On Scheduling Data Loading and View Maintenance in Soft Real-time Data Warehouses

Nguyen Hoang Vu   Vivekanand Gopalkrishnan

Nanyang Technological University, 50 Nanyang Avenue, Singapore
ng0003vu@ntu.edu.sg, asvivek@ntu.edu.sg

Abstract

Data warehouses contain historic data providing information for analytical processing, decision making and data mining tools. However, several business intelligence applications nowadays require access to real-time data to make sound decisions. As a consequence, there is a great demand to incorporate new data from sources to the data warehouse as fast as possible. That motivates the construction of real-time data warehouse. Despite high demand, moving from a traditional data warehouse to a real-time one is not straightforward since we have to deal with the problem of efficiently scheduling various activities, viz., updates, view maintenances and OLAP transactions *in a timely manner*. In addition, OLAP transactions are now associated with deadlines (transaction timeliness) and data freshness requirement (data timeliness). Balancing between these two requirements poses another challenge in real-time data warehousing context. In this paper, we present an efficient technique aimed at updating data and performing view maintenance for real-time data warehouses while still enforcing these two timing requirements for the OLAP transactions. Our proposed approach aims at addressing the issues of applying updates and performing view maintenance while still effectively serving user OLAP queries in real-time data warehouse environment. Through extensive empirical studies, we demonstrate the efficiency of ORAD in achieving the goal of building a real-time data warehouse.

1 Introduction

A real-time data warehouse serves the purpose of monitoring the status of real-world objects in a dynamic environment. It enables enterprise managers to take effective decisions on the basis of the information that it provides them. Therefore, accuracy and timeliness of the information is a prime requirement. For example, in financial institutions, trading is done in time differences of seconds or minutes, which requires latest information regarding available prices and stocks. Too much latency in the data can cause the organization to lose a large amount of money.

15th *International Conference on Management of Data COMAD 2009, Mysore, India, December 9–12, 2009*
©*Computer Society of India, 2009*

In contrast to traditional data warehouses which perform batch-job data maintenance on a periodic basis [24, 25, 27, 13, 8, 26], in real-time data warehouses, updates are continuously supplied to the system, thus requiring the data to be refreshed to meet the freshness requirement. If these updates are not applied at the right time, this will result in transactions reading stale data, which would thus, present problems for the business organization. However, it is not just enough to apply updates; even business transactions must be allowed to read the updated data at the same time. But in a real-time system, updates and transactions come at a high speed which results in the need of an efficient scheduling strategy such that they do not conflict with each other. Particularly, query results in real-time data warehouses are expected on real-time data. Nevertheless, data may be refreshed while the query is being processed, so most real-time data warehouse systems allow queries to read some version of data that is valid from the time the query has started. Depending on the frequency of data updates, some systems even permit reading slightly stale data. This is a type of trading off data timeliness for transaction timeliness [11]. In any case, traditional process of running periodic maintenance batches [18] does not satisfy data freshness requirements of real-time data warehouses. Hence, in real-time data warehouses, we need efficient scheduling policies to deal well with write-only updates propagated from sources, view maintenances caused by updates, and read-only OLAP transactions.

When transactions' deadlines are top priority, reserving multiple data versions to avoid conflict between OLAP transactions and maintenance activities is shown to be an efficient solution [11, 18]. However, the versions take up too much space, and the cost for selecting data versions may affect transaction execution time, causing them to miss their deadlines. Though transaction deadlines may also be met by sacrificing data freshness requirements, this should

be done in a controlled manner, since letting transactions read too old data does not yield any business value. Therefore a good scheduling approach should reduce transactions missing deadlines, increase data freshness, and reduce version scanning costs. In particular, the scheduler should result in:

- low p_{MD}: percentage of transactions missing their deadline
- high $p_{success}$: percentage of transactions successfully meeting their deadline, without *reading* stale data
- low overhead (storage and access costs) of reserving multiple data versions

In order to achieve the above goals, we propose an efficient technique, called ORAD (*O*n-demand *R*e*A*l-time *D*ata Warehouse), for performing updates and view maintenances in *soft* real-time data warehouses (i.e., missing a transaction's deadlines makes the transaction useless, but it is not detrimental to the system) while enforcing the two timing requirements on OLAP transactions. ORAD, like other *on-demand* [1, 11] or *right-time* [23] approaches, performs updates and view maintenances only when required. Similar to [11], ORAD decouples these tasks to further reduce the system workload. However, unlike it, ORAD favors giving transactions *as fresh as possible* data versions, and proposes optimizations to enable easy availability of newer data versions. While this version selection scheme yields higher data freshness for transactions, it may lengthen the transaction's execution time by increasing the number of on-demand requests. However, through extensive theoretical and empirical analysis, we demonstrate that ORAD's optimizations help alleviate this adverse factor and lead to better performance than previous best-in-breed scheduling approaches [11], in terms of both transaction and data timeliness.

The rest of this paper is organized as follows. Related work is discussed in Section 2. ORAD approach is detailed and analyzed in Section 3. Section 4 describes our experimental setup procedure. Empirical comparisons with contemporary approaches are presented in Section 5. Finally, the paper is summarized in Section 6 with directions for future work.

2 Literature Review

This section provides a formal discussion on the current approaches towards both real-time database and real-time data warehouse. By decomposing the real-time data warehouse problem into two parts, the feasibility of applying real-time database techniques to real-time data warehouse is pointed out. Details are given below.

2.1 Background

The issues of building a real-time data warehouse can be divided into two complementary parts: (a) quickly and robustly transforming changes from operational data sources into data warehouse records, and (b) updating the data warehouse using those records in a timely manner. The first problem has been studied for a long time in the literature [5, 4, 28, 29]. The descriptions of such a method are as follows (see Figure 1):

- A monitor resides on the top of each individual data source from which it collects changes of interest and propagate to the data warehouse. If the source supports relational-style triggers, the monitor only needs to pass on that information. Otherwise, it may need to extract the difference between successive snapshots of the source data (called *delta changes* [9, 28]).
- A centralized integrator at the data warehouse is responsible for translating delta updates from monitors to final records for loading purpose. In cases when the table to be updated in the data warehouse is constructed from more than one source (e.g., a join view), the integrator needs to build appropriate queries and send them to related sources.

