p$ happens at source j between U_i and $A_{i,j}$. This causes the transaction state to be unequal to the queried state for source j. This results in $A_{i,j} = V_j<U_{i+1}>[ss_i]$, which includes t_p. Since t_p does not belong to the transaction state, this extra tuple t_p always introduces duplicate view tuples (Table 4.2).

In MAS, the extract agent checks with view agent in cooperation domain after receiving each $A_{i,j}$. The extract agent detects U_{i+1} (insert of t_p) and deletes all the view tuples with t_p from $A_{i,j}$ (Process ExtractAgent Line 8). There can be more than one concurrent update and they are all compensated in the same way. After all required sources are queried, the final query answer must be $A_i = V<U_i>[ss_{i-1}]$ since compensation reverts $A_{i,j}$ to the transaction state for each source j queried.

We shall now reason that this answer must be applied to the warehouse at the warehouse state ws_{i-1}. The subsequent updates U_{i+1}, U_{i+2} … happen after ss_{i-1} and must arrive at the warehouse after U_i because U_i happens at state ss_{i-1} and we assumed no message transmission disorder. In MAS, U_{i+1} can only be written to the warehouse after U_i is written. Therefore, the warehouse state must remain at ws_{i-1} when U_i is applied and thus the view tuple applied to the warehouse is $V<U_i>[ws_{i-1}]$. Finally, therefore, the view tuple applied to the warehouse due to U_i is $A_i = V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$.

Since we assumed that $V[ss_{i-1}] = MV[ws_{i-1}]$ and we showed that $V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$. By Equations (7) and (8), $MV[ws_i] = V[ss_i]$.

Case I (c): At least one deletion between U_i and $A_{i,j}$ at source j

This case is similar to Case I(b). In MAS, $Q_{i,j}$ is evaluated at each source j. Suppose a concurrent update $U_{i+1} = - t_p$ happens at source j between U_i and $A_{i,j}$. This causes the transaction state to be unequal to the queried state for source j. This results in $A_{i,j} = V_j<U_{i+1}>[ss_i]$, which is missing t_p. Since t_p belongs to the transaction state, the absence of t_p always results in missing view tuples (Table 4.2).

In MAS, the extract agent checks with view agent in cooperation domain after receiving each $A_{i,j}$. The extract agent detects U_{i+1} (deletion of t_p) and then it adds view tuples with t_p to $A_{i,j}$ (Process ExtractAgent Line 8). There can be more than one concurrent update and they are all compensated in the same way. After all required sources are queried, the

final query answer must be $A_i = V<U_i>[ss_{i-1}]$ since compensation reverts $A_{i,j}$ to the transaction state for each source j queried.

We shall now reason that this answer must be applied to the warehouse at the warehouse state ws_{i-1}. The subsequent updates, including all concurrent updates that happen after ss_{i-1}, must arrive at the warehouse after U_i because U_i happens at state ss_{i-1} and we assumed no message transmission disorder. In MAS, U_{i+1} can only be written to the warehouse after U_i is written. (Process Blackboard, Line 6) Therefore, the warehouse state must remain at ws_{i-1} when U_i is applied and thus the view tuple applied to the warehouse is $V<U_i>[ws_{i-1}]$. Finally, therefore, the view tuple applied to the warehouse due to U_i is $A_i = V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$.

Since we assumed $V[ss_{i-1}] = MV[ws_{i-1}]$ and we showed that $V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$. By Equations (7) and (8), $MV[ws_i] = V[ss_i]$.

Case II: U_i is a deletion

Assume $U_i = - t_m$ where t_m is a tuple of relation r_m in view V defined above. In MAS, update U_i is stored in a buffer according to the order of message arrival. The v-agent verifies whether the update message is associated with its view. Then, the e-agent sends a query to the source of each relation r in the definition of the view.

Case II (a): No U_{i+1} between U_i and $A_{i,j}$ at any source

This case is identical to Case I(a) except that U_i is now a deletion. As before, $Q_{i,j}$ is evaluated at each source j. Since there is no concurrent update at any source, the transaction state equals the queried state for each of these sources. By the same argument in Case I(a), the view tuple applied to the warehouse due to U_i is $A_i = V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$.

Since we assumed that $V[ss_{i-1}] = MV[ws_{i-1}]$ and we showed that $V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$. By Equations (7) and (8), $MV[ws_i] = V[ss_i]$.

Case II (b): At least one insertion between U_i and $A_{i,j}$ at source j

This case is similar to Case I(b). In MAS, $Q_{i,j}$ is evaluated at each source j. $U_{i+1} = + t_p$ happens at source j between U_i and $A_{i,j}$. This causes the transaction state to be unequal to the queried state for source j. This results in $A_{i,j} = V_j<U_{i+1}>[ss_i]$, which includes t_p. Since t_p is inserted at or after ss_i, its update notification must be subsequent to U_i. As a result, view tuple(s) with t_p must not exist in the warehouse. In essence, the query answer contains these view tuple(s) to be deleted but they do not exist in the warehouse.

In MAS, the extract agent checks with the view agent in the cooperation domain after receiving each $A_{i,j}$. The extract agent detects U_{i+1} (insert of t_p) and deletes all the view tuples with t_p from $A_{i,j}$. (Process ExtractAgent Line 8) There can be more than one concurrent update and they are all compensated in the same way. After all required sources are queried, the final query answer must be $A_i = V<U_i>[ss_{i-1}]$ since compensation reverts $A_{i,j}$ to the transaction state for each source j queried.

We shall now show that this answer must be applied to the warehouse at the warehouse state ws_{i-1}. The subsequent updates U_{i+1}, U_{i+2} … happen after ss_{i-1} and must arrive at the warehouse after U_i because U_i happens at state ss_{i-1} and we assumed no message transmission disorder. In MAS, U_{i+1} can only be written to the warehouse after U_i is written. Therefore, the warehouse state must remain at ws_{i-1} when U_i is applied and thus the view tuple applied to the warehouse is $V<U_i>[ws_{i-1}]$. Finally, therefore, the view tuple applied to the warehouse due to U_i is $A_i = V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$.

Since we assumed $V[ss_{i-1}] = MV[ws_{i-1}]$ and we showed that $V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$. By Equations (7) and (8), $MV[ws_i] = V[ss_i]$.

Case II (c): At least one deletion between U_i and $A_{i,j}$ at source j

This case is similar to Case I(c). In MAS, $Q_{i,j}$ is evaluated at each source j. Suppose a concurrent update $U_{i+1} = - t_p$ happens at source j between U_i and $A_{i,j}$. This causes the transaction state to be unequal to the queried state for source j. This results in $A_{i,j} = V_j<U_{i+1}>[ss_i]$, which is missing t_p. Since t_p exists in state ss_{i-1} and no update after U_i is processed, the warehouse must contain view tuple(s) with t_p. In essence, the update U_i is supposed to delete view tuple(s) with t_p from the warehouse but $A_{i,j}$ is missing t_p, leaving extra tuples in the warehouse view.

In MAS, the extract agent checks with the view agent in the cooperation domain after receiving each $A_{i,j}$. The extract agent detects U_{i+1} (deletion of t_p) and adds view tuple(s) with t_p to $A_{i,j}$ (Process ExtractAgent Line 8). There can be more than one concurrent update and they are all compensated in the same way. After all required sources are queried, the final query answer must be $A_i = V<U_i>[ss_{i-1}]$ since compensation reverts $A_{i,j}$ to the transaction state for each source j queried.

We now show that this answer must be applied to the warehouse at the warehouse state ws_{i-1}. The subsequent updates, including all concurrent updates that happen after ss_{i-1}, must arrive at the warehouse after U_i because U_i happens at state ss_{i-1} and we assumed no message transmission disorder. In MAS, U_{i+1} can only be written to the warehouse after U_i is written. Therefore, the warehouse state must remain at ws_{i-1} when U_i is applied and thus the view tuple applied to the warehouse is $V<U_i>[ws_{i-1}]$. Finally, therefore, the view tuple applied to the warehouse due to U_i is $A_i = V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$.

Since we assumed $V[ss_{i-1}] = MV[ws_{i-1}]$ and we showed that $V<U_i>[ss_{i-1}] = V<U_i>[ws_{i-1}]$, by Equations (7) and (8), $MV[ws_i] = V[ss_i]$.

From Case I and Case II, we proved $MV[ws_i] = V[ss_i]$ for any update U_i and any possibility of concurrent update. We establish a one to one relationship between the

sequence of warehouse states and a sequence of consistent source states. Therefore, the multi-agent IIVM algorithm is complete.

4.6 Summary

The general types and levels of consistency were discussed. The multi-agent IIVM system of this thesis was designed to satisfy the completeness consistency requirement. While other systems generally ignore the multi-view consistency requirement, our system also aims to satisfy this requirement. The multi-agent IIVM system was proven by induction to be complete. It was shown that under all possible concurrent update interference scenarios, the multi-agent system is able to revert the view state to reflect the correct source state by using compensation. Only Source-View Consistency was considered since Multi-View Consistency is enforced by cooperation, which is trivial.

5 System Implementation

5.1 Introduction

A prototype multi-agent IIVM system was implemented to verify the design and to evaluate the performance. In this chapter, the prototype system is discussed. Section 5.2 gives an overview of the prototype system including its purpose, functionalities, and limitations. Section 5.3 describes the data source and the data warehouse used in the prototype system. Section 5.4 discusses the different modules composing the multi-agent IIVM system. Section 5.5 gives an overview of the tools used to construct the prototype system. Section 5.6 concludes the chapter.

5.2 Overview of Prototype System

Our main purpose for building a prototype system is to measure the improvement in performance and verify the system using real databases as data sources.

The prototype system consists of three primary components, working as a three-tier system. The data warehouse is at the top level and it contains several relational tables. These relational tables represent the materialized views. The data sources are at the bottom level. The data sources are relational tables stored in different databases and these relational tables are subject to updates. The multi-agent IIVM system is at the middle level and it is the software that manages the transfer of updates from the bottom level to the top level. Update notifications from the data sources are received by the multi-agent IIVM system and are applied to the data warehouse views appropriately. The multi-agent IIVM system is created entirely by Java multi-threaded programming.

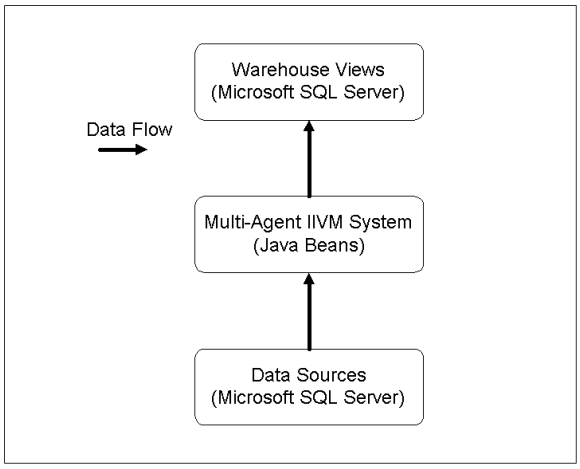

Figure 5-1 Prototype System Architecture

The prototype system has some slight modifications compared to the IIVM system described in this thesis to simplify implementation. The modifications and assumptions made in the prototype system are:

1. The prototype system uses a trigger and polling mechanism to simplify the update message dispatching described in this thesis.

2. The prototype system assumes that the data network is a no loss network without message transmission disorder. That is, the messages are guaranteed to be received at the destinations in the same sending order by the sources.

3. The prototype system assumes the definition for all warehouse views are simple SPJ expressions. There are no aggregations such as average, maximum, and minimum.

5.3 Data Source and Data Warehouse

The data sources used in the data warehouse prototype are relational tables in Microsoft SQL Server 2000. Databases were implemented to represent different data sources. The effect would be identical to distributed sources.

The choice of Microsoft SQL Server 2000 was based on its availability for this project. It has nothing particular that makes it suitable for this project. The basic requirements for the data source are that it must be able to execute SQL statements and it must be a JDBC or an ODBC data source. This is because the multi-agent IIVM system is created in Java and it can only communicate with a JDBC data source or an ODBC data source through the ODBC-JDBC bridge. JDBC and ODBC will be discussed later in this chapter.

5.3.1 Data Source

Change detection is the process of monitoring the source for changes to the data and propagating those changes to the data warehouse. This functionality relies on translation since 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.

Several approaches can be considered for change detection at the sources:

- Cooperative sources: Sources that provide active database capabilities, so that notifications of changes 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 a monitor to query the information at the source, so that periodic polling can be used to detect changes of interest.

- Snapshot sources: Periodic dumps, or snapshots, of the data are provided off-line, and changes are detected by comparing successive snapshots.

Cooperative sources require the sources to initiate update notification and send it to the warehousing system. This approach is useful when all the data sources in the system are known and programmable. Snapshot sources are more sophisticated and are usually implemented by specialized software packages.

