

# The Lord of the Rings: Efficient Maintenance of Views at Data Warehouses<sup>\*</sup>

D. Agrawal<sup>1</sup>, A. El Abbadi<sup>1</sup>, A. Mostéfaoui<sup>2</sup>, M. Raynal<sup>2</sup>, and M. Roy<sup>2</sup>

<sup>1</sup> Department of Computer Science, University of California,  
Santa Barbara, CA 93111, USA

<sup>2</sup> IRISA, Université de Rennes I  
Campus de Beaulieu, 35042 Rennes, France

**Abstract.** Data warehouses have become extremely important to support online analytical processing (OLAP) queries in databases. Since the data view that is obtained at a data warehouse is derived from multiple data sources that are continuously updated, keeping a data warehouse up-to-date becomes a crucial problem. An approach referred to as the *incremental view maintenance* is widely used. Unfortunately, a precise and formal definition of view maintenance (which can actually be seen as a distributed computation problem) does not exist. This paper develops a formal model for maintaining views at data warehouses in a distributed asynchronous system. We start by formulating the view maintenance problem in terms of abstract update and data integration operations and state the notions of correctness associated with data warehouse views. We then present a basic protocol and establish its proof of correctness. Finally, we present an efficient version of the proposed protocol by incorporating several optimizations. So, this paper is mainly concerned with basic principles of distributed computing and their use to solve database related problems.

## 1 Introduction

*Context of the study:* As the capability for storing data increases, there is a greater need for developing analysis tools that provide aggregation and summarization capabilities. In the context of databases, a new class of query processing referred to as OLAP (*OnLine Analytical Processing*) has emerged for aggregating and summarizing vast amount of data. A typical example of an OLAP query is to identify total sales for a product type in a geographic region over a specific period of time. As this example illustrates, since OLAP queries need to scan the data for aggregation, executing them on traditional DBMSs will adversely impact the OLTP (*OnLine Transaction Processing*) workload, which typically consists of short update transactions. Furthermore, enterprise wide data is typically stored on multiple database repositories. Such geographical distribution leads to expensive distributed processing for OLAP queries which demand interactive response

---

<sup>\*</sup> This work was supported in part by an NSF/INRIA grant INT-0095527.

times. Unlike the OLTP workloads where up-to-date data is mandatory, OLAP queries typically involve historical data which need not be absolutely up-to-date. This property of OLAP queries has resulted in the emergence of the notion of *data warehouses*.

Data warehouses exploit the lack of *strict currency* requirement of OLAP queries by creating a redundant *view* of the data that is derived from the data repositories or multiple DBMSs comprising an enterprise. In the context of relational DBMSs, this view is specified in terms of an SPJ-expression (select-project-join) over the multiple relations contained in the data repositories. Once the view is created, a critical issue is how to keep the view both consistent as well as approximately current with respect to the changes that occur in the data sources from which the view is derived. One possibility is to re-compute the entire view periodically, e.g., daily, weekly, or monthly. However, such re-computation is unnecessary and wasteful [8]. Instead the view can be maintained incrementally. That is, a view is created initially. Subsequently, whenever changes occur at the data sources, a view maintenance algorithm is initiated to determine the incremental changes (both augmentation as well as removal) to the base view. Numerous protocols have been proposed to find efficient solution to the incremental maintenance problem (e.g., [2, 6, 9, 14, 18]). However, most of these approaches are database centric and fail to clearly formalize the problem resulting in subtle inefficiencies as well as ad-hoc restrictions on the system.

*Content of the paper:* The aim of this paper is to use basic principles of distributed computing to address and solve the view maintenance problem. In that sense, the paper lies at a “borderline” where distributed computing and databases cross. So, since view maintenance involves multiple data sources in an asynchronous environment, we formulate this problem as a distributed computation problem using notions from distributed algorithms and distributed computing. Our goal is to obtain a formal understanding of the system components of a data warehouse architecture much in the same way as concurrency control theory does that for database management systems.

To attain this goal, the paper starts by providing a precise definition of the notion of views in terms of abstract operations with well-defined properties, stating the safety and liveness requirements that provide an abstract and well-defined specification of the problem. Then, the paper introduces an algorithm along with its proof of correctness. Note that very few attempts have been made in the data warehouse literature to define and formally prove the correctness of view maintenance algorithms in asynchronous environments. Furthermore, most data warehousing system architectures require the FIFO property for communication channels between the warehouse and the data sources. The proposed algorithm imposes no such restriction. It is simple, powerful, proved correct and can be generalized to address multi-term view. Hence, the results presented in the paper are (1) a precise (and abstract) definition of the data warehouse view maintenance problem, and (2) a new simple and efficient view maintenance protocol with its proof.