The second problem is surprisingly identical to what techniques in the real-time database field pursue [1, 11]. In particular, they are also concerned with integrating updates to the database as soon as possible. The only difference lies in the availability of such updates: in real-time data warehouse, updates are normally obtained by joining with other data sources whereas in real-time database, they are assumed to be available at hand [11]. Hence in some sense, much of what is known about real-time database scheduling, including theory and specific algorithms to testing procedure, can be applied in real-time data warehouse (obviously with some modifications). Our approach, similar to [23], deals with the second problem. More specifically, updates considered in ORAD are assumed to be in the format that can be used to update the data warehouse directly.

2.2 Related Work

Table 1 shows the characteristics of our proposed approach in comparison to those of outstanding techniques in the literature. The detailed discussion is given below.

In the field of real-time database, Adelberg. et al [1] introduce four different algorithms for scheduling updates and transactions (but not view maintenances) namely Updates First, Transactions First, Split Updates and On-Demand Updates. Extensive experimental results demonstrate that On-Demand Updates

Technique	Real-time	Trans. TL.	Data TL.	Schedule
Our technique	$\surd$	$\surd$	$\surd$	Updates, View Maintenances, Trans.
Kao et al. [11]	$\surd$	$\surd$	$\surd$	Updates, View Maintenances, Trans.
Adelberg et al. [1]	$\surd$	$\surd$	$\surd$	Updates, Trans.
Adelberg et al. [2]	$\surd$	$\surd$	$\surd$	View Maintenances, Trans.
Kim et al. [12]	$\surd$	$\surd$	Not addressed	Updates, Trans.
Santos et al. [20]	$\surd$	Not addressed	$\surd$	Not formally addressed
Thomsen et al. [23]	$\surd$	Not addressed	$\surd$	Not formally addressed
Thiele et al. [21]	$\surd$	Not addressed	$\surd$	Updates, Trans.
Luo et al. [16]	Not addressed	N/A	N/A	N/A
Quass et al. [18]	Not addressed	N/A	N/A	N/A

Table 1: Comparison of existing techniques (*Trans. TL.* is transaction timeliness, *Data TL.* is data timeliness, *Trans.* is User Transaction).

yields the best performance among all the algorithms in terms of both p_{MD} and $p_{success}$. Transactions First (TF) also performs decently, as it yields high response time and is good at meeting transaction deadlines. However, the transactions end up with lower quality (too stale) data, thus, making it inefficient for real-time use. In another paper [2], they explore the problem of balancing between view maintenances and transactions and observe that view recomputations often come in bursts, following the principle of *update locality*. The pursued approach nevertheless, similar to [1], do not consider the issue of scheduling in a system where three types of activities are present.

Kao et al. [11] enhance the results obtained in [1, 2] by proposing three scheduling policies, viz., first-Updates, then-Recomputations, finally-Transactions (URT), On-Demand and On-Demand-Hybrid that also take into account view maintenances. Among these, the On-Demand-Hybrid approach which allows transactions to read slightly stale data (ODRTB, called ODH-RCS in [11]) yields the lowest value for p_{MD}. The article also considers updates and view maintenances to be decoupled, which helps reduce unnecessary workload caused by view maintenance activities. Besides, these two activities are only carried out when required, therefore, significantly reduce the overall system load. However, the paper does not present an efficient method to reduce the storage or I/O overheads of reserving multiple data versions, which may worsen the deadline missing rate [18]. Though ODRTB was designed for real-time databases, it is also applicable for real-time data warehouses since its goal is to schedule activities to minimize p_{MD}.

Considering the work done in data warehouses, Quass et al. [18] propose 2VNL (two-version no locking), a technique to support a 24-hour online data warehouse that allows readers and maintenance transactions to operate at the same time without locking each other. An OLAP transaction in 2VNL has a high probability of being restarted if its required data has expired, while nVNL mitigates this problem by maintaining multiple versions of data, similar to RCS strategy proposed in [11]. However, in 2VNL as well as nVNL, only updates and transactions are considered. Furthermore, both 2VNL and nVNL are designed to handle only one maintenance transaction at a time, and if this assumption were to be relaxed, they would need significant modifications to be usable in a real-time data warehouse. In addition, since neither approach was designed for real-time purposes, no explicit consideration is given to transaction deadlines.

For soft real-time data warehouses, Kim et al. [12] propose COB, a technique that allows multiple maintenance transactions (updates) and OLAP transactions (but no view maintenance), to be executed concurrently while ensuring serialization. Besides, a data object is updated when it is not being accessed by any OLAP transaction, i.e, data freshness is obviously uncontrolled. COB is in fact an incomplete version of the Transaction-First (TF) scheduling algorithm [1], which while yielding a low value for p_{MD}, gives transactions very low quality data (i.e., $p_{success}$ is very low). On demand (OD) on the other hand, has been shown to yield the best overall performance among other scheduling algorithms, including TF [1].

The issue of data loading in real-time data warehouses is also studied in [20, 23]. In [20], real-time data is stored in staging tables having a schema structure similar to the original data warehouse. Although their problem addressed is identical to ours, the critical issue of conflict management among updates and OLAP transactions is not handled at all. Furthermore, view maintenances are also not mentioned.

Thomsen et al. [23] present RiTE, a technique providing an on-demand data loading methodology for real-time data warehouses. RiTE is similar to our approach in the sense that data is only loaded when required. Particularly, the issue of data loading in RiTE is modeled as a provider-consumer architecture in which a provider driver serves data requests from multiple consumer ones. Data coming from sources (already processed) are managed at the provider site, and only loaded to the *catalyst* (like a staging area) when there is a need by any consumer. For each data warehouse table, a view is constructed to: (a) integrate its data residing in both the warehouse and the cata-

lyst, (b) make the process of updating data be transparent to consumers. For reducing the catalyst's size, ad-hoc data loadings from the catalyst to the warehouse are performed whenever a triggering condition is met (e.g., the system load is below a specified threshold). However, as a pioneer work in real-time data warehouse, updates considered in RiTE are insertions of new data records only. RiTE does not consider the read-write data conflicts between updates and OLAP transactions, and hence does not address our problem. However, when updates are append-only, RiTE can be incorporated into our technique for faster data loading.