The change detection mechanism adopted by our prototype system is a combination of logged source and queryable source. A trigger is installed in each source table to track any changes in the table. A log table in each database holds information about the changes produced by the triggers. The warehousing system makes queries to the log table periodically to poll the changes.

The trigger for each relational table is designed to store the change information in a log file in the same source database. The information includes database name, table name, attribute name, old value, and new value. No translation is necessary as a standardized format is used for storing the change information.

5.3.2 Data Warehouse

The data warehouse stores materialized views in the form of relational tables. The relational tables are defined by the SQL VIEW statements and the actual data are retrieved and stored as materialized views.

The data warehouse is implemented as a database in the Microsoft SQL Server. Each relational table in the data warehouse database represents a materialized view. The

definition of a view is created using an SPJ expression, implemented by SQL VIEW statement:

CREATE VIEW <view name> AS
SELECT <relations>
FROM <table name>
WHERE <select conditions>

The attribute name used in the data warehouse can be different from the corresponding attribute name used at the data source. When a view is added to the system, a view agent is created which holds the translation. In other words, the view agent holds the meta-data for both the source data and the warehouse data.

5.4 Multi-Agent IIVM System Modules

In this section, the different Java classes comprising the multi-agent IIVM system are discussed. For each class description, the constructor method and the run method are omitted for simplicity. The run method is required for each thread class in Java and it is the main control for each thread. Figure 5-2 is the class diagram for the multi-agent IIVM system.

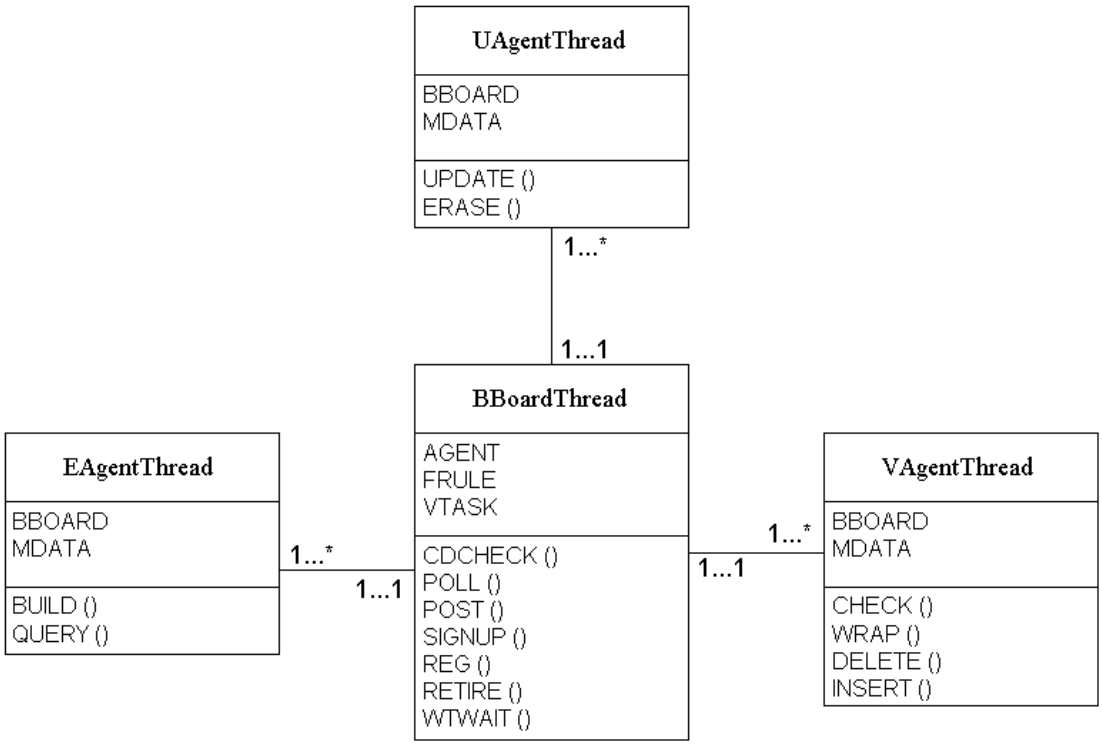

Figure 5-2 Class diagram of prototype system

5.4.1 Update Agent Module

The update agent is implemented by a Thread class named UAgentThread class. This class contains methods for managing update notifications from the source.

Class UAgentThread

Methods:

UPDATE: to make connection to the source and query the log table for update.

ERASE: to erase the record from the log table after the update is received.

Properties:

BBOARD: the blackboard used by this update agent.

MDATA: the object that holds the meta-data of a data source.

5.4.2 View Agent Module

The view agent is implemented by a Thread class named VAgentThread class. This class contains methods for processing the update notifications and modifying the warehouse views.

Class VAgentThread

Methods:
CHECK: to detect if the current update affects the view.
WRAP: to produce a task request.
DELETE: to delete view tuple(s) from the view.
INSERT: to insert a view tuple(s) into the view.

Properties:
BBOARD: the blackboard used by this view agent.
MDATA: the object that holds the metadata of the data sources and the warehouse view.

5.4.3 Extract Agent Module

The Extract Agent is implemented by a Thread class named EAgentThread class. This class contains methods for performing sub-queries and checking for concurrent updates.

Class EAgentThread

Methods:
BUILD: to manage the assembly processes of a view tuple.

QUERY: to perform query on a source table.

Properties:
BBOARD: the blackboard used by this extract agent.
MDATA: the object that holds the meta-data of a data source.

5.4.4 Blackboard Module

The Blackboard is implemented by a Thread class named BboardThread class. This class contains methods for managing the update tasks and messages from the agents. These methods are public methods that are available for any agent in the system to call.

Class BboardThread

Methods:
CDCHECK: to check if there is any pending action due to the cooperation domain of the calling agent.
POLL: to check if there is pending update task.
POST: to post update task to the blackboard.
SIGNUP: to sign up for an update task.
REG: to register the agent to the blackboard system.
RETIRE: to remove the agent from the blackboard system.
WTWAIT: to block until all view agents in the cooperation domain are allowed to update the warehouse view.

Properties:
AGENT: a list of all the agents registered in the system.
FRULE: a fuzzy rule adopted by the blackboard in extract task allocation.
VTASK: the queue of update tasks arranged in the order of notification arrival.

5.5 Software Implementation

In this section, the software tools and standards used in creating the prototype system shall be discussed.

5.5.1 SQL and ODBC

SQL is a standardized language used to create, manipulate, examine, and manage relational databases. SQL was standardized in 1992 so that a program could communicate with most database systems without having to change the query commands. However a connection must be made to a database before sending SQL commands, and each database vendor has a different interface to do so, as well as different extensions of SQL.

ODBC (Open Database Connectivity), a C-based interface to SQL-based database engines, provides a consistent interface for communicating with a database and for accessing the database meta-data (for example, information about the database system vendor, how the data is stored). Individual vendors provide specific drivers or "bridges" from ODBC to their particular database management system. Consequently, ODBC and SQL enable connection to a database and manipulation in a standard way. ODBC has become an industry standard.

5.5.2 Java JDBC

Although SQL is well suited for manipulating databases, it was not designed to be a general application language; rather, it was intended to be used only as a means of communicating with databases. The multi-agent IIVM system was created in Java and a driver is needed to feed SQL statements to a database and process results for data manipulation.

A JDBC driver is a class that implements the JDBC Driver interface and understands how to convert program requests (typically SQL queries) for a particular database. Within JDBC there are four particularly important classes: DriverManager, Connection, Statement, and ResultSet. Each class relies on its previous class for instance creation and corresponds to an indispensable phase of database access:

- The DriverManager object loads and configures the database driver on the client.
- The Connection object performs connection and authentication to the database server using URLs.
- The Statement object moves SQL to the database engine for preprocessing and eventually execution.
- The ResultSet object allows for the inspection of results from the executed query.

In a typical application, a Driver is passed to the DriverManager to obtain a Connection. In the system prototype, the target data source (Microsoft SQL Server) is an ODBC data source. In order to connect to an ODBC source from a Java program, JdbcOdbcDriver must be used. A Statement is then created and used to update the database or execute a query. A query returns a ResultSet containing the requested data. DatabaseMetaData and ResultSetMetaData classes are available to provide information about a database or a ResultSet.