*Roadmap:* The paper is made up of six sections. Section 2 introduces the underlying data/query model. Section 3 provides a specification of the data view management problem. Then, Section 4 presents and proves correct a data view management protocol. Section 5 presents the case of multi-term views. Finally, Section 6 provides a few concluding remarks.

## 2 The Model

The system model we consider is based on the data warehouse architecture [2, 17, 18]. The system consists of two types of components: the data sources and the data warehouse. We assume that there are  $n$  distinct data sources, which can be modeled as data objects denoted  $x_1, \dots, x_n$ . These data objects are updated independently of each other as a result of transactions. Note that in this paper we consider transactions that are restricted to a single data source. In [17], Zhuge *et al.* first develop this model and then generalize it to settings where transactions access multiple data sources on different sites. Our future work will expand our approach to include this generalized model. From a database perspective,  $x_i$  can be correlated with a relation in a DBMS. The data warehouse can be modeled as a function  $F$  whose current value depends on the current state of all data objects  $x_i$ .

*Data Sources:* A data object  $x_i$  can be updated by transactions issued by its clients. There are two types of update operations, denoted  $\oplus$  and  $\ominus$ . These operations are such that:

- $\oplus$  and  $\ominus$  are inverse of each other:  $\oplus^{-1} \equiv \ominus$  and  $\ominus^{-1} \equiv \oplus$  (this means that  $(a \oplus b = c) \Leftrightarrow (a = c \ominus b)$ ). Let  $\perp$  be the value such that  $\forall a : (a \oplus \perp = a \ominus \perp = a)$ .
- $\oplus$  is associative (this means that  $((a \oplus b) \oplus c) = (a \oplus (b \oplus c))$ ).

In database terminology, the  $\oplus$  operation can be viewed as an insertion of a tuple in a relation whereas  $\ominus$  can be viewed as a deletion of a tuple from the relation. A data object  $x_i$  is accessed by one operation at a time. Let  $t = 1, 2, \dots$  represents the time progression. The local history  $h_i$  of  $x_i$  is the sequence of values taken by  $x_i$  at different time instants. It is denoted  $(x_i^{[t]})_{t=1,2,\dots}$ . Let us note that if  $x_i$  is not updated between  $t1$  and  $t2$ , we have  $x_i^{[t1]} = x_i^{[t1+1]} = \dots = x_i^{[t2]}$  <sup>(3)</sup>. It is assumed that  $\forall i : \forall t : x_i^{[t]} \neq \perp$ . Note that  $x_i$  is sometimes used to denote the data variable, and sometimes to denote its value. This is done according to the context and when there is no ambiguity.

*Data Warehouse:* The data warehouse (DWH) provides its clients (when they query it) with the value of a function  $F$  defined on the data objects  $x_1, \dots, x_n$ . It is assumed that DWH processes one query at a time. This means that the

<sup>3</sup> This is sometimes called *stuttering*. The important point is that  $h_i$  includes all the updates of  $x_i$ .

execution of DWH can be modeled as a sequence of values taken by  $F$ , namely the sequence of values returned by the queries. Let us now define the type of the functions  $F$  that will be supported by DWH. Let  $\otimes$  be an operation that is:

- associative ( $a \otimes (b \otimes c) = (a \otimes b) \otimes c$ ),
- commutative ( $a \otimes b = b \otimes a$ ), and
- distributive over  $\oplus$  and  $\ominus$  ( $a \otimes (b \text{ op } c) = (a \otimes b) \text{ op } (a \otimes c)$ ) where  $\text{op}$  is either  $\oplus$  or  $\ominus$ . Let us note that we consequently have  $a \otimes \perp = \perp$  <sup>(4)</sup>.

In relational database terminology, the  $\otimes$  can be viewed as a join operation among multiple relations. At an abstract level, the join operation corresponds to a cross-product of tuples in two relations. However, it is often used with a conditional operator (e.g., equality) so that the number of tuples in the output is limited. Let a *term* be the product ( $\otimes$ ) of a subset of the data objects, e.g.,  $x_2 \otimes x_5 \otimes x_6$  is a term. The functions  $F$  we consider are sums ( $\oplus$ ) of such terms. Here is an example of such a function on six data objects  $F(x_1, \dots, x_6) = (x_2 \otimes x_5 \otimes x_6) \oplus (x_1 \otimes x_2 \otimes x_4 \otimes x_5) \oplus (x_3 \otimes x_6)$ . Typically, the data warehouse function  $F$  involves a single term which is a join operation on  $n$  relations.