Thiele et al. [21] introduce WINE, a technique for scheduling updates and OLAP queries based on the notion of QoS (quality of service) and QoD (quality of data). To a certain extent, QoS and QoD are equivalent to the transaction and data timeliness requirements. However, QoS is based mainly on transaction response time, and hence does not address the importance of meeting transaction deadline whereas it is the most important factors in real-time systems [1, 2, 11]. As pointed out in [1], as long as transactions meet their deadlines, the response time issue can be ignored. WINE also does not consider the problem of view maintenances. Thus, we choose not to compare with WINE in our empirical study. Another report documented in [22] recently by the same author proposes the concepts of global and local strategies for scheduling activities in real-time data warehouses. Similar to [21], it assesses the optimal way to balance between data freshness and response time requirements of user queries. This is once again different from our philosophy taken in this paper. Therefore, details about and comparison with the technique are not further explored.

A novel method for reordering transactions to improve transaction throughput is proposed in [16]. It functions by exploiting the dependencies among and knowledge on the currently running transactions and the transactions waiting to be run. Nevertheless, it does not explicitly consider the problem of scheduling updates and view maintenances whereas these are the two important factors in our scheduling algorithm.

Recent work by Brucker et al. [6] propose a J2EE architecture for an ETL environment which supports real-time data extraction and transformation. In this paper, we assume such real-time ETL (RTETL) exists.

3 Methodology

In a classical data warehouse, the ETL module performs data extraction, transformation and loading in a batch mode. When moving to a real-time data warehouse, the ETL is improved to operate in a timely-oriented manner. Recent work by Brucker et al. [6] proposes a J2EE architecture for an ETL environment which supports real-time data extraction and transformation. The RTETL asynchronously propagates

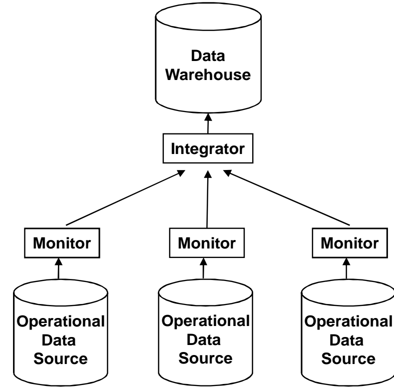


Figure 1: Monitor-Integrator Architecture.

changes from operational data sources to the real-time data warehouse. Pending updates are then stored in a special data area called *unapplied set*, structurally similar to that in [11]. The real-time data warehouse system comprises of the current data objects in the data warehouse along with the *unapplied set*. Our goal is to schedule the pending updates and OLAP transactions on the data objects. The real-time data warehouse architecture of our method is shown in Figure 2.

3.1 The solvability condition and data versions

We make the following assumption in our work which we call the condition for *the solvability of the problem*: "For any interval of time of length Δt where $0 \leq \Delta t < \infty$ and for any data object o, the number of updates for o arriving at the data warehouse during this interval is finite". This is derived from the fact that computing resources in practice are limited. This assumption is also applicable to [10, 11, 18, 12] but it is implicitly used. The assumption implies the average number of updates arriving in one unit of time is finite. This is practically true since even in fast domain like financial stock, this number is just about 500 updates per second [11].

Using this assumption, we build the index for the set of versions of a data object o as follow:

- **Step 1:** Before the very first burst containing the insertion of o arrives at the data warehouse, we set $i = 1$.
- **Step 2:** For every burst of updates on o arriving at the data warehouse, we denote its arrival time by t and the set of updates contained in the burst as S_B. For every update $U \in S_B$, we denote the data carried by U as $U.o$. Due to the above assumption, the number of updates for o arriving during the time interval $[t, t]$ is finite, i.e. S_B is a finite set. Thus, the cardinality of S_B can be represented as some integer N [7].

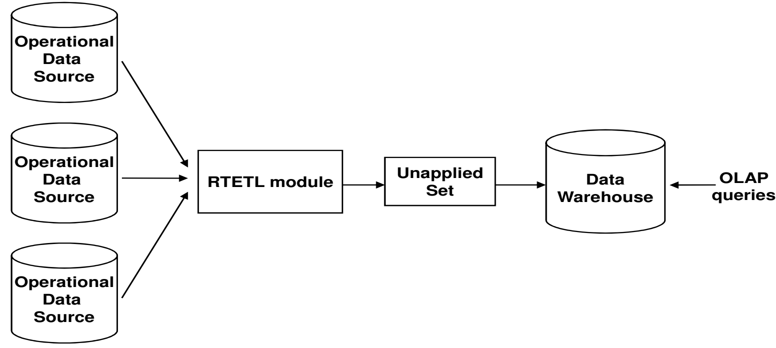


Figure 2: Real-time data warehouse architecture.

Therefore, there exists a bijection F from S_B to $\{i, i+1, \ldots, i+N-1\}$. Choosing such an F, for every update $U \in S_B$, $F(U)$ is called the index of $U.o$ and $U.o$ is denoted as $o_{F(U)}$.

- **Step 3:** For any subsequent burst of updates on o arriving at the data warehouse, repeat step 2 with $i = N$.

From the indexing scheme, it can be seen that for any two different versions of o, their indexes are different. In practice, the function F can be decided based on the order of each update in the unapplied set, thus, it is intuitive to assign lower index values for updates nearer to the start of the set.

Using the above assumption, it can also be derived that for any interval of time of length Δt where $0 \leq \Delta t < \infty$ and for any data object o, the number of versions of o available in the data warehouse during this interval is finite (since each version is created by one update).

3.2 Preliminaries

Data objects can be classified as base and view data objects. A view data object is derived from one or more base data objects. As a preliminary work, we do not consider views derived from other views as well as index maintenance [9]. These are considered in future work. Versions of a data object o are indexed using the set of integers starting at 1 (c.f., Section 3.1). A version with index i is denoted as o_i, and the next version o_{i+1} invalidates o_i if and when it arrives. Version management schemes sequentially delete old versions based on various policies [11, 18].

Definition 1. [VALIDITY INTERVAL] The validity of o_i, denoted as $VI(o_i)$, is defined as $[LTB(o_i), UTB(o_i))$ where $LTB(o_i)$ is the lower time bound, and $UTB(o_i)$ is the upper time bound of o_i. $VI(o_i)$ specifies that o_i is valid from $LTB(o_i)$ and becomes invalid from $UTB(o_i)$ onward.

If o is a base data object, then $LTB(o_i)$ equals the arrival time of the ith update on o in the data warehouse system, and $UTB(o_i)$ equals the arrival time of the next update on o (or ∞ if there was no further update). If o is a view data object, the validity interval of o_i depends on whether or not it was updated (recomputed) on demand by an OLAP transaction, as explained in Section 3.3.