The syntax for connecting to a database as used in the system prototype is as follows:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
Connection connection = DriverManager.getConnection("jdbc:odbc:<database name>",
"<user name>", "<password>");
Statement stmt = connection.createStatement();
ResultSet resultset = stmt.executeQuery(<SQL statements>);
```

The ResultSet object "resultset" contains the answer of the SQL query.

5.5.3 Java Multi-Threaded Architecture

A sequential program is one that has a beginning, an execution sequence, and an end. At any given time during the runtime of the program, there is a single point of execution. A thread is similar to the sequential programs. A single thread also has a beginning, a sequence, and an end, and at any given time during the runtime of the thread, there is a single point of execution. However, a thread itself is not a program; it cannot run on its own. Rather, it runs within a program. In essence, a thread is a single sequential flow of control within a program.

A thread is sometimes called a lightweight process. A thread is similar to a real process in that a thread and a running program are both a single sequential flow of control. However, a thread is considered lightweight because it runs within the context of a full-blown program and takes advantage of the resources allocated for that program and the program's environment.

The multi-agent IIVM system is a multi-threaded system. Each agent runs on a separate thread during the operation. A multi-threaded architecture is essential for a multi-agent system because it makes parallel processing possible. The blackboard runs on a separate thread as well such that message management is independent of the agents' processes.

A Java thread must be in one of four states (Figure 5-3): New Thread, Runnable, Not Runnable, and Dead. New Thread is the state when the thread is instantiated. Dead is the state when the thread is terminated. Runnable is the state when the thread is in its flow of sequence. Not Runnable is the state when the thread is blocked and it can be resumed only if a specified condition is satisfied.

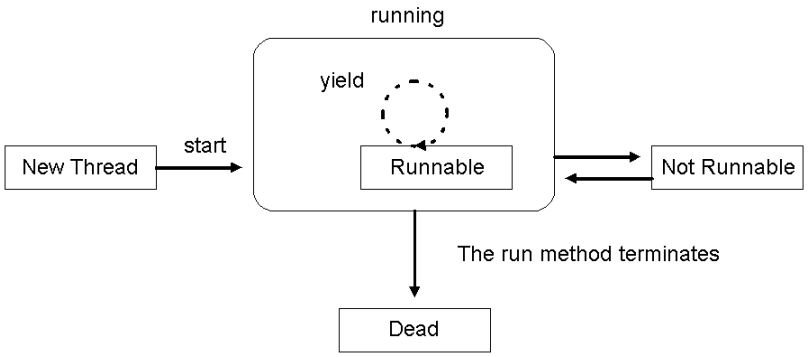

Figure 5-3 Possible states of a Java thread

With these four possible states, an agent thread must be put into Not Runnable state when there is no pending task since a non-busy agent thread should not be terminated. In the multi-agent IIVM system, when an agent thread polls the blackboard but there is no pending task, the agent thread will be blocked (Not Runnable state) until a task exists. The agent thread can return to the Runnable state only when a task exists on the blackboard. In other words, the agent thread is continuously blocked by the blackboard when it is non-busy. The BBoardThread methods POLL and WTWait are such blocking methods. In Java, this type of blocking method is called the synchronized method.

5.5.4 Java Beans

A Java Bean is a collection of one or more Java classes that serves as a self-contained, reusable component. A Java Bean can be a discrete component used in building a user interface, or a non-UI component such as a data module or computation engine. More specifically, a Java Bean is a reusable software component that can be visually manipulated in builder tools. In effect, Beans publish their attributes and behaviors through special method signature patterns that are recognized by beans-aware application construction tools.

In the multi-agent IIVM system, all agents and the blackboard are implemented as Java Beans. These Java Beans are reusable in other systems, such as a complete data warehousing system equipped with OLAP and data mining tools. Using a bean-aware application such as JBuilder, the agent and the blackboard Java Beans can be integrated into the complete data warehousing system to work with other Java Bean components.

5.6 Summary

The software configuration and the tools used to create the multi-agent IIVM prototype were discussed. In contrast to previous researches, which used custom-made data sources and data warehouse, we adopted a commercially available SQL data server. Hence, some functions available in previous research, such as the data change monitoring mechanism at the sources, were absent, but the data sources in this study were more realistic. The prototype system is a Java multi-threaded program connecting to Microsoft SQL Server databases using the JDBC-ODBC bridge.

6 Experimental Studies

6.1 Introduction

The performance of the IIVM system for a data warehouse can be measured by the currency of its views. The currency is determined by the processing time for performing view maintenance and it is the time latency of a view update. Other qualitative performance criteria include scalability, portability, and controllability. In this chapter, we shall provide a quantitative analysis to evaluate system performance. By introducing a multi-agent framework, we aimed to maximize parallel processing in view maintenance. The gain in efficiency will be reflected by the reduction of processing time, hence the time latency of view update.

In order to evaluate time latency, a series of tests were performed using the software prototype to determine the time latency for a view update given a specific set of operation parameters. Other published algorithms were emulated in the same environment and test results were collected for comparison. In the following sub-sections, we shall discuss the test results.

As mentioned in Chapter 2, almost all existing data warehousing systems currently adopt DIVM. Before comparing the multi-agent IIVM system with other IIVM systems, it is important to justify the need for using IIVM by comparing the performance of IIVM and DIVM. Therefore, part of the testing compared the data warehouse availability for IIVM and DIVM. A DIVM system was emulated in the same setup.

6.2 Other IIVM Systems

To determine the performance of the multi-agent system, it is important to provide a comparison with the existing systems. The most popular algorithms published on this subject are C-Strobe and SWEEP. Other systems cannot be used for a variety of reasons, such as the completeness consistency level is not enforced or the system is not designed for distributed sources. C-Strobe and SWEEP enforce the completeness consistency level and the data warehouse model adopted by this study was shared by both systems. Therefore, they are the most suitable systems for direct comparison.

6.2.1 C-Strobe

MAS and C-Strobe can be compared by the following table:

	C-Strobe	MAS
Consistency Level	Completeness	Completeness
Parallel / Sequential	Sequential	Parallel
Key Assumption	Assume key available for each relation in each view	No key assumption
Detection of Concurrent Updates	Assume the source to carry all the transactions on view in a log table	No assumption on source. Concurrent updates are detected entirely in the warehouse
Compensation Queries	Compensation queries required for every concurrent update detected	No compensation queries required
Message Transmission Disorder	Assume no message transmission disorder	Assume no message transmission disorder

Table 6.1 Comparison of MAS and C-Strobe algorithms

In C-Strobe, each update is processed completely before starting the following update in the queue. That includes all the compensation queries that may be required. In this sense, C-Strobe is a sequential system. C-Strobe checks for concurrent updates only after

the view tuple is fully assembled, which is the reason why compensation queries to the sources are needed to assemble the compensation view. Further compensation may be necessary due to concurrent updates during the compensation queries. In both systems, it is assumed that there is no message transmission disorder (messages arrive in the same order as transactions at the sources). This can be rectified by incorporating time stamps at the sources [28] and it is not considered in this thesis project. Example 3-4 in Section 3.4.6 would have been done by C-Strobe as illustrated by the following diagram:

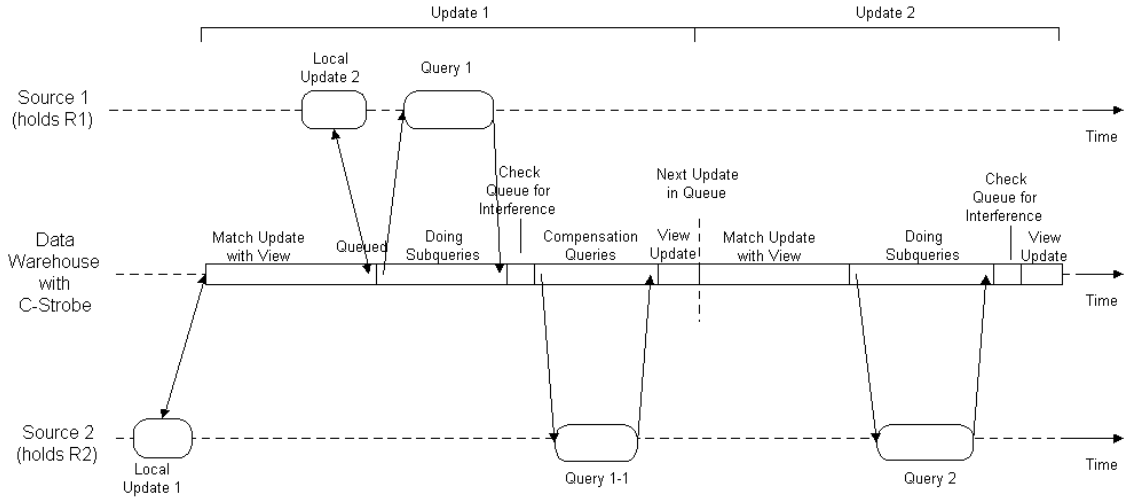

Figure 6-1 Sequence diagram for C-Strobe example

6.2.2 SWEEP

MAS and SWEEP are compared in the following table:

	SWEEP	MAS
Consistency Level	Completeness	Completeness
Parallel / Sequential	Sequential	Parallel
Key Assumption	No key assumption	No key assumption
Detection of Concurrent Updates	Concurrent updates are detected entirely in the warehouse	Concurrent updates are detected entirely in the warehouse
Compensation Queries	Local compensation by compensating each sub-query. No compensation queries required.	Local compensation by compensating each sub-query. No compensation queries required.
Message Transmission Disorder	Assume no message transmission disorder	Assume no message transmission disorder

Table 6.2 Comparison of MAS and SWEEP algorithms

The only difference between SWEEP and MAS is that SWEEP handles each update in a strict sequence. That is, a view update is completed before the next one is processed. Sub-queries are done in a sequence. Other than that, the MAS adopted similar IIVM approach in SWEEP. Example 3-4 in Section 3.4.6 would have been done by SWEEP as illustrated by the following diagram:

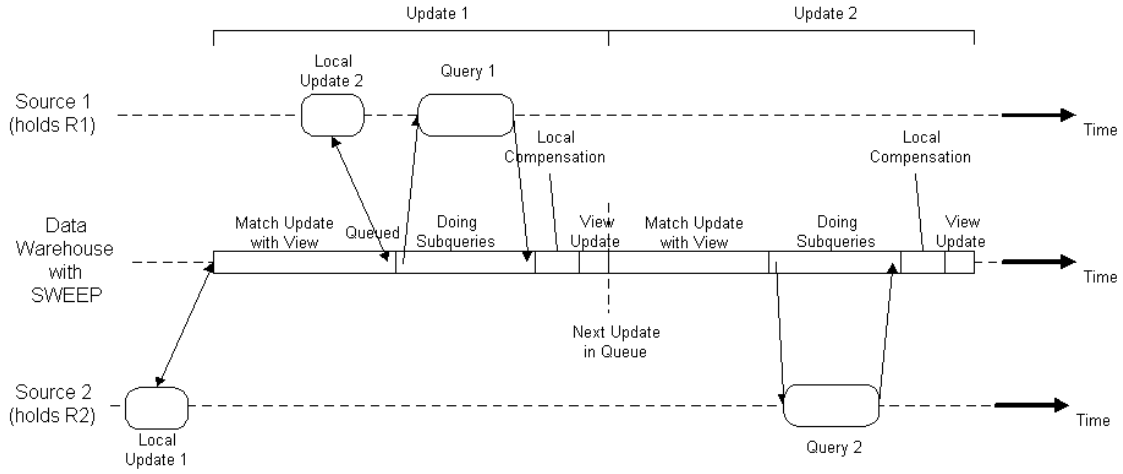

Figure 6-2 Sequence diagram for SWEEP example

6.3 Performance Test Results