### 3 The View Management Problem

The *View Management* problem is concerned with the queries issued by the clients of DWH. Let a *view* be the result returned by a query. Intuitively, the views have to be consistent with respect to the data values, and, as time progresses, they have to incorporate the updates that have been applied to the data. More formally, we express the View Management problem as a set of three properties we name **Validity**, **Order Consistency** and **Up-to-dateness**. The first two are safety properties, while the third is a liveness property. They are defined as follows.

- **Validity.** A view  $f$  obtained by a query is such that  $f = F(x_1^{[t_1]}, \dots, x_n^{[t_n]})$ , where  $\forall i \in [1..n]$ ,  $t_i \in [1, 2, \dots]$  (the value  $x_i^{[t_i]}$  appears in the history of the data object  $x_i$ ).  
Validity states that a query always returns a meaningful value.
- **Order Consistency.** Let  $q_1$  (resp.  $q_2$ ) be a query that returns the value  $f1 = F(x_1^{[t_{11}]}, \dots, x_n^{[t_{1n}]})$  (resp. the value  $f2 = F(x_1^{[t_{21}]}, \dots, x_n^{[t_{2n}]})$ ). If  $f1$  appears before  $f2$  in the history of DWH, then  $\forall i \in [1..n]$ ,  $t_{1i} \leq t_{2i}$ .  
Order consistency means that the values returned by a sequence of queries are consistent with respect the local history of each data object  $x_i$ .
- **Up-to-dateness.** Let us assume an infinite sequence of queries.  $\forall i \in [1..n]$ ,  $\forall t \in [1, 2, \dots]$ , there is a query that returns  $f = F(\dots, x_i^{[t']}, \dots)$  with  $t' \geq t$ .  
Up-to-dateness states that, provided there are enough queries, any update of a data object  $x_i$  will be used to compute a view, or will be overwritten by a more recent update.

---

<sup>4</sup> Let us remark that, due to the properties we assume, the operations  $\oplus$  and  $\otimes$  define a *ring* mathematical structure.

Specifying a problem as a set of abstract properties is important for several reasons. First, it does not attach the problem to a specific context or a specific set of mechanisms that allow to solve it. Then, it allows the protocol designer to provide a formal proof of its solution: showing that a particular protocol solves the view maintenance problem requires to establish that any of its execution satisfies the **Validity**, **Order Consistency** and **Up-to-dateness** properties.

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, different notions of consistency of the view have been identified in [11, 18]. *Convergence* where the updates are eventually incorporated into the materialized view. *Strong consistency* where the order of state transformations of the view at the data warehouse corresponds to the order of the state transformations at the data sources. *Complete consistency* where every state of the data sources is reflected as a distinct state at the data warehouse, and the ordering constraints among the state transformations at the data sources are preserved at the data warehouse. These notions are different from the ones we have proposed. Our proposed notions are based on when the state of the warehouse is queried. In contrast, the database notions are based on the state of the data warehouse. We feel that the former is more appropriate since it imposes weaker constraints on the maintenance algorithm<sup>5</sup>.

## 4 The Case of a Simple View (Single Term)

This section presents a protocol that solves the view maintenance problem when the function  $F$  is a single term. So, without loss of generality, this section considers that  $F(x_1, \dots, x_n) = x_1 \otimes \dots \otimes x_n$ . (The case where  $F$  includes several terms is considered in [3].)

### 4.1 Preliminaries

We consider a distributed system in which each data object  $x_i$  is located on a distinct node. So, from now on,  $x_i$  is used to denote both a data object and the node where it is located. Moreover, the DWH entity is assumed to be located on another node. The nodes communicate through reliable (not necessarily FIFO) channels. The nodes are assumed to be reliable. We assume an asynchronous system, i.e., there is no bound on processing time and message transfer delays.

There are two extreme solutions. One is to have a DWH node that is “memoryless”: each time it is queried, it asks the data sources and computes the value of  $F$  according to the values it gets. The other is to maintain a copy of every data  $x_i$  at the DWH node. These solutions are highly inefficient, in terms of communication (former solution) or storage (latter solution). That is why an “*incremental*” data view computation has been proposed by several authors [2, 6, 17, 8, 14, 18,

---

<sup>5</sup> In the presence of an infinite number of queries, it is possible to show that the proposed specification implies strong consistency.