Let us denote the start time of an OLAP transaction T as $s(T)$, the arrival time of an update U as $arr(U)$, and $Ver(o, t)$ as the set of version numbers of any data object o that can be found at time t in the data warehouse system. Consider a base data object o having some version in the data warehouse system at time t (i.e., $Ver(o, t) \neq \emptyset$), and let m be the smallest element in $Ver(o, t)$.

Lemma 1 *The union of validity intervals of all versions of o forms an open continuous range which is lower bounded by the LTB of the oldest version. Mathematically,*

$$\bigcup_{i \in Ver(o,t)} VI(o_i) = [LTB(o_m), \infty)$$

Proof. From Definition 1, we know that $UTB(o_i) = LTB(o_{i+1})$, and the last version has $UTB = \infty$. Also, since only old versions are deleted sequentially, the validity intervals of successive versions of o_m can be combined to give the continuous range as desired.

Consider an OLAP transaction T which is based on a set of data objects $QuerySet(T)$. Let $ReadSet(T)$ be the corresponding set of data versions that are actually read, one version for every object $o \in QuerySet(T)$, and let $VI(T)$ be the validity interval for candidate versions of the remaining objects to be read. Initially, when T starts at $s(T)$, $VI(T)$ is set to $[s(T), \infty)$. As T reads desired data objects, its validity interval may change in order to avoid reading inconsistent versions of data objects. Particularly, after reading a data version o_i such that $VI(o_i) \cap VI(T) \neq \emptyset$, $VI(T)$ is set to $[max(LTB(T), LTB(o_i)), min(UTB(T), UTB(o_i)))$. If the scheduler cannot find any version for a required

data object within $VI(T)$, then T is aborted. So a candidate data version must be valid within $VI(T)$.

Data inconsistency may also be caused by append-only updates [23], so a transaction should only read data already available in the data warehouse system when it begins. If this requirement is not enforced, using ODRTB [11], a transaction T may be aborted if it tries to read a data object that is not in the data warehouse system before $s(T)$. In other words, if T reads a base object o at time t_r, for every version $k \in Ver(o, t_r), LTB(o_k) > UTB(T)$. Such object o does exist if it was created by an append-only update U which arrived before T could read o, and by that time, its validity interval had reduced so that $UTB(T) < arr(U)$. ODRTB does not handle this problem of append-only updates explicitly, but in ORAD, we enforce this requirement. More specifically, only data objects $o \in QuerySet(T)$ such that $Ver(o, s(T)) \neq \emptyset$, are considered by T. This solution can also be used by ODRTB.

Besides remaining valid within $VI(T)$, in order to maintain correctness, the data versions in $ReadSet(T)$ should also be relatively consistent with each other.

Definition 2. [RELATIVE CONSISTENCY] A set of data versions S (of different data objects) is said to be relatively consistent if

$$\bigcap \{VI(o_i) : o_i \in S\} \neq \emptyset$$

So the scheduling algorithm should ensure that all objects in $QuerySet(T)$ should be represented by some version in $ReadSet(T)$, and all versions should be relatively consistent with each other and valid within $VI(T)$.

3.3 Scheduling Algorithm

In our scheduling policy, priorities among OLAP transactions are set using the *earliest-deadline-first* priority assignment. A tardy transaction is useless and should be aborted by the system to avoid wasting resources [11]. Similar to [18], no locking protocol is used for data accesses, since there are no read-write conflicts among OLAP transactions and other write activities.

Recall that when the requirement of restricting append-only updates is enforced, base data object o is only considered by a transaction T if some version of it existed in the data warehouse system when the transaction started. From Lemma 1, we know that at time $s(T)$, the union of all validity intervals of o's versions forms an open continuous range whose lower bound is the LTB of the oldest version (at time $s(T)$). Since $Ver(o, s(T)) \neq \emptyset$, this range contains $s(T)$. Recalling $VI(T)$ is set to $[s(T), \infty)$ initially, it is *intuitive* to see the scheduler always finds a version of o no matter how wide $VI(T)$ is. Thus, in ORAD, in order to increase the freshness of data read by T, when T reads an object o (base or view), we always choose the latest version of o whose VI overlaps with that of T. The details of this approach are presented below, and the correctness of the procedure is shown in Theorem 1.

When update U arrives on a base data object o: The process for handling U is described in Algorithm 1 (Process Update). Let o_i be the data version of o carried in U. $VI(o_i)$ is set to $[arr(U), \infty)$. Concurrently, $UTB(o_{i-1})$ is set to $arr(U)$ signifying that o_{i-1} is only valid up until the arrival time of o_i. Let v_j be the latest version of any view data object v derived from o. If $UTB(v_j) = \infty$, we set $UTB(v_j)$ to $arr(U)$. Transactions reading any data objects affected by the arrival of U will have their UTB changed accordingly.

Algorithm 1: PROCESS UPDATE

```
1  Set VI(o_i) = [arr(U), ∞)
2  Set UTB(o_{i-1}) = arr(U)
3  foreach transaction T that has read o_{i-1} do
4      Set UTB(T) = min(UTB(T), UTB(o_{i-1}))
5  foreach view data object v derived from o do
6      Set v_j to the latest version of v
7      if UTB(v_j) = ∞ then
8          Set UTB(v_j) = arr(U)
9          foreach transaction T that has read v_j do
10             Set
               UTB(T) = min(UTB(T), UTB(v_j))
```

When OLAP transaction T arrives: The process for handling T is described in Algorithm 2 (Process Transaction). T's validity interval is set to $[s(T), \infty)$. For each data object o read by T (i.e., $Ver(o, s(T)) \neq \emptyset$):

- If o is a base data object: We choose o_i, the latest version among all versions of o available in the data warehouse system, such that $VI(o_i) \cap VI(T) \neq \emptyset$. If o_i has not been applied (still in the unapplied set), an update (OD-update) is triggered inheriting the priority of T. When the OD-update finishes, T reads o_i and $VI(T)$ is set to $VI(o_i) \cap VI(T)$. This is different from ODRTB [11] which selects the version whose VI has biggest intersection with $VI(T)$.

- If o is a view data object: Similar to the case of base data objects, we choose the latest version o_i of o such that $VI(o_i) \cap VI(T) \neq \emptyset$. If such version o_i exists, we set $VI(T)$ to $VI(o_i) \cap VI(T)$. Otherwise (i.e., no current version of o has VI overlapping with $VI(T)$), a recomputation (OD-recom) R to compute a relevant version of o for T is triggered. R inherits the priority and the validity interval of T, and is processed in the same way as an OLAP transaction. The new version of

o has VI equal to the intersection of $VI(R)$ and all the validity intervals of the data versions read by R.