In this section, we shall discuss the test results obtained by using the multi-agent software prototype. The time latency was measured by a clock thread and an alarm thread, working together to calibrate the time. The clock thread was set to display periodically the time using the **wait(time)** method and the alarm thread displays a message after a set time has elapsed. The alarm thread was used to verify the clock thread for each test.

6.3.1 MAS-IIVM vs. DIVM

A series of tests was performed to compare the MAS-IIVM (Multi-Agent IIVM) process with the DIVM process in terms of the data warehouse down time required. The MAS-IIVM process takes place continuously and does not require a designated down time to perform the updates. For the MAS-IIVM system used in these tests, only the warehouse view being updated is unavailable during its view modification and the warehouse is mostly available. We consolidated the unavailable time segments for all warehouse views in MAS-IIVM to obtain a data warehouse down time for comparison. DIVM

requires all the data warehouse views to be unavailable when performing view updates in a batch.

The DIVM process can be summarized in the following steps:

1. Copies of all the pre-update source tables are available at the warehouse before the view maintenance.
2. At each source during operation, transactions are logged with time stamps.
3. When the view maintenance begins, the warehouse merges the transaction logs from all the sources using time stamps.
4. Queries are performed internally at the warehouse to assemble new views.
5. The same changes are applied to the pre-update source tables.
6. The warehouse performs updates to the warehouse views in the order of the merged transaction log.

Figure 6-3 plots the increase in unavailability of the data warehouse with respect to the number of updates. The unavailability is measured by the time a view is unavailable due to view maintenance. The plot shows that MAS-IIVM provides higher view availability regardless of the number of updates processed and the gain in availability increases with increasing update frequency.

Figure 6-4 plots the increase in unavailability of the data warehouse with respect to the view complexity. View complexity is measured by the number of *join* relations defined by the SPJ expression of the view. During the DIVM process, the sources are made unavailable for the data warehouse to perform view update by querying the log tables and snapshots. A higher view complexity requires a greater number of queries to the log tables to reassemble the view. During the MAS-IIVM process, queries are done in a separate process during which time the views are still available. Therefore, the view down time is insensitive to the view complexity in MAS-IIVM.

Figure 6-5 shows the increase in unavailability of the data warehouse with respect to the data warehouse size (number of views). For DIVM, the unavailability is directly proportional to the number of views in the data warehouse, regardless of whether the views are affected by any update. This is because the whole data warehouse is unavailable during the update process. For MAS-IIVM, the unavailability is not sensitive to the number of views because only the views affected by the update are made briefly unavailable during the view tuple modification. All other views in the data warehouse are completely available.

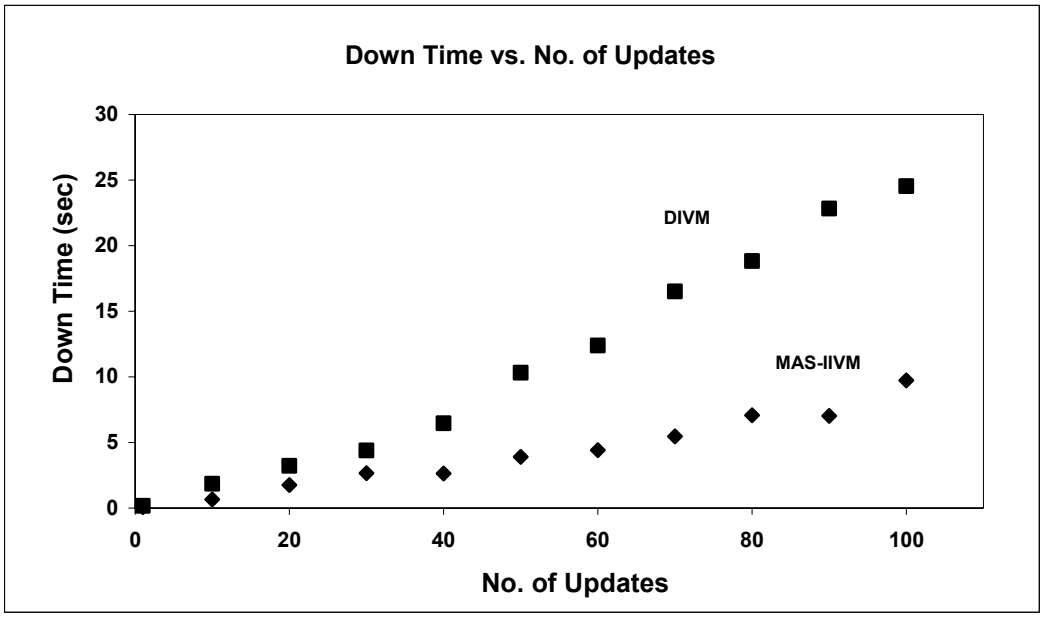

Figure 6-3 Plot of view unavailability with respect to number of updates

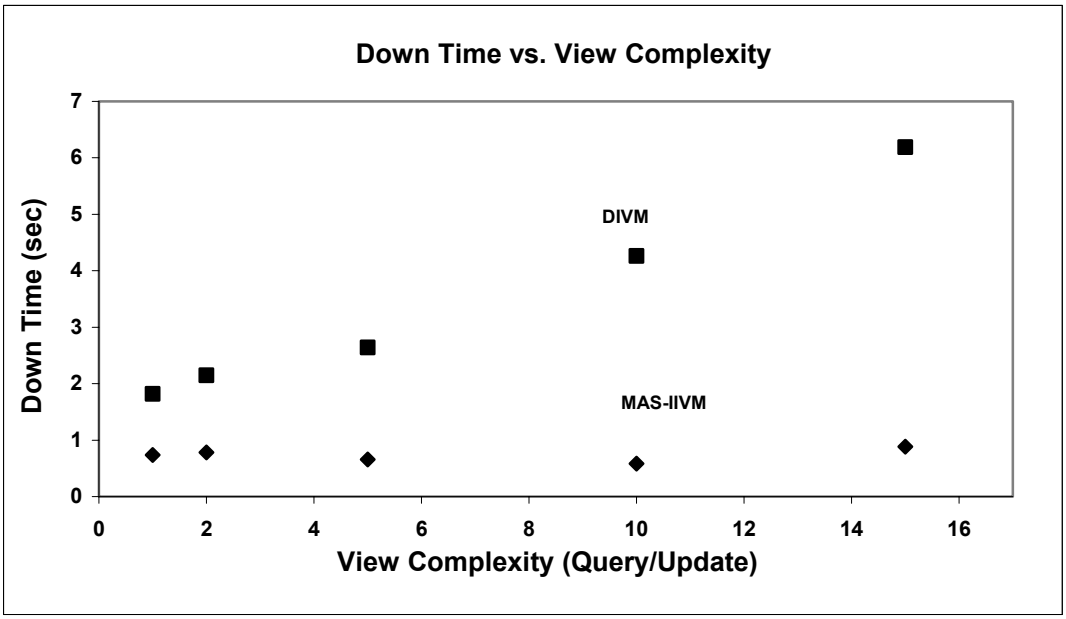

Figure 6-4 Plot of view unavailability with respect to view complexity

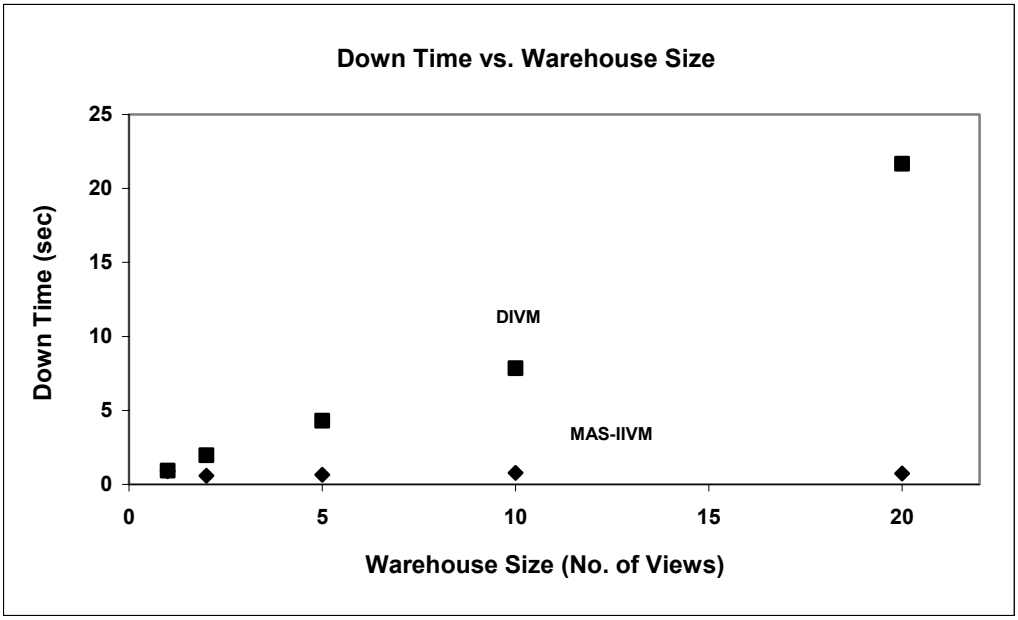

Figure 6-5 Plot of view unavailability with respect to warehouse size

6.3.2 MAS vs. C-Strobe vs. SWEEP

In this section, we shall compare the performance of the multi-agent system with the C-Strobe and the SWEEP systems. The algorithms as presented in publications were examined in detail and were emulated in the same experimental setup. Figures 6-6 to 6-10 show the effects of different important parameters on the time latency for these systems.

C-Strobe assumes that key attribute exists in each view to eliminate the need for compensation queries, except for the case where a deletion interferes with an insertion (insert-delete case). In order to provide a fair comparison, the insert-delete case is selected for all experiments so that the key assumption does not provide an advantage for the C-Strobe system.

Data Warehouse Size

The effect of the data warehouse size on the time latency is presented in Figure 6-6. The warehouse size is measured in terms of the number of unaffected views, which is the number of views in the data warehouse that are not affected by the update. C-Strobe and SWEEP systems execute a sequential scan that includes all the views in the system. For MAS, the view agents check the update in separate processes and in parallel. This provides better performance as the size of the data warehouse increases.

Update Traffic

The effect of update traffic on the time latency is presented in Figure 6-7. The update traffic is the total number of updates that are submitted to the system. The C-Strobe and the SWEEP systems check for concurrent update in the system queue, which contains the total update traffic. For MAS, all updates are distributed to the queue of each view agent as soon as they enter the data warehouse. Therefore, each view agent only needs to check a small subset of the system queue for concurrent update.

Concurrent Update

The effect of concurrent updates on the time latency is presented in Figure 6-8. Concurrent updates are updates that change the transaction state of a source before the current update executes a sub-query on it. The C-Strobe system needs to perform compensation queries for each interfering update detected. The SWEEP and the MAS systems perform local compensation, eliminating the need for compensation queries. The MAS also has a slight advantage of a smaller update queue to check for concurrent update.

Multiple View Update

The effect of multiple view updates on the time latency is presented in Figure 6-9. The highest time latency among the views updated is measured while varying the number of affected views. Both the C-Strobe and the SWEEP systems perform each update of view sequentially using a system queue, even if an update affects multiple views. The MAS is able to perform the updates of multiple views in parallel since each affected view agent accepts task from the blackboard concurrently through multicasting.

Number of Extract Agents

The effect of the number of available extract agents on the total query time is presented in Figure 6-10. Notice that the query processes of C-Strobe and SWEEP systems are equivalent to the multi-agent system using one extract agent. This part repeats the previous test with different number of extract agents. If a single extract agent is used, the total query time is the same as other systems. As the number of extract agents increases in the multi-agent system, the total query time decreases as more tasks can be performed in parallel.

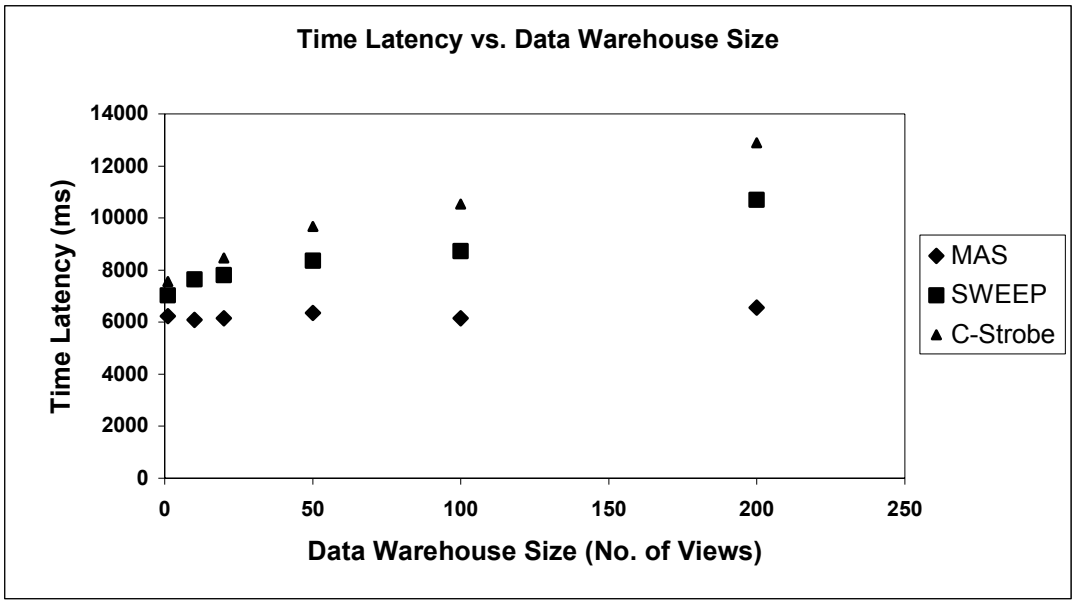

Figure 6-6 Plot of view time latency with respect to data warehouse size

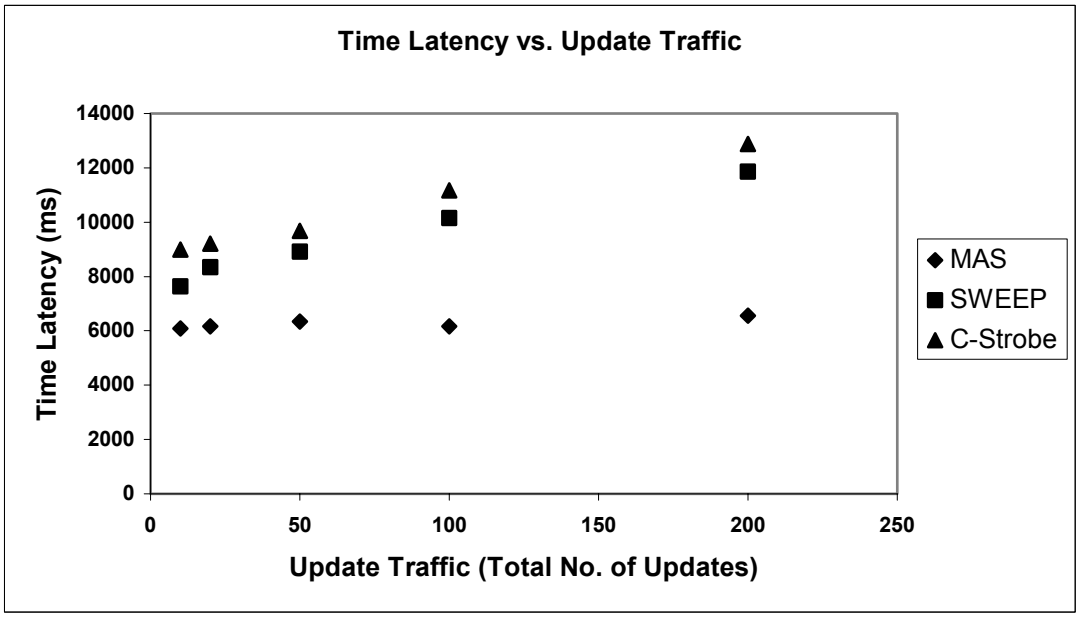

Figure 6-7 Plot of view time latency with respect to update traffic

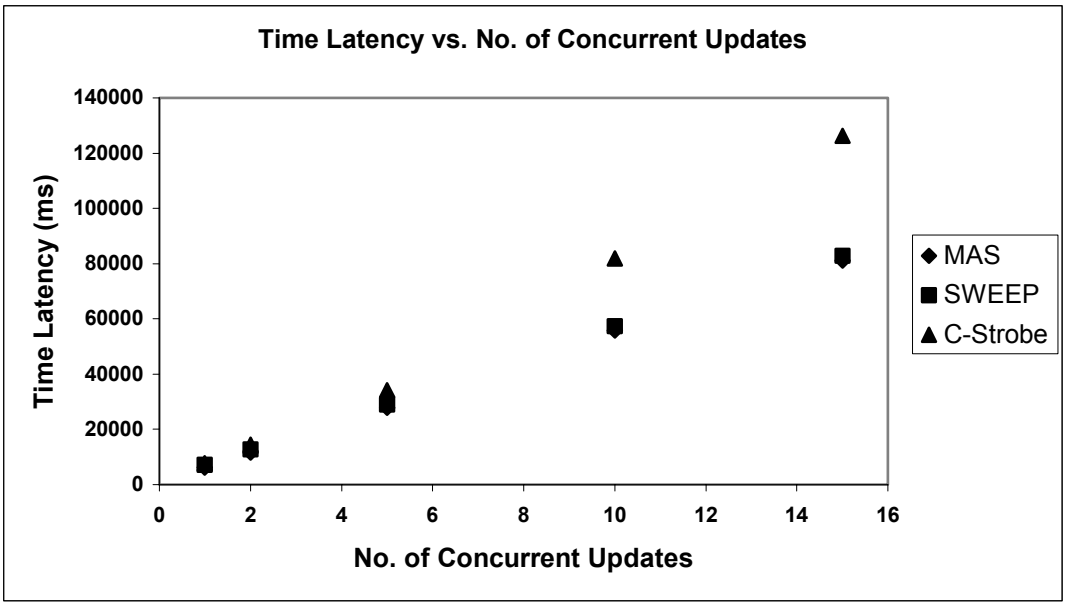

Figure 6-8 Plot of view time latency with respect to number of concurrent updates

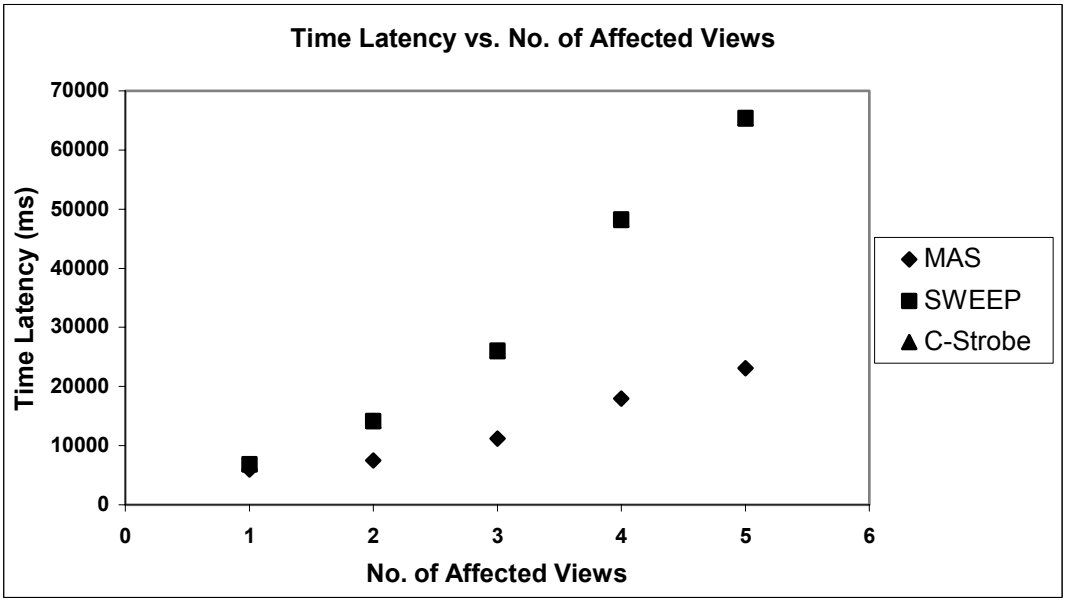

Figure 6-9 Plot of view time latency with respect to number of affected views

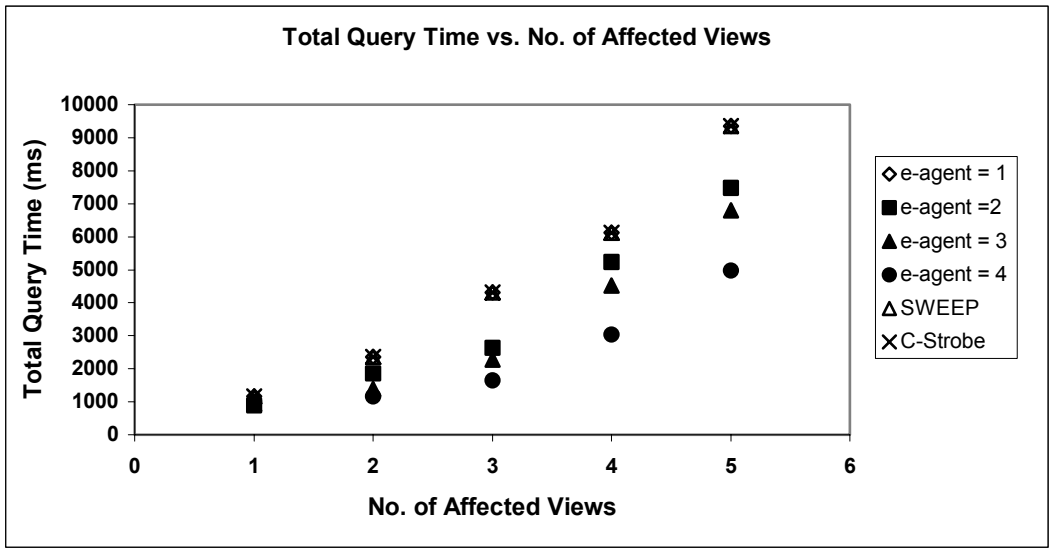

Figure 6-10 Plot of total query time with different number of extract agents

6.3.3 Processing Time for MAS

The breakdown of process time for the multi-agent system is presented in Figure 6-11. The chart shows that the blackboard accounted for an insignificant amount of process time since blackboard serves only as a medium of communication and it provides no control of the system. The three types of agents roughly shared the total process time, which implied that the IIVM steps were shared equally among the agents and no bottleneck existed.

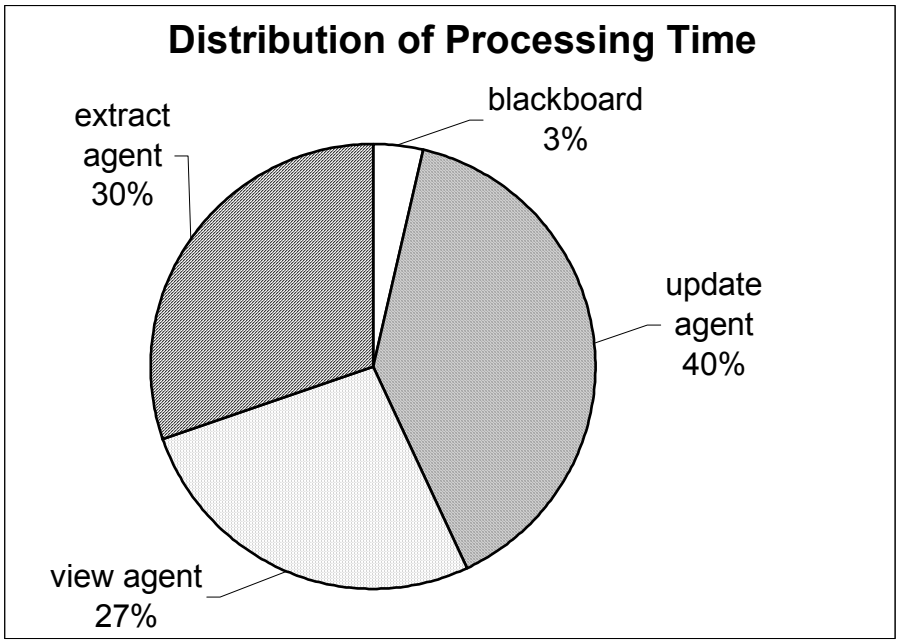

Figure 6-11 Plot of distribution of the MAS processing time

6.4 Summary

This chapter presents the quantitative results from the experimental studies. The MAS-IIVM system was demonstrated to provide higher view availability for the data warehouse than the DIVM system in all operating scenarios investigated. The importance of view availability was discussed in Chapter 2. The comparison of the multi-agent IIVM system with two existing IIVM systems (C-Strobe and SWEEP) showed that the multi-agent IIVM system gives higher throughput because of parallel processing. The higher throughput translates to higher currency of views in the data warehouse.

7 Conclusions

The major contribution of this thesis project is a multi-agent system for immediate incremental view maintenance in data warehousing. The main result is an approach for pipelining the processes involved in view maintenance. A multi-agent approach has been verified to be effective in providing immediate incremental view maintenance and continuous access for data warehouses.