11, 15]. In general, these approaches maintain at the DWH node the last computed value of  $F$ , and only compute the corresponding  $\Delta_F$  when a data object  $x_i$  is updated to a new value  $x_i \oplus \delta_i$ . More precisely, let  $F_0 = F(x_1, \dots, x_i, \dots, x_n)$ . The idea is for DWH to only compute (with the help of the data nodes) the value  $\Delta_F$  such that

$$F(x_1, \dots, x_i \oplus \delta_i, \dots, x_n) = F_0 \oplus \Delta_F.$$

The main problem that needs to be addressed concerns the view maintenance in the presence of concurrent updates to different data objects. If two data objects, e.g.,  $x_i$  and  $x_j$ , are concurrently updated to  $x_i \oplus \delta_i$  and  $x_j \oplus \delta_j$ , the values  $\Delta_F$  that are computed must ensure that the values obtained by the queries at DWH are always valid, order consistent and up-to-date.

## 4.2 The Basic Protocol

This section introduces a *basic* (abstract) protocol solving the view maintenance problem. The protocol is abstract [10] in the sense it uses a token that perpetually moves on a directed ring made up of the data nodes. The ring assumption and the perpetual motion of the token are only used to provide a simple presentation of the protocol principles. So, after having presented and proved the protocol expressed in such an abstract way, the simplifying assumptions (ring and perpetual motion of the token) are eliminated to provide a (concrete) more efficient protocol (Section 4.4) that does not require data nodes to know each other, and that is quiescent (i.e., there is no activity when there are neither data update nor queries).

So, let us assume that the data nodes  $x_1, \dots, x_n$  define a ring. For any node  $x_i$ , the function `next_data` gives the data node that follows  $x_i$  on the ring.

*The “no concurrent updates” case:* Let us consider the case where a single data object  $x_i$  is updated to  $x_i \oplus \delta_i$ . As the value  $F(x_1, \dots, x_i, \dots, x_n)$  is kept by DWH, the aim is to compute  $\Delta_F$  such that  $F(x_1, \dots, x_i \oplus \delta_i, \dots, x_n) = F(x_1, \dots, x_i, \dots, x_n) \oplus \Delta_F$ , and to supply the DWH with it.

A simple idea is the following<sup>6</sup>. Let the token start from  $x_i$  and carry the value  $\delta_i$ . The token goes to  $x_j = \text{next\_data}(x_i)$ . When it receives the token, the node  $x_j$  computes  $x_j \otimes \delta_i$ , and sends the token with this new value to  $x_k = \text{next\_data}(x_j)$ , and so on. When, the token returns to  $x_i$ , it carries the value  $x_1 \otimes \dots \otimes x_{i-1} \otimes \delta_i \otimes x_{i+1} \otimes \dots \otimes x_n$ , which (as shown by the proof) is  $\Delta_F$ . So, when it receives the token, the node  $x_i$  has only to send the token value to DWH.

*The “concurrent updates” case:* Allowing concurrent computations of the  $\Delta_F$  increments corresponding to concurrent updates (e.g.,  $x_i + \delta_i$  and  $x_j + \delta_j$ ) poses difficult problems. In order to compute the correct  $\Delta_F$ , each of the following terms  $x_i \otimes \delta_j$ ,  $x_j \otimes \delta_i$ , and  $\delta_i \otimes \delta_j$ , must be taken into account exactly once. The concurrency of the computations induced by the updates can produce a

<sup>6</sup> This idea has been used in several protocols, with an implementation that is not ring/token-based.

$\Delta_F$  including twice the term  $\delta_i \otimes \delta_j$  (the situation is worse if there are more concurrent updates). This is called an *error term*, and a main challenge for the protocols that allow multiple  $\Delta_F$  computations due to concurrent updates is to appropriately manage the error terms.

Here, we solve the problem posed by concurrent updates with a pipelining technique. More precisely, the token is an array  $token[1..n]$  with one entry per data. The entry  $token[i]$  plays the role of the simple token of the “no concurrent updates” case as far as  $x_i$  updates are concerned. As we will see, this is particularly efficient.