Algorithm 2: PROCESS TRANSACTION

```
1  Set VI(T) = [s(T), ∞)
2  foreach data objects o read by T do
3      Set o_i to the latest data version of o such that
       VI(o_i) ∩ VI(T) ≠ ∅
4      if o is a base data object then
5          if o_i has not been applied then
6              Trigger an OD-update to apply o_i
7          Set VI(T) = VI(T) ∩ VI(o_i)
8      else
9          if no version of o has VI overlap with
           VI(T) then
10             Trigger an OD-recom R for T
11             Set o_i to the version computed by R
12         Set VI(T) = VI(T) ∩ VI(o_i)
13 Commit T
```

When the system is idling: The data warehouse system is considered to be idle when there is no running OLAP transaction. During the idle period, we apply updates and perform view maintenances using the URT scheduling policy [11] and switch back to our normal scheduling policy when a new transaction arrives.

3.4 Theoretical justification of our heuristic

Now we shall analyze the correctness of the heuristic used in ORAD. In other words, we must show that a transaction will be able to find a required data object in the data warehouse system, as long some version of the object existed before the transaction started. Consider an active transaction T and a base data object $o \in QuerySet(T)$ that exists in the data warehouse system when T starts (i.e., $Ver(o, s(T)) \neq \emptyset$). Let us denote the time T reads o as t_r.

Theorem 1 *At t_r, there exists a version o_i of o in the data warehouse system such that $VI(T) \cap VI(o_i) \neq \emptyset$.*

Proof. From Lemma 1, the union of all validity intervals of o's versions that can be found in the data warehouse system at t_r forms a continuous range $[LTB(o_m), \infty)$ where m is the minimum element of $Ver(o, t_r)$. We now prove that $LTB(o_m) \leq LTB(T)$.

Assume $LTB(o_m) > LTB(T)$. If $m = 1$ then o_m is the first version of o. Since o has some version in the data warehouse system before $s(T)$, we have $LTB(o_m) = LTB(o_1) \leq s(T) \leq LTB(T)$. This contradicts with $LTB(o_m) > LTB(T)$. Therefore, $m > 1$ and o_{m-1} is defined.

Note that $UTB(o_{m-1}) = LTB(o_m)$, i.e., $UTB(o_{m-1}) > LTB(T)$. Thus, o_{m-1} must also be present in the data warehouse system no matter what version pruning schemes are utilized (Section 3.5). This contradicts with the fact that m is the minimum element of $Ver(o, t_r)$. Hence, it holds that

$$LTB(o_m) \leq LTB(T)$$

i.e.,

$$VI(T) \cap [LTB(o_m), \infty) \neq \emptyset$$

Let

$$z \in VI(T) \cap [LTB(o_m), \infty)$$

From Lemma 1, there exists $i \in Ver(o, t_r)$ such that $z \in VI(o_i)$. In other words,

$$VI(T) \cap VI(o_i) \neq \emptyset$$

According to Theorem 1, if a base data object o read by T exists in the data warehouse when T starts, at the time T reads o, ORAD will always be able to find a version o_i of o such that $VI(o_i) \cap VI(T) \neq \emptyset$, regardless of the width of $VI(T)$. This is an important finding opposing the theory proposed in [11] that the wider $VI(T)$, the more likely that T can find a version of o that is consistent with what T has already read. Our theory on the other hand specifies that we only need to choose latest version of base data object whose VI has some intersection with $VI(T)$, no matter how wide the intersection is. The search for data versions for transactions under ORAD also becomes easier since we only need to look for the latest data version whose VI overlaps with that of T. However, this comes with some costs which are mentioned in Section 3.5. For view data objects, if no versions are found in the data warehouse system, an OD-recom will be triggered. Since an OD-recom is processed in the same way as an OLAP transaction and it reads only base data objects, using Theorem 1, a version relevant for T is always computed successfully. In other words, the scheduler is always able to find a required data object in the data warehouse system for transaction T.

3.5 Optimizations

ORAD, like ORDTB [11] performs updates and view maintenances during the idle period. According to [11], this causes most of the data to be kept up to date and reduces transactions delay due to on-demand requests (OD-updates and OD-recoms). However, since ORAD favors recent data versions more, the number of on-demand requests required by OLAP transactions may increase. Therefore we propose to prioritize updates based on their potential to be called on demand. Another side-effect of our heuristic is that it permits

efficient mechanisms to reduce the overhead (storage and searching costs) of reserving multiple data versions.

3.5.1 Reducing the number of on-demand requests

We use a cache (C_{ft}) to keep track of data objects that have higher frequency (and hence future potential) of access by OLAP transactions. This helps bring updates on more frequently-accessed data to be applied first during the system idle period, thus reducing the workload of applying updates and performing view maintenances after that period, which in turn decreases the on-demand requests triggered by transactions.

C_{ft} may be implemented easily by storing the identities of objects themselves (as in this work), or by storing *primitive terms* [15, 17] used in the selection conditions of OLAP transactions. This data structure can be based in memory, by restricting the number of object identities/primitive terms stored in it. To satisfy the space constraint, similar to the approach in [15], the standard clock algorithm (an approximation of the LRU algorithm) is employed to maintain C_{ft}. The maintenance is carried out on the commit of every OLAP transaction. When the system is idle, updates to data objects transmitted from RTETL are prioritized according to the order of object identities/primitive terms in C_{ft} that they satisfy. Thus, C_{ft} helps bring more important updates to be applied first.

As a consequence, not only does the freshness level of data increase, but the amount of data that transactions have to materialize is also further reduced. Few transactions require materialized views which are invalidated by updates [5, 17, 19], and efficient incremental maintenance strategies can be used to quickly update them. The benefit of such prioritization was demonstrated in [1], where it was assumed that the data could be classified into *importance* levels. However, as mentioned in the same article, such classification is very hard to obtain in practice since data could have many different values. Our heuristic is therefore more applicable in the real-world context.

3.5.2 Reducing memory overhead to store data versions

In ORAD, the memory overhead of storing multiple data versions is minimized by applying the following observations.