The prototype is not a simulation but an operational software package that can be used to perform view maintenance on real databases through a network. However, some components have to be improved before the methodology can be an effective tool. The prototype system is able to maintain views whose data are derived from any ODBC or JDBC sources, which includes most commercially available databases. The Java fuzzy number package and the fuzzy inference system were developed particularly for the fuzzy blackboard. These packages also can be re-used in other Java programs.

A series of tests were performed to verify the performance of the multi-agent system. First, the multi-agent IIVM system was tested against its DIVM counterpart. Test results showed that the multi-agent IIVM system provides greater improvement in warehouse availability as the following parameters increase: (i) Number of Updates, (ii) Query/Update Ratio, and (iii) Size of Data Warehouse. In particular, the multi-agent IIVM system has no dependency on the latter two factors since the data warehouse is online when queries are performed.

Secondly, the multi-agent IIVM system was tested against the existing IIVM algorithms, namely SWEEP and C-Strobe. The existing algorithms were studied and emulated in the same system setup. Test results showed that the multi-agent IIVM system incurs less

time latency for the warehouse views (i.e., shorter processing time), as the following parameters increase: (i) Size of Data Warehouse, (ii) Update Traffic, (iii) Number of Concurrent Updates, and (iv) Number of Affected Views.

The results in the experimental studies are based on a small source data table size of approximately 100 records. It is expected that using a larger database size in the experiments would provide more accurate measurement for the system performance. The gain in performance by the multi-agent IIVM system should be magnified because larger database size results in longer queries, which could take even more advantage from parallel processing.

As far as data consistency is concerned, it was verified by a formal proof that the multi-agent IIVM system satisfies the highest consistency level - completeness. The multi-agent system satisfies the consistency requirement and it is able to provide good performance. Other advantages of a multi-agent system include improved scalability and maintainability.

Based on the results and experience of this thesis project, we recommend the following future research:

- Investigating the effect of the source database size on the time latency of the IIVM systems. A larger database size in the scale of thousands of records should be used to evaluate the performance.

- Improving the system to include more complex types of view. Even though SPJ expressions are useful and common for data warehouses, aggregations such as averages are also important for data warehouses. Other common warehouse storage structures should be considered, such as data cubes. A possible

implementation in the multi-agent architecture is to incorporate new agents to perform aggregations.

- Incorporating a connection pooling system to share data source connections. In the current system, a new connection is made for every update and query. The connection time is significant as seen in Appendix B - Experimental Data.

- Investigating the effect of message transmission disorder in the system. In this project, the network was assumed to be perfectly FIFO (First In First Out). Update notifications sent from the sources were assumed to arrive at the data warehouse at the same time order. This assumption may not hold due to uncertainties in the network. More work, such as that described in reference [28], is needed.

Appendix A

In this appendix, the possible effects of data inconsistency in IIVM are described through examples.

Case I: Insertion

Case I (a): An insertion interferes

The effect of an interfering insertion is extra tuple(s) in the final view. An example is illustrated in Table A.1. At time t_2, the query answer of the first **insert** gives two tuples due to the second **insert**. At time t_3, the query answer of the second **insert** adds another tuple and the final view is incorrect because of the extra tuple.

Time	Action	$R_1(W, X)$	$R_2(X, Y)$	$V_1(W, X, Y)$
t_0		(1, 2)		
t_1	**insert**(R_2, (2, 3))	(1, 2)	(2, 3)	
t_2	**insert**(R_1, (1, 2))	(1, 2), (1, 2)	(2, 3)	(1, 2, 3), (1, 2, 3)
t_3		(1, 2), (1, 2)	(2, 3)	(1, 2, 3), (1, 2, 3), (1, 2, 3)

Table A.1 An insertion is interfered by an insertion

Case I (b): A deletion interferes

The effect of an interfering deletion is missing state(s). An example is illustrated in Table A.2. At time t_2, the query answer of the first **insert** does not add a tuple to the view due to the second **delete**. At time t_3, the query answer of the second **delete** removes the only tuple in the view and the final view is correct. However, the tuple (1, 2, 3) was never inserted and this source state is missed.

Time	Action	$R_1(W, X)$	$R_2(X, Y)$	$V_1(W, X, Y)$
t_0		(1, 2)	(2, 4)	(1, 2, 4)

t_1	**insert**(R_2, (2, 3))	(1, 2)	(2, 3), (2, 4)	(1, 2, 4)
t_2	**delete**(R_1, (1, 2))		(2, 3), (2, 4)	(1, 2, 4)
t_3			(2, 3), (2, 4)	

Table A.2 An insertion is interfered by a deletion

Case I (c): An update interferes

The effect of an interfering **update** is extra tuple(s) in the final view and a missing state. An example is illustrated in Table A.3. At time t_2, the query answer of the first **insert** adds the modified tuple due to the second **update**. At time t_3, the query answer of the second **update** adds another tuple to the view. As a result, the final view is incorrect because the extra tuple and a state involving the tuple (2, 2, 3) is missing.

Time	Action	R_1(W, X)	R_2(X, Y)	V_1(W, X, Y)
t_0		(1, 2), (2, 2)		
t_1	**insert**(R_2, (2, 3))	(1, 2), (2, 2)		
t_2	**update**(R_1, (2, 2), (3, 2))	(1, 2), (3, 2)	(2, 3)	(1, 2, 3), (3, 2, 3)
t_3		(1, 2), (3, 2)		(1, 2, 3), (3, 2, 3), (3, 2, 3)

Table A.3 An insertion interfered by an update

Case II: Deletion

Case II (a): An insertion interferes

There is no erroneous effect for an interfering insertion on a deletion. An example is illustrated in Table 2.9. At time t_2, the first **delete** successfully deletes the only tuple in the view. However, the system would also attempt to delete a non-existing tuple (2, 2, 3) from the view. The final view is correct and there is no missing state.

Time	Action	R_1(W, X)	R_2(X, Y)	V_1(W, X, Y)
t_0		(1, 2)	(2, 3)	(1, 2, 3)
t_1	**delete**(R_2, (2, 3))	(1, 2)		(1, 2, 3)
t_2	**insert**(R_1, (2, 2))	(1, 2), (2, 2)		
t_3		(1, 2), (2, 2)		

Table A.4 A deletion interfered by an insertion

Case II (b): A deletion interferes

The effect of an interfering deletion is extra tuple(s) in the final view. An example is illustrated in Table A.5. At time t_1, the query answer of the first **delete** removes only one tuple from the view due to the second **delete**. At time t_3, the query answer of the second **delete** does not have any effect and the final view is incorrect because of the extra tuple.

Time	Action	R_1(W, X)	R_2(X, Y)	V_1(W, X, Y)
t_0		(1, 2), (1, 2)	(2, 3)	(1, 2, 3), (1, 2, 3)
t_1	**delete**(R_2, (2, 3))	(1, 2)		(1, 2, 3), (1, 2, 3)
t_2	**delete**(R_1, (1, 2))	(1, 2), (1, 2)		(1, 2, 3)
t_3		(1, 2)		(1, 2, 3)

Table A.5 A deletion interfered by a deletion

Case II (c): An update interferes

The effects of an interfering **update** are extra tuple(s) in the final view. Again, it has the combined effects of both an interfering deletion and interfering insertion. An example is illustrated in Table A.6. At time t_2, the first **delete** fails to remove the tuple (2, 2, 3) since it has been modified by the second **update**. As a result, the final view is incorrect because the extra tuple and a state involving the tuple (2, 2, 3) is missing.

Time	Action	R_1(W, X)	R_2(X, Y)	V_1(W, X, Y)
t_0		(1, 2), (2, 2)	(2, 3)	(1, 2, 3), (2, 2, 3)
t_1	**delete**(R_2, (2, 3))	(1, 2), (2, 2)		(1, 2, 3), (2, 2, 3)
t_2	**update**(R_1, (2, 2), (3, 2))	(1, 2), (3, 2)		(2, 2, 3)
t_3		(1, 2), (3, 2)		(2, 2, 3)

Table A.6 A deletion interfered by an update

Case III: Update

Case III (a): An insertion interferes

The effect of an interfering **insert** is extra tuple(s) in the final view. It has the combined effects of both a deletion and an insertion interfered by an insertion. An example is illustrated in Table A.7. At time t_2, the **update** includes the inserted tuple by the **insert**. At time t_3, the query answer of the **insert** adds another tuple to the view. As a result, the final view is incorrect because the extra tuple.

Time	Action	R_1(W, X)	R_2(X, Y)	V_1(W, X, Y)
t_0		(1, 2)	(2, 3)	(1, 2, 3)
t_1	**update**(R_2, (2, 3), (2, 4))	(1, 2)	(2, 3)	(1, 2, 3)
t_2	**insert**(R_1, (2, 2))	(1, 2), (2, 2)	(2, 4)	(1, 2, 4), (2, 2, 4)
t_3		(1, 2), (2, 2)	(2, 4)	(1, 2, 4), (2, 2, 4), (2, 2, 4)

Table A.7 An update interfered by an insertion

Case III (b): A deletion interferes

The effect of an interfering **delete** is extra tuple(s) in the final view and a missing state. It has the combined effects of both a deletion and an insertion interfered by a deletion. An example is illustrated in Table A.8. At time t_2, the **update** fails to update the tuple (2, 2, 3) due to the **delete**. At time t_3, the system attempts to delete a tuple (2, 2, 4) but fails to find it in the warehouse. As a result, the final view is incorrect because the extra tuple. Also, a source state involving the tuple (2, 2, 4) is missing.

Time	Action	R_1(W, X)	R_2(X, Y)	V_1(W, X, Y)
t_0		(1, 2), (2, 2)	(2, 3)	(1, 2, 3), (2, 2, 3)
t_1	**update**(R_2, (2, 3), (2, 4))	(1, 2), (2, 2)	(2, 4)	(1, 2, 3), (2, 2, 3)
t_2	**delete**(R_1, (2, 2))	(1, 2)	(2, 4)	(1, 2, 4), (2, 2, 3)