The previous idea is implemented as follows. The token (initially equal to  $[\perp, \dots, \perp]$ ) and a sequence number generator  $sn$  (initialized to 0) are initially placed on a data node, and then move perpetually on the ring. The  $sn$  variable is used to order the successive values of  $\Delta_F$  that are computed; they are sent by the data nodes to DWH and will be denoted  $\Delta_F^1, \Delta_F^2, \dots$ . Initially DWH keeps the value of the initial view, namely  $f_0 = F(v_1, \dots, v_n)$  (where  $v_i$  is the initial value of  $x_i$ ). Then, it proceeds incrementally: when it receives  $\Delta_F^x$ , it computes  $f_x = f_{x-1} \oplus \Delta_F^x$  (the sequence numbers allow it to consider the received  $\Delta_F$  increments in the correct order).

*A single term protocol:* The protocol is described in Figure 1. It is made up of three parts. Two describe the behavior of a node  $x_i$ : what it does when it receives the pair  $(token, sn)$ , and what it does when  $x_i$  is updated. The third part describes the behavior of DWH when it receives a  $\Delta_F$  value. The lines 14-17 correspond to the previous discussion on the incremental behavior of DWH. Its local variable  $next\_sn$  (initialized to 0) allows it to correctly order the  $\Delta_F$  increments.

In addition to  $x_i$ , the manager of  $x_i$  maintains a local variable  $\Delta_i$  (initialized to  $\perp$ ). This variable is used to record updates of  $x_i$  between two consecutive visits of the token. More precisely, each time the token leaves the  $x_i$  node (line 10),  $\Delta_i$  is reset to  $\perp$  (line 9). Then,  $\Delta_i$  aggregates the updates of  $x_i$  (line 13) until the next visit of the token. (When we consider an update of  $x_i$  (lines 11-13), we only consider the case of the  $\oplus$  operation, as the  $\ominus$  operation can be expressed from  $\oplus$  as it is its inverse.)

The behavior of the node  $x_i$  when it receives the pair  $(token, sn)$  can be decomposed into two parts. In the first part (lines 2-5), the node provides DWH with the next  $\Delta_F$  value, if necessary. After a full revolution of the token from  $x_i$  to itself, the  $token[i]$  entry contains the increment  $\Delta_F$  corresponding to the previous  $\Delta_i$  multiplied by the value of each other  $x_j$  when the token visited it. If  $\Delta_i$  was null (a  $\perp$  value), then  $\Delta_F$  is null and no message is sent to DWH. Basically, this part (together with the perpetual motion of the token, line 10) ensures the **Up-to-dateness** property (liveness). The second part (lines 6-10) concerns the token management, and basically ensures the **Validity** and **Order Consistency** properties (safety). In this part, the  $x_i$  node updates each token entry: (1)  $token[i]$  at line 6 which corresponds to the new updates, and (2)  $token[j]$ ,  $j \neq i$ , at lines 7-8 which corresponds to the effects of  $x_i$  on updates of other nodes. Let us consider the case where a single data object  $x_k$  is updated (hence  $\Delta_k \neq \perp$ , while

```

(1) when  $token\_sn(token, sn)$  is received by  $x_i$ :
(2)   let  $\Delta_F = token[i]$ ;
(3)   if  $(\Delta_F \neq \perp)$  then  $sn \leftarrow sn + 1$ ;
(4)                                     send  $incr(\Delta_F, sn)$  to DWH
(5)   endif;
(6)    $token[i] \leftarrow \Delta_i$ ;
(7)    $\forall j \in [1..n], j \neq i$  :
(8)       do  $token[j] \leftarrow (token[j] \otimes (x_i \ominus \Delta_i))$  enddo;
(9)    $\Delta_i \leftarrow \perp$ ;
(10)  send  $token\_sn(token, sn)$  to  $next\_data$ 

(11) when  $update(\delta_i)$  is received by  $x_i$ :
(12)   $x_i \leftarrow x_i \oplus \delta_i$ ;
(13)   $\Delta_i \leftarrow \Delta_i \oplus \delta_i$ 

(14) when  $incr(\Delta_F, sn)$  is received by DWH:
(15)  wait  $(next\_sn = sn)$ ;
(16)   $f \leftarrow f \oplus \Delta_F$ ;
(17)   $next\_sn \leftarrow next\_sn + 1$ 

```

**Fig. 1.** An Basic Single Term View Maintenance Protocol

$\forall j \neq k : \Delta_j = \perp$ ). Then, starting from  $x_k$ , we have  $\forall j \neq k : token[j] = \perp$ , and  $token[k] = \Delta_k$ . When the token moves along the ring, we have:  $\forall j \neq k : token[j]$  remains equal to  $\perp$ , while  $token[k]$  is updated at line 8 by each visited node as follows  $token[k] \leftarrow (token[k] \otimes x_i)$ . Consequently, when the token returns to