Under our policy, a data version whose UTB is not greater than the LTB of a transaction would never be accessed by it. So, given $\mathcal{T}$, the set of active transactions in ORAD, we intuitively prune out those data versions whose $UTB \leq LTB(\forall \mathrm{T} \in \mathcal{T})$. This pruning process can be carried out when an OLAP transaction starts or commits. ODRTB on the other hand only discards those data versions whose $UTB \leq s(\forall \mathrm{T} \in \mathcal{T}) - \Delta$. In [11], Δ is chosen to be large ($10s$), in order to achieve a low p_{MD} . So, since $s(T) \leq LTB(T)$ holds $\forall \mathrm{T} \in \mathcal{T}$ in ORAD, and due to the high Δ, the pruning rule used by ODRTB is too weak compared to ORAD's.

Another consequence of choosing the latest data versions whose VI overlaps with that of the active transactions, is that any transaction T starting after the idle period only requires latest data versions at the time the period ends (using Theorem 1 and the fact that when T starts, $VI(T) = [s(T), \infty)$). Hence, it is unnecessary to reserve old versions of data objects but only the latest ones during the idle period. Thus, URT for maintaining absolute accuracy (URT/ACS in [11]) is the most suitable choice. However, we make some modifications compared to [11]. Particularly, each base data object is still associated with a VI equals to $[arr(U), \infty)$ where U is the latest update on the data object. On the other hand, each view data object is associated with a VI equals to $[arr(U_{max}), \infty)$ where U_{max} is the update with the largest arrival time among others in the batch used for refreshing the view data object. When the system is idle, ORAD maintains only one data version per data object while ODRTB still uses the same pruning rule. All these indicate that the memory overhead incurred by ORAD is smaller than that of ODRTB. This further reduces the version scanning cost of ORAD.

3.5.3 Reducing I/O overhead to access data versions

Since ORAD favors more recent data versions, its I/O overhead of scanning versions is greatly reduced. Particularly, when looking for a suitable data version for a transaction T, we only need to scan starting from the latest version and stop when the first satisfied version is identified. In the best case, the searching process can be done with only one version scan.

ODRTB [11], on the other hand does not propose any explicit scheme to minimize the I/O cost of searching data versions. Assuming the interested data object o has more than one version at the time T reads o, the width-based selection scheme of ODRTB has to scan: (a) in the best case, at least two data versions, and (b) in the worst case, all the versions of o, to identify the relevant one if every existing version's VI has some overlap with $VI(T)$. The overhead becomes worse if the number of data versions per object is high. When the I/O overhead per data version is not trivial (which is very likely as mentioned in [18]), our method will take lesser time to serve a transaction.

4 Simulation Setup

Though simulations may not give a complete picture of what really happens, it is shown to be very successful in simulating various real-time systems, real-

Parameter	Description	Value
λ_B	Arrival rate of update bursts (/sec)	1.2
λ_U	Arrival rate of updates within a burst (/sec)	33
λ_T	Arrival rate of OLAP transactions (/sec)	2.0
$[F_{o_min}, F_{o_max}]$	Fan-out	[0, 4]
$[BS_{min}, BS_{max}]$	Burst size	[1, 12]
N_{op}	Number of read operations performed by a transaction	50
$[S_{min}, S_{max}]$	Slack factor	[1.3, 3.0]
N_{BASE}	Numbers of base data objects	6000
N_{MV}	Number of materialized view objects	600
$p_{frequent}$	Rate of "hot" object accesses in an OLAP transaction	0.7
p_{uf}	Rate of update on frequent data	0.5
t_{ver}	I/O cost to process a data version (ms)	2.0
t_{IO}	I/O cost per operation (ms)	5.0
t_{CPU}	CPU cost per operation (ms)	1.0
N_{max}	Maximum size of C_{ft}	600
N_f	The number of data objects frequently accessed	3000

Table 2: Description of the parameters.

time databases and real-time data warehouses in the literature. Thus, as with several previous works in both real-time data warehouse and real-time database [1, 11, 12, 14, 22], we use *simulated experiments* to demonstrate the effectiveness of our approach.

Although ODRTB [11] targets real-time databases, its core idea can also be applied to the real-time data warehouse architecture employed in this paper (c.f., Section 2). Other approaches [1, 12, 18, 20, 23, 21] do not address all the issues of scheduling updates, view maintenances and OLAP transactions, and are therefore not compared.

We use the following metrics to assess the efficiency of ORAD and ODRTB:

- the transaction deadline miss rate p_{MD}
- the fraction of transactions meeting their deadline without having read stale data $p_{success}$
- the average number of versions per data object n_{avg}

The first and second metrics have been used widely in existing work [1, 11]. The last metric is utilized to measure the memory consumption of ORAD and ODRTB. Every time when version pruning finishes, the average number of versions per object is recorded. At the end of the simulation, n_{avg} is taken to be the average of these values. The simulation setup is the same as in [11]. The simulation parameters and their respective default values are shown in Table 2. It is highlighted that the parameter values are chosen as reasonable values for a typical financial application [1, 2, 11]. More descriptions are given below.

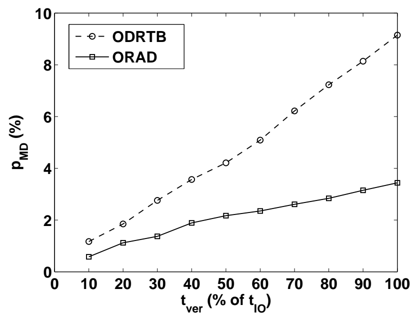
Data Warehouse Model. The numbers of base (N_{BASE}), view data objects (N_{MV}), and frequently accessed (N_f) data objects in the data warehouse are set to 6000, 600, and 3000 respectively. A similar frequency rate is chosen in [1]. All objects in our model are disk-based. The I/O cost to access a version during the searching process is represented as t_{ver}. In [11], the value of t_{ver} is relatively small. In this work, we vary t_{ver} from 10% to 100% of t_{IO} (the I/O cost per transaction operation) since such costs are not trivial in practice [18]. The number of view objects that a base one derives (i.e., fan-out) is chosen randomly from the uniformly distributed range $[F_{o_min}, F_{o_max}]$. Likewise, each view object has a (random) number of base parents.

Update Model. Updates come in the form of bursts where burst arrival is modeled as a Poisson process with an arrival rate λ_B. The number of updates within a burst is sampled uniformly from the range $[BS_{min}, BS_{max}]$. Furthermore, update arrival in a burst is simulated as another Poisson process with an arrival rate λ_U. An update has a probability of p_{uf} to be on a frequently accessed object, and as in [1], its default value is set to 0.5.