t_3		(1, 2)	(2, 4)	(1, 2, 4), (2, 2, 3)

Table A.8 An update interfered by a deletion

Case III (c): An update interferes

The effect of an interfering **update** is extra tuple(s) in the final view and a missing state. It has the combined effects of both a deletion interfered by an **update** and an insertion interfered by an **update**. An example is illustrated in Table A.9. At time t_2, the **update**

fails to update the tuple (2, 2, 3) due to the second **update**. At time t_3, the second **update** adds another tuple (3, 2, 4) to the view. As a result, the final view is incorrect because the two extra tuples. Also, a source state involving the tuple (2, 2, 4) is missing.

Time	Action	R_1(W, X)	R_2(X, Y)	V_1(W, X, Y)
t_0		(1, 2), (2, 2)	(2, 3)	(1, 2, 3), (2, 2, 3)
t_1	**update**(R_2, (2, 3), (2, 4))	(1, 2), (2, 2)	(2, 4)	(1, 2, 3), (2, 2, 3)
t_2	**update**(R_1, (2, 2), (3, 2))	(1, 2), (3, 2)	(2, 4)	(1, 2, 4), (2, 2, 3), (3, 2, 4)
t_3		(1, 2), (3, 2)	(2, 4)	(1, 2, 4), (2, 2, 3), (3, 2, 4), (3, 2, 4)

Table A.9 An update interfered by an update

Appendix B

Test 1: Warehouse Down Time vs. Number of Updates

Experiment Parameters	
# of Updates / View	Variable
# of Views modified / Update	2
# of Queries / Update	2
update operation is insertion	

Table B.1 Parameters for Test 1

DIVM	(All data represent median values; All units in seconds)				
Number of Updates	Total time for source 1 sub-queries	Total time for source 2 sub-queries	Total time for updating view 1	Total time for updating view 2	Total warehouse down time
1	0.015	0.015	0.074	0.051	0.155
10	0.294	0.147	0.780	0.586	1.807
20	0.312	0.624	1.318	1.318	3.571
30	0.898	0.449	1.757	2.212	5.317
40	1.171	0.587	3.535	3.120	8.413
50	0.732	0.737	2.570	4.418	8.458
60	1.770	0.884	3.953	3.953	10.560
70	2.062	1.034	5.162	3.598	11.856
80	1.182	1.176	6.240	4.685	13.284
90	2.635	1.322	5.930	5.930	15.816
100	1.560	1.497	5.857	7.375	16.288

Table B.2 Test 1 data for DIVM

IIVM	(All data represent median values; All units in seconds)		
Number of Updates	Total time for updating first view	Total time for updating second view	Total warehouse down time
1	0.051	0.074	0.063
10	0.586	0.659	0.622
20	1.171	1.171	1.171
30	1.977	1.542	1.759
40	2.635	2.635	2.635
50	3.294	4.418	3.856
60	4.425	4.680	4.552
70	5.162	5.162	5.162

80	6.240	5.271	5.756
90	7.020	5.271	6.146
100	8.837	5.140	6.988

Table B.3 Test 1 data for IIVM

Test 2: Warehouse Down Time vs. Query / Update Ratio (View Complexity)

Experiment Parameters:	
# of Updates / View	10
# of Views Modified / Update	2
# of Sub-Queries / Update	Variable
update operation is insertion	

Table B.4 Parameters for Test 2

DIVM	(All data represent median values; All units in seconds)			
Query/Update Ratio	Total time for performing sub-queries	Total time for updating View 1	Total time for updating View 2	Total warehouse down time
1	0.295	0.659	0.737	1.691
2	0.589	0.737	0.514	1.841
5	1.471	0.780	0.586	2.836
10	2.928	0.586	0.659	4.173
15	4.680	0.884	0.737	6.301

Table B.5 Test 2 data for DIVM

IIVM	(All data represent median values; All units in seconds)		
Query/Update Ratio	Total time for updating View 1	Total time for updating View 2	Total warehouse down time
1	0.659	0.737	0.698
2	0.659	0.659	0.659
5	0.737	0.586	0.662
10	0.737	0.514	0.626
15	0.780	0.659	0.719

Table B.6 Test 2 data for IIVM

Test 3: Warehouse Down Time vs. Number of Updated Views

Experiment Parameters	
# of Updates / View	10
# of Views Modified / Update	Variable
# of Sub-Queries / Update	2
update operation is insertion	

Table B.7 Parameters for Test 3

DIVM	(All data represent median values; All units in seconds)			
Number of Updated Views	Total time for Source 1 sub-queries	Total time for Source 2 sub-queries	Total time for updating views	Total warehouse down time
1	0.078	0.075	0.780	0.933
2	0.299	0.147	1.318	1.764
5	0.732	0.369	2.928	4.029
10	0.732	0.736	7.375	8.843
20	2.950	1.477	10.279	14.706

Table B.8 Test 3 data for DIVM

IIVM	(All data represent median values; All units in seconds)	
Number of Updated Views	Total time for updating views	Total warehouse down time
1	0.780	0.780
2	0.659	0.659
5	0.586	0.586
10	0.737	0.737
20	0.514	0.514

Table B.9 Test 3 data for IIVM

Test 4: Distribution of Processing Time

Experiment Parameters	
# of Updates / View	1
# of Views Modified / Update	1
# of Sub-Queries / Update	2
update operation is insertion	

Table B.10 Parameters for Test 4

IIVM	(All data represent median values; All units in seconds)		
Module	Processes	Processing time	Total processing time
Blackboard	Processing	0.018	0.018
Update Agent	Source Connection	0.186	
	Source Update Query	0.022	0.208
Extract Agent	Extract Connection	0.150	
	Extract Query	0.010	0.160
View Agent	Warehouse Connection	0.070	
	Warehouse Update	0.070	0.140

Table B.11 Test 4 data for IIVM-MAS

Test 5 (MAS): Warehouse View Time Latency vs. No. of Unaffected Views (Warehouse Size)

Experiment Parameters					
# of Updates / View	2	# of Unrelated Updates Pending	10	# of Update Agents	1
# of Views Modified / Update	1	# of Interfering Updates	1	# of Extract Agents	1
# of Sub-Queries / Update	2	# of Views in System	Variable	# of View Agents	Variable
- update operations are insertion, interfered by a deletion					

Table B.12 Parameters for Test 5 - MAS

PART 1

IIVM - MAS	(All data represent median values; All units in seconds)					
No. of Unaffected Views	Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Consolidated time for Insertion
1	0.009	0.156	0.009	0.753	0.008	0.250
10	0.011	0.159	0.009	0.261	0.008	0.297
20	0.009	0.109	0.008	0.438	0.009	0.308
50	0.009	0.127	0.008	0.762	0.009	0.407
100	0.008	0.148	0.009	0.291	0.011	0.273
200	0.008	0.136	0.011	0.769	0.009	0.285

PART 2

IIVM - MAS	(All data represent median values; All units in seconds)					
Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Consolidated time for Deletion	Total latency time
0.008	0.145	0.000	0.400	0.000	5.855	7.593
0.008	0.127	0.000	0.593	0.000	4.915	6.388
0.009	0.074	0.000	0.444	0.000	4.926	6.334
0.009	0.137	0.000	0.377	0.000	4.486	6.331
0.011	0.136	0.000	0.299	0.000	3.763	4.948
0.009	0.308	0.000	0.344	0.000	4.769	6.647

Table B.13 Test 5 data for IIVM - MAS

Test 5 (SWEEP): Warehouse View Time Latency vs. No. of Unaffected Views (Warehouse Size)

Experiment Parameters			
# of Updates / View	2	# of Unrelated Updates Pending	10
# of Views Modified / Update	1	# of Interfering Updates	1
# of Sub-Queries / Update	2	# of Views in System	Variable
update operation is insertion, interfered by a deletion			

Table B.14 Parameters for Test 5 - SWEEP

PART 1

IIVM - SWEEP	(All data represent median values; All units in seconds)					
No. of Unaffected Views	Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Time for Insertion
1	0.009	0.373	0.078	0.261	0.091	0.109
10	0.093	0.156	0.079	0.438	0.106	0.297
20	0.156	0.159	0.091	0.762	0.093	0.112
50	0.397	0.109	0.093	0.291	0.086	0.308
100	0.909	0.127	0.106	0.042	0.085	0.407
200	2.119	0.148	0.086	0.148	0.077	0.273

PART 2

IIVM - SWEEP	(All data represent median values; All units in seconds)					
Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Time for Deletion	Total latency time
0.008	0.240	0.084	0.685	0.077	3.763	5.777
0.085	0.075	0.095	0.299	0.083	4.486	6.292
0.171	0.095	0.077	0.344	0.095	4.915	7.070
0.463	0.469	0.082	0.377	0.069	4.926	7.669
1.059	0.145	0.070	0.400	0.076	5.855	9.282
1.818	0.127	0.071	0.593	0.082	4.769	10.312

Table B.15 Test 5 data for IIVM - SWEEP

Test 5 (C-Strobe): Warehouse View Time Latency vs. No. of Unaffected Views (Warehouse Size)

Experiment Parameters			
# of Updates / View	2	# of Unrelated Updates Pending	10
# of Views Modified / Update	1	# of Interfering Updates	1
# of Sub-Queries / Update	2	# of Views in System	Variable
update operation is insertion, interfered by a deletion			

Table B.16 Parameters for Test 5 - C-Strobe

PART 1

IIVM - C Strobe	(All data represent median values; All units in seconds)						
Time for Insertion	Associating updates with views	Time for Source 1 sub-queries	Time for Source 2 sub-queries	Checking for interfering updates	Time for Source 1 compensate queries	Time for Source 2 compensate queries	Checking for interfering updates
1	0.009	0.156	0.291	0.106	0.095	0.400	0.079
10	0.078	0.159	0.042	0.093	0.145	0.593	0.066
20	0.159	0.109	0.148	0.086	0.127	0.444	0.078
50	0.455	0.127	0.171	0.085	0.074	0.377	0.099
100	1.059	0.148	1.051	0.077	0.137	1.051	0.087
200	1.852	0.753	0.769	0.086	0.136	0.031	0.074

PART 2

IIVM - C Strobe	(All data represent median values; All units in seconds)					
Time for Insertion	Associating updates with views	Time for Source 1 sub-queries	Time for Source 2 sub-queries	Checking for interfering updates	Time for Deletion	Total latency time
0.109	0.009	0.240	0.685	0.086	3.733	5.998
0.297	0.093	0.075	0.299	0.077	4.190	6.207
0.889	0.156	0.469	0.344	0.081	5.156	8.246
0.308	0.397	0.095	0.921	0.093	5.970	9.172
0.407	0.909	0.145	0.400	0.068	5.855	11.394
0.800	2.119	0.127	0.593	0.085	4.915	12.340

Table B.17 Test 5 data for IIVM - C-Strobe

Test 6 (MAS): Warehouse View Time latency vs. No. of Unrelated Updates in Queue (Update Traffic)

Experiment Parameters					
# of Updates / View	2	# of Unrelated Updates Pending	Variable	# of Update Agents	1
# of Views Modified / Update	1	# of Interfering Updates	1	# of Extract Agents	1
# of Sub-Queries / Update	2	# of Views in System	10	# of View Agents	10
- update operation is insertion, interfered by a deletion					

Table B.18 Parameters for Test 6 - MAS

PART 1