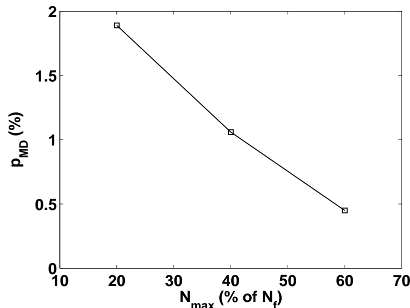
Transaction Model. Transaction arrival on the other hand is modeled as a Poisson process with arrival rate λ_T. The number of read operations performed by a transaction (N_{op}) is set to 50, and the probability that a read operation accesses a frequent data object ($p_{frequent}$) is set as 0.7. The associated I/O (t_{IO}) and CPU (t_{CPU}) costs per operation are set to 5.0 and 1.0 ms, respectively. Similar to *all* approaches for real-time study [1, 11, 12], the deadline of a transaction T is set to:

$$d(T) = arr(T) + SL \times ex(T)$$

where $arr(T)$ is the arrival time of T, SL is the slack factor (uniformly distributed in the range $[S_{min}, S_{max}]$), and $ex(T)$ is the expected execution



(a) p_{MD} vs. t_{ver}



(b) p_{MD} vs. N_{max}

Figure 3: Effect of parameters on deadline misses.

time of T. Particularly:

$$ex(T) = N_{op} \times (t_{IO} + t_{CPU})$$

To ensure that C_{ft} does fully capture frequent access patterns of transactions, its size, N_{max} is expressed as a percentage of N_f.

5 Empirical Results and Analysis

p_{MD} **vs.** t_{ver}**:** This experiment investigates the effect of the version-scanning I/O overhead (t_{ver}) on p_{MD} by varying the value of t_{ver} from 10% to 100% of the I/O cost per operation (t_{IO}). The result (Fig. 3(a)) shows that as t_{ver} increases, p_{MD} of both algorithms increases but ORAD scales better than ODRTB. This complies with our argument about the I/O overhead of version scanning. The transaction deadline miss rate of ODRTB is very sensitive to t_{ver}. That is because when t_{ver} approaches t_{IO}, the scanning cost approaches the cost to perform one read operation, i.e., slack factor is less likely able to compensate the execution time of transactions. If each data object read by an OLAP transaction has more than one version suitable for it, ODRTB's total cost to perform one operation is about *three* times t_{IO} (since at least two versions are accessed). In contrast, the impact of t_{ver} on ORAD's performance is relatively mild because of our version selection scheme is less affected by t_{ver} and less dependent on the number of versions per data object. As observed during the simulation, ORAD usually needs only one scan to identify the relevant data version. On the other hand, ODRTB usually requires three to four times on average.

p_{MD} **vs.** N_{max}**:** This experiment aims to assess the impact of the maximum size of C_{ft} (N_{max}) on the performance of ORAD. It is noted that the larger N_{max} is, the better C_{ft} is in caching frequently accessed data. Because of that, N_{max} value is varied from 20% to 60% of N_f. This setting is achievable since C_{ft} only stores object identities. Similar setting is used in [15]. The result in Fig. 3(b) shows that p_{MD} decreases as N_{max} increases, i.e., more transactions meet their deadlines. This once again conforms to our argument in Section 3.5. However, even with small cache size (20% of the total number of frequently accessed objects), the transaction deadline miss rate of ORAD is still less than 2%. This implies the efficiency of other optimizations used in ORAD.

p_{MD} **and** $p_{success}$ **vs.** λ_T**:** This experiment evaluates the impact of transaction arrival rate (λ_T) on p_{MD} and $p_{success}$ of both ORAD and ODRTB. We vary λ_T from 0.5 to 5.0 transactions per second. All the other parameters are set to their default values. We observe from the result (Figs. 4(a) and 4(b)) that as λ_T increases, p_{MD} increases while $p_{success}$ decreases. That means, when the system load is high, both transaction timeliness and data timeliness decrease. This agrees with previous studies [1, 11]. However, ORAD always achieves lower p_{MD} and higher $p_{success}$ than ODRTB. That results from the total effects of our optimizations.

We also investigated the effect of update burst arrival rate λ_B on both techniques performance. Particularly, for each value of λ_B, we run the simulation with λ_T varying from 0.5 to 5.0 transactions per second and observe how p_{MD} and $p_{success}$ change. The values obtained also show that ORAD always yields lower p_{MD} and higher $p_{success}$ than ODRTB.

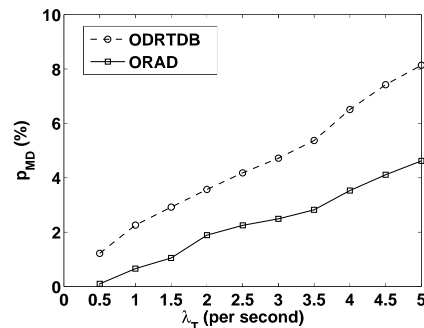
n_{avg} **vs.** λ_T**:** To assess the memory overhead of ORAD and ODRTB, we keep default values for other parameters while varying λ_T from 0.5 to 5.0 transactions per second. From the results obtained (Fig. 4(c)), we see that ORAD always consumes less memory than ODRTB. More specifically, ODRTB's memory overhead overall is about two to three times that of ORAD. This shows that our pruning strategy is much more efficient than that of ODRTB. Furthermore, the high memory overhead of ODRTB worsens its version scanning overhead since there are more candidate versions to access. In general, ORAD requires less than two versions per data objects. Hence, ORAD's memory consumption is also less than that of 2VNL [18].

6 Conclusions and future work

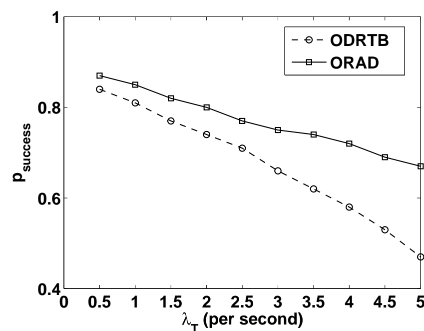
In this paper, we first factored the issues of building real-time data warehouses into two separate parts and through that, pointed out the feasibility of applying available scheduling techniques in the field of real-time database to real-time data warehouse. We then pre-

sented an efficient approach for data loading and view maintenance in soft real-time data warehouses. Our approach, called ORAD, aims at scheduling updates, view maintenances and OLAP transactions while still enforcing the transaction and data timeliness requirements of OLAP transactions. Through extensive empirical study, ORAD has been shown to outperform current the state-of-art approach on every aspect.