IIVM - MAS	(All data represent median values; All units in seconds)					
No. of Unaffected Views	Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Consolidated time for Insertion
10	0.009	0.373	0.008	0.261	0.009	0.109
20	0.011	0.156	0.009	0.438	0.011	0.297
50	0.009	0.109	0.009	0.291	0.009	0.308
100	0.008	0.127	0.009	0.042	0.008	0.407
200	0.008	0.148	0.008	0.148	0.008	0.800

PART 2

IIVM - MAS	(All data represent median values; All units in seconds)					
Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Consolidated time for Deletion	Total latency time
0.008	0.145	0.000	0.685	0.000	3.733	5.341
0.009	0.127	0.000	0.299	0.000	5.970	7.326
0.009	0.137	0.000	0.921	0.000	4.190	5.992
0.009	0.136	0.000	0.400	0.000	5.855	7.000
0.008	0.308	0.000	0.593	0.000	4.915	6.945

Table B.19 Test 6 data for IIVM - MAS

Test 6 (SWEEP): Warehouse View Time latency vs. No. of Unrelated Updates in Queue (Update Traffic)

Experiment Parameters			
# of Updates / View	2	# of Unrelated Updates Pending	Variable
# of Views Modified / Update	1	# of Interfering Updates	1
# of Sub-Queries / Update	2	# of Views in System	10
update operation is insertion, interfered by a deletion			

Table B.20 Parameters for Test 6 - SWEEP

PART 1

IIVM - SWEEP	(All data represent median values; All units in seconds)					
No. of Unaffected Views	Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Time for Insertion
10	0.091	0.373	0.086	0.291	0.078	1.031
20	0.106	0.156	0.187	0.042	0.159	0.762
50	0.093	0.159	0.391	0.148	0.455	0.655
100	0.086	0.109	0.794	0.171	1.059	1.059
200	0.085	0.127	1.818	0.364	1.852	0.762

PART 2

IIVM - SWEEP	(All data represent median values; All units in seconds)					
Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Time for Deletion	Total latency time
0.091	0.095	0.070	0.685	0.070	5.156	8.118
0.106	0.145	0.151	0.299	0.151	5.970	8.233
0.093	0.127	0.445	0.344	0.445	5.855	9.208
0.086	0.074	1.049	0.921	1.049	4.915	11.371
0.085	0.137	1.843	0.400	1.843	4.926	14.241

Table B.21 Test 6 data for IIVM - SWEEP

Test 6 (C-Strobe): Warehouse View Time latency vs. No. of Unrelated Updates in Queue (Update Traffic)

Experiment Parameters			
# of Updates / View	2	# of Unrelated Updates Pending	Variable
# of Views Modified / Update	1	# of Interfering Updates	1
# of Sub-Queries / Update	2	# of Views in System	10
update operation is insertion, interfered by a deletion			

Table B.22 Parameters for Test 6 - C-Strobe

PART 1

IIVM - C Strobe	(All data represent median values; All units in seconds)						
Time for Insertion	Associating updates with views	Time for Source 1 sub-queries	Time for Source 2 sub-queries	Checking for interfering updates	Time for Source 1 compensate queries	Time for Source 2 compensate queries	Checking for interfering updates
10	0.079	0.156	0.261	0.079	0.095	0.719	0.093
20	0.091	0.159	0.438	0.182	0.145	0.485	0.219
50	0.106	0.109	0.762	0.530	0.127	1.031	0.650
100	0.093	0.127	0.291	0.926	0.074	0.762	0.934
200	0.086	0.148	0.042	1.712	0.137	0.655	1.814

PART 2

IIVM - C Strobe	(All data represent median values; All units in seconds)					
Time for Insertion	Associating updates with views	Time for Source 1 sub-queries	Time for Source 2 sub-queries	Checking for interfering updates	Time for Deletion	Total latency time
0.109	0.106	0.127	0.299	0.084	4.190	6.398
0.297	0.093	0.074	0.344	0.148	5.156	7.830
0.250	0.086	0.137	0.921	0.389	4.926	10.023
0.308	0.085	0.136	0.400	0.900	4.915	9.950
0.407	0.077	0.308	0.593	2.108	5.855	13.941

Table B.23 Test 6 data for IIVM - C-Strobe

Test 7 (MAS) - Warehouse View Time Latency vs. No. of Interfering Updates

Experiment Parameters					
# of Updates / View	2	# of Unrelated Updates Pending	15	# of Update Agents	1
# of Views Modified / Update	1	# of Interfering Updates	Variable	# of Extract Agents	1
# of Sub-Queries / Update	2	# of Views in System	10	# of View Agents	10
- update operations are insertion, interfered by a deletion					

Table B.24 Parameters for Test 7 - MAS

PART 1

IIVM - MAS	(All data represent median values; All units in seconds)					
No. of Unaffected Views	Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Consolidated time for Insertion
1	0.009	0.156	0.008	0.261	0.009	0.109
2	0.008	0.159	0.018	0.438	0.017	0.297
5	0.008	0.109	0.053	0.762	0.042	0.889
10	0.009	0.127	0.093	0.291	0.077	0.308
15	0.011	0.148	0.128	0.042	0.128	0.407

PART 2

IIVM - MAS	(All data represent median values; All units in seconds)						
Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Consolidated time for Deletion	Subsequent Checking	Total latency time
0.008	0.240	0.000	0.299	0.000	4.926	0.000	6.025
0.016	0.149	0.008	0.688	0.008	8.973	0.019	10.796
0.045	2.344	0.031	4.603	0.036	18.814	0.156	27.892
0.106	0.952	0.077	4.000	0.095	47.692	0.714	54.542
0.139	2.182	0.030	8.898	0.130	72.634	1.909	86.786

Table B.25 Test 7 data for IIVM - MAS

Test 7 (SWEEP) - Warehouse View Time Latency vs. No. of Interfering Updates

Experiment Parameters			
# of Updates / View	2	# of Unrelated Updates Pending	10
# of Views Modified / Update	1	# of Interfering Updates	1
# of Sub-Queries / Update	2	# of Views in System	Variable
update operation is insertion, interfered by a deletion			

Table B.26 Parameters for Test 7 - SWEEP

PART 1

IIVM - SWEEP	(All data represent median values; All units in seconds)					
No. of Unaffected Views	Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Time for Insertion
1	0.079	0.159	0.009	0.291	0.009	0.400
2	0.091	0.109	0.021	0.042	0.017	0.593
5	0.106	0.127	0.046	0.148	0.042	0.444
10	0.093	0.148	0.086	0.171	0.077	0.377
15	0.086	0.136	0.127	0.364	0.128	0.678

PART 2

IIVM - SWEEP	(All data represent median values; All units in seconds)					
Associating updates with views	Time for Source 1 sub-queries	Checking for interfering updates	Time for Source 2 sub-queries	Checking for interfering updates	Time for Deletion	Total latency time
0.078	0.095	0.127	0.685	0.130	4.842	0.000
0.159	0.291	0.159	0.597	0.120	8.381	0.019
0.455	0.636	0.139	1.719	0.119	23.846	0.171
1.059	0.741	0.128	9.206	0.108	49.153	0.763
1.389	2.055	0.127	6.000	0.120	73.889	1.615

Table B.27 Test data for Test 7 - SWEEP

Test 7 (C-Strobe) - Warehouse View Time Latency vs. No. of Interfering Updates

Experiment Parameters			
# of Updates / View	2	# of Unrelated Updates Pending	15
# of Views Modified / Update	1	# of Interfering Updates	Variable
# of Sub-Queries / Update	2	# of Views in System	10
update operation is insertion, interfered by a deletion			

Table B.28 Parameters for IIVM - C-Strobe

PART 1

IIVM - C Strobe	(All data represent median values; All units in seconds)						
Time for Insertion	Associating updates with views	Time for Source 1 sub-queries	Time for Source 2 sub-queries	Checking for interfering updates	Time for Source 1 compensate queries	Time for Source 2 compensate queries	Checking for interfering updates
1	0.078	0.156	0.753	0.139	0.095	0.299	0.112
2	0.079	0.159	0.261	0.257	0.291	0.688	0.350
5	0.091	0.109	0.438	0.636	0.636	1.884	0.566
10	0.106	0.127	0.762	1.154	0.741	4.000	1.107
15	0.093	0.148	0.291	1.926	2.055	8.898	1.821

PART 2

IIVM - C Strobe	(All data represent median values; All units in seconds)					
Time for Insertion	Associating updates with views	Time for Source 1 sub-queries	Time for Source 2 sub-queries	Checking for interfering updates	Time for Deletion	Total latency time
0.719	0.093	0.145	0.400	0.000	4.190	7.181
0.485	0.156	0.254	1.186	0.120	11.709	15.996
1.031	0.397	0.370	2.222	1.186	24.576	34.142
0.762	0.909	1.370	6.849	4.846	49.259	71.992
0.655	1.589	2.034	8.898	12.586	67.295	108.288

Table B.29 Test 7 data for IIVM - C-Strobe

Test 8 (MAS / SWEEP) - Warehouse View Time Latency vs. No. of Affected Views

Experiment Parameters					
# of Updates / View	2	# of Unrelated Updates Pending	10	# of Update Agents	1
# of Views Modified / Update	Variable	# of Interfering Updates	1	# of Extract Agents	1
# of Sub-Queries / Update	2	# of Views in System	Variable	# of View Agents	Variable
- update operations are insertion, interfered by a deletion					

Table B.30 Parameters for Test 8 - MAS

IIVM - MAS	(All data represent median values; All units in seconds)				
# of Affected Views	Associating updates with views	Total Query + Interference Check	Consolidated Insertion Time	Consolidated Deletion Time	Total Time
1	0.009	1.092	0.000	4.842	5.943
2	0.021	2.180	0.000	5.257	7.458
3	0.032	3.270	0.000	7.880	11.182
4	0.034	4.368	0.000	13.563	17.965
5	0.042	5.461	0.000	17.599	23.102

Table B.31 Test 8 data for IIVM - MAS

Experiment Parameters			
# of Updates / View	2	# of Unrelated Updates Pending	10
# of Views Modified / Update	Variable	# of Interfering Updates	1
# of Sub-Queries / Update	2	# of Views in System	Variable
update operation is insertion, interfered by a deletion			

Table B.32 Parameters for Test 8 - SWEEP

IIVM - SWEEP	(All data represent median values; All units in seconds)				
# of Affected Views	Associating updates with views	Total Query + Interference Check	Total Insertion Time	Total Deletion Time	Total Time
1	0.009	1.091	0.902	4.842	6.844
2	0.016	2.203	1.896	10.040	14.154

3	0.024	3.329	3.092	19.614	26.058
4	0.036	4.469	2.691	41.062	48.258
5	0.053	5.621	2.723	57.000	65.397

Table B.33 Test 8 data for IIVM - SWEEP

Test 8 (C-Strobe) - Warehouse View Time Latency vs. Number of Affected Views

Experiment Parameters			
# of Updates / View	2	# of Unrelated Updates Pending	10
# of Views Modified / Update	Variable	# of Interfering Updates	1
# of Sub-Queries / Update	2	# of Views in System	Variable
update operation is insertion, interfered by a deletion			