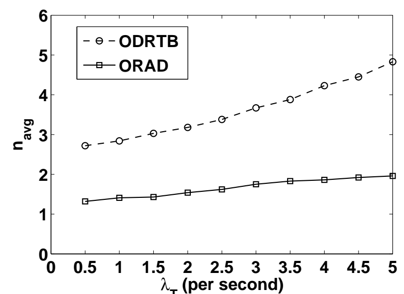
In future work, we are considering to integrate ORAD with RTETL into an open source RDBMS, and evaluate its performance with actual system studies. We are also investigating the issue of index maintenance in real-time environment as well as batching (all operations are performed on a per object basis) since under the batch setting, our method may suffer high overheads. However, object clustering can be employed in this scenario to cluster similar objects into the same group and the scheduling policy is designed to work with groups of objects instead of individual entities. As a consequence, this will help reduce the overheads caused by batching. The notion of object similarity is an interesting point to explore. Another promising research direction is to combine the findings here to those in stream mining, like outlier detection in time series [3], to construct a complete system for learning tasks. It is widely known that most data mining applications rely on the data provided by traditional data warehouses. With the availability of real-time data warehouses, it is natural to utilize them in streaming and time series contexts where lots of updates and data retrievals (read activities) need to be performed in real-time. Thus, a complete prototype as such will definitely be beneficial for the knowledge discovery process.



(a) p_{MD} vs. λ_T



(b) $p_{success}$ vs. λ_T



(c) n_{avg} vs. λ_T

Figure 4: Effect of transaction arrival rate.

References

[1] Brad Adelberg, Hector Garcia-Molina, and Ben Kao. Applying update streams in a soft real-time database system. In *SIGMOD Conference*, pages 245–256, 1995.

[2] Brad Adelberg, Ben Kao, and Hector Garcia-Molina. Database support for efficiently maintaining derived data. In *EDBT*, pages 223–240, 1996.

[3] Fabrizio Angiulli and Fabio Fassetti. Detecting distance-based outliers in streams of data. In *CIKM*, pages 811–820, 2007.

[4] Jose A. Blakeley, Neil Coburn, and Per-Ake Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Transactions on Database Systems*, 14(3):369–400, 1989.

[5] Jose A. Blakeley, Per-Ake Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In *SIGMOD Conference*, pages 61–71, 1986.

[6] Robert M. Bruckner, Beate List, and Josef Schiefer. Striving towards near real-time data integration for data warehouses. In *DaWaK*, pages 317–326, 2002.

[7] George L. Cain. *Introduction to General Topology*. Addison-Wesley, 2001.

[8] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1):65–74, 1997.

[9] Latha S. Colby, Akira Kawaguchi, Daniel F. Lieuwen, Inderpal Singh Mumick, and Kenneth A. Ross. Supporting multiple view maintenance policies. In *SIGMOD Conference*, pages 405–416, 1997.

[10] Ben Kao, Kamyiu Lam, Brad Adelberg, Reynold Cheng, and Tony S. H. Lee. Updates and view maintenance in soft real-time database systems. In *CIKM*, pages 300–307, 1999.

[11] Ben Kao, Kamyiu Lam, Brad Adelberg, Reynold Cheng, and Tony S. H. Lee. Maintaining temporal consistency of discrete objects in soft real-time database systems. *IEEE Transactions on Computers*, 52(3):373–389, 2003.

[12] Namgyu Kim and Songchun Moon. Concurrent view maintenance scheme for soft real-time data warehouse systems. *Journal of Information Science and Engineering*, 23(3):725–741, 2007.

[13] Wilburt Labio, Yue Zhuge, Janet L. Wiener, Himanshu Gupta, Hector Garcia-Molina, and Jennifer Widom. The whips prototype for data warehouse creation and maintenance. In *SIGMOD Conference*, pages 557–559, 1997.

[14] Edmond Lau and Samuel Madden. An integrated approach to recovery and high availability in an updatable, distributed data warehouse. In *VLDB*, pages 703–714, 2006.

[15] Gang Luo. Efficient detection of empty-result queries. In *VLDB*, pages 1015–1025, 2006.

[16] Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael Watzke. Transaction reordering with application to synchronized scans. In *DOLAP*, pages 17–24, 2008.

[17] Gang Luo and Philip S. Yu. Content-based filtering for efficient online materialized view maintenance. In *CIKM*, pages 163–172, 2008.

[18] Dallan Quass and Jennifer Widom. On-line warehouse view maintenance. In *SIGMOD Conference*, pages 393–404, 1997.

[19] Raghu Ramakrishnan, Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Efficient incremental evaluation of queries with aggregation. In *SLP*, pages 204–218, 1994.

[20] Ricardo Jorge Santos and Jorge Bernardino. Real-time data warehouse loading methodology. In *IDEAS*, pages 49–58, 2008.

[21] Maik Thiele, Ulrike Fischer, and Wolfgang Lehner. Partition-based workload scheduling in living data warehouse environments. In *DOLAP*, pages 57–64, 2007.

[22] Maik Thiele and Wolfgang Lehner. Evaluation of load scheduling strategies for real-time data warehouse environments. In *BIRTE*, 2009.

[23] Christian Thomsen, Torben Bach Pedersen, and Wolfgang Lehner. RiTE: Providing on-demand data for right-time data warehousing. In *ICDE*, pages 456–465, 2008.

[24] Jennifer Widom. Research problems in data warehousing. In *CIKM*, pages 25–30, 1995.

[25] Jennifer Widom. Review - an overview of data warehousing and olap technology. *ACM SIGMOD Digital Review*, 1, 1999.

[26] Ming-Chuan Wu and Alejandro P. Buchmann. Research issues in data warehousing. In *BTW*, pages 61–82, 1997.

[27] Yue Zhuge, Hector Garcia-Molina, Joachim Hammer, and Jennifer Widom. View maintenance in a warehousing environment. In *SIGMOD Conference*, pages 316–327, 1995.

[28] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. The strobe algorithms for multi-source warehouse consistency. In *PDIS*, pages 146–157, 1996.

[29] Yue Zhuge, Hector Garcia-Molina, and Janet L. Wiener. Consistency algorithms for multi-source warehouse view maintenance. *Distributed and Parallel Databases*, 6(1):7–40, 1998.