Table B.34 Parameters for Test 8 - C-Strobe

IIVM - CStrobe	(All data represent median values; All units in seconds)				
# of Affected Views	Total view-update checking time	Total sub-queries + interference checking + compensation queries	Total insertion time	Total deletion time	Total time
1	0.009	1.074	0.902	4.842	6.827
2	0.017	2.167	1.896	10.040	14.119
3	0.025	3.276	3.092	19.614	26.007
4	0.031	4.406	2.691	41.062	48.190
5	0.043	5.546	2.723	57.000	65.312

Table B.35 Test 8 data for IIVM - C-Strobe

Test 9 (MAS / SWEEP / C-Strobe) - Total Query Time vs. No. of Affected Views (For Varying No. of Extract Agents)

Experiment Parameters					
# of Updates / View	2	# of Unrelated Updates Pending	10	# of Update Agents	1
# of Views Modified / Update	Variable	# of Interfering Updates	1	# of Extract Agents	Variable
# of Sub-Queries / Update	2	# of Views in System	Variable	# of View Agents	Variable
- update operations are insertion, interfered by a deletion					

Table B.36 Parameters for Test 9 - IIVM

# of Extract Agents = 4											
# of Affected Views	A	B	C	D	E	F	G	H	I	J	Total Net Query Time
1	0.874					0.109					0.984
2	0.874	0.109				0.109	0.273				1.366
3	0.874	0.109	0.164			0.109	0.273	0.710			2.240
4	0.874	0.109	0.164	0.437		0.109	0.273	0.710	0.328		3.005
5	0.874	0.109	0.164	0.437	0.055	0.109	0.273	0.710	0.328	1.858	4.918

Table B.37 Test 9 data for IIVM (No. Extract Agents = 4)

# of Extract Agents = 3											
# of Affected Views	A	B	C	D	E	F	G	H	I	J	Total Net Query Time
1	0.162					0.757					0.919
2	0.162	0.054				0.757	0.162				1.135
3	0.162	0.054	0.216			0.757	0.162	0.270			1.622
4	0.162	0.054	0.216	0.216		0.757	0.162	0.270	2.649		4.486
5	0.162	0.054	0.216	0.216	0.541	0.757	0.162	0.270	2.649	1.730	6.757

Table B.38 Test 9 data for IIVM (No. Extract Agents = 3)

Test 9 (MAS / SWEEP / C-Strobe) - Total Query Time vs. No. of Affected Views (For Varying No. of Extract Agents)

# of Extract Agents = 2											
# of Affected Views	A	B	C	D	E	F	G	H	I	J	Total Net Query Time
1	0.055					0.811					0.865
2	0.055	0.492				0.811	0.973				2.330
3	0.055	0.492	0.273			0.811	0.973	1.676			4.279
4	0.055	0.492	0.273	0.109		0.811	0.973	1.676	0.811		5.199
5	0.055	0.492	0.273	0.109	1.093	0.811	0.973	1.676	0.811	1.135	7.427

Table B.39 Test 9 data for IIVM (No. Extract Agents = 2)

# of Extract Agents = 1											
# of Affected Views	A	B	C	D	E	F	G	H	I	J	Total Net Query Time
1	0.380					0.759					1.139
2	0.380	0.380				0.759	0.316				1.835
3	0.380	0.380	0.380			0.759	0.316	0.380			2.595
4	0.380	0.380	0.380	1.519		0.759	0.316	0.380	1.962		6.076
5	0.380	0.380	0.380	1.519	0.633	0.759	0.316	0.380	1.962	2.595	9.304

Table B.40 Test 9 data for IIVM (No. Extract Agents = 1)

Notations:
A: Net Query Time for View 1 (Insertion)
B: Net Query Time for View 2 (Insertion)
C: Net Query Time for View 3 (Insertion)
D: Net Query Time for View 4 (Insertion)
E: Net Query Time for View 5 (Insertion)
F: Net Query Time for View 1 (Deletion)
G: Net Query Time for View 2 (Deletion)
H: Net Query Time for View 3 (Deletion)
I: Net Query Time for View 4 (Deletion)
J: Net Query Time for View 5 (Deletion)

Bibliography

[1] A. Gupta, I.S. Mumick, V.S. Subrahmanian, "Maintaining Views Incrementally," in *Proc. of the ACM SIGMOD Conference*, Washington DC, USA, May 1993, pp. 157-166.

[2] A. Haddadi, "Communication and Cooperation in Agent Systems," *Lecture Notes in Artificial Intelligence 1056*, Springer, London, UK, 1995.

[3] A. Kawaguchi, D. Lieuwen, I. Mumick, D. Quass, K. Ross, "Concurrency Control Theory for Deferred Materialized Views," in *Proc. of the International Conference on Database Theory*, Athens, Greece, January 1997.

[4] B. Lewis, D.J. Berg, *Multithreaded Programming with Java Technology*, Prentice Hall, Upper Saddle River, NJ, USA, 2000.

[5] C. J. White, *Sybase Adaptive Server IQ - A High-Performance Database for Decision Processing*, January 1999. http://www.sybase.com/detail_list/1,3691,2291,00.html (February 11, 2001)

[6] C. Marshall, *Enterprise Modeling With UML: Designing Successful Software Through Business Analysis*, Addison-Wesley, Reading, MA, USA, 2000.

[7] D. Agrawal, A. El Abbadi, et al., "Efficient View Maintenance at Data Warehouses," in *Proc. of the ACM SIGMOD Conference*, Tucson, AZ, USA, May 1997, pp. 417-427.

[8] E. Hanson, "A Performance Analysis of View Materialization Strategies," in *Proc. of ACM SIGMOD International Conference on Management of Data*, San Francisco, CA, USA, May 1987, pp. 440-453.

[9] F. von Martial, "Coordinating Plans of Autonomous Agents," *Lecture Notes in Artificial Intelligence 610*, Springer-Verlag, London, UK, 1992.

[10] H. Garcia-Molina, W. J. Labio, J. Yang, "Expiring Data in a Warehouse," in *Proc. of the 24th VLDB Conference*, New York, NY, USA, August 1998.

[11] H. Garcia-Molina, W. J. Labio, J. L. Wiener, Y. Zhuge, "Distributed and Parallel Computing Issues in Data Warehousing," in *Proc. of ACM Principles of Distributed Computing Conference*, 1999.

[12] H. Gupta, "Selection of Views to Materialize in a Data Warehouse," in *Proc. of the International Conference on Database Theory*, Athens, Greece, January 1997.

[13] IBM Corp., *IBM DB2 Warehouse Manager for AS/400 - Foundation for Business Intelligence Applications*. http://www.ibmlink.ibm.com/usalets&parms=H_200-176 (February 11, 2001)

[14] IBM Corp., *Consultant Report: AS/400 Data Warehouse Solutions for the Business User*. http://www.as400.ibm.com/CONSLT/DATABASE.HTM (February 11, 2001)

[15] I. Mumick, D. Quass, B. Mumick, "Maintenance of Data Cubes and Summary Tables in a Warehouse," in *Proc. of the ACM SIGMOD Conference*, Tuscon, AZ, USA, May 1997.

[16] J. J. Patrick, *SQL Fundamentals*, Professional Technical Reference, Prentice Hall, Upper Saddle River, NJ, USA, 1999.

[17] J. Widom, "Research Problems in Data Warehousing," in *Proc. of the 4th International Conference on Information and Knowledge Management (CIKM)*, MD, USA, November 1995.

[18] L. S. Colby, T. Griffin, L. Libkin et al., "Algorithms for Deferred View Maintenance," in *Proc. of the ACM SIGMOD Conference*, Montreal, QB, Canada, June 1996, pp. 469-480.

[19] Microsoft Corp., *Microsoft SQL Server 2000 Operation Manual*, 2000.

[20] Oracle Corp., *New Features Overview of the New OWB - Technical White Paper.* http://otn.oracle.com/products/warehouse/pdf/owb3ifo_twp.pdf (February 11, 2001)

[21] P. Krneta, Sybase Inc., *Sybase Adaptive Server IQ with Multiplex - A New Data Warehousing Paradigm for User and Data Scalability*, June 2000. http://www.sybase.com/detail_list/1,3691,2291,00.html (February 11, 2001)

[22] R. Chen, W. Meng, "Precise Detection and Proper Handling of View Maintenance Anomalies in a Multi-Database Environment," in *Proc. of the Fifth International Conference on Database Systems for Advanced Applications (DASFAA)*, Melbourne, Australia, April 1-4, 1997, pp.391-400.

[23] R. Engelmore, T. Morgan, *Blackboard Systems*, Addison-Wesley Publishing Co., Reading, MA, USA, 1988.

[24] R. Hull, G. Zhou, "A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches," in *Proc. of the ACM SIGMOD Conference*, Montreal, QB, Canada, June 1996, pp. 481-492.

[25] R. Mattison, *Data Warehousing: Strategies, Technologies, And Techniques*, McGraw-Hill, NY, USA, 1996.

[26] R. Winter, *Efficient Data Warehousing for a New Era - Sybase Adaptive Server IQ Multiplex and the Needs of E-Business*, November 2000. http://www.sybase.com/detail_list/1,3691,2291,00.html (February 11, 2001)

[27] S. Abiteboul, R. Hull, V. Vianu, *Foundations of Databases*, Addison-Wesley Publishing Co., Reading, MA, USA, 1995.

[28] T.W. Ling, Y. Liu, "An Efficient View Maintenance Algorithm for Data Warehousing," in *Advances in Database Technologies: ER'98 Workshops on Data Warehousing and Data Mining, Mobile Data Access, and Collaborative Work Support and Spatio-Temporal Data Management*, Singapore, November 1998, pp. 169-180.

[29] W.H. Inmon, *Data Warehouse Performance*, John Wiley, NY, USA, 1999.

[30] W.H. Inmon, J. A. Zachman, J. G. Geiger, *Data Stores, Data Warehousing, and the Zachman Framework: Managing Enterprise Knowledge*, McGraw-Hill, NY, USA, 1997.

[31] W. J. Labio, Y. Zhuge, J. L. Wiener, H. Gupta, H. Garcia-Molina, J. Widom, "The WHIPS Prototype for Data Warehouse Creation and Maintenance," in *Proc. of the ACM SIGMOD Conference*, Tuscon, AZ, USA, May 1997.

[32] W. Litwin, L. Mark, N. Roussopoulos, "Interoperability of Multiple Autonomous Databases," in *ACM Computing Surveys*, Vol. 22, No. 3, September 1990, pp. 267-293.

[33] W. Wobcke, M. Pagnucco, C. Zhang, "Agents and Multi-Agent Systems - Formalisms, Methodologies, and Applications," *Lecture Notes in Artificial Intelligence 1441*, Springer, London, UK, 1998, pp. 164.

[34] Y. Breitbart, H. Garcia-Molina, A. Siberschatz, "Overview of Multidatabase Transaction Management," in *VLDB Journal*, Vol. 1, No. 2, October 1992, pp. 181-239.

[35] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom, "View Maintenance in a Warehousing Environment," in *Proc. of the ACM SIGMOD Conference*, San Jose, CA, USA, May 1995.

[36] Y. Zhuge, J. L. Wiener, H. Garcia-Molina, "Multiple View Consistency for Data Warehousing," in *Proc. of the International Conference on Data Engineering*, Birmingham, UK, April 1997.

[37] Y. Zhuge, H. Garcia-Molina, J. L. Wiener, "Consistency Algorithms for Multi-Source Warehouse View Maintenance," in *Journal of Distributed and Parallel Databases*, Vol. 6, No. 1, January 1998, pp. 7-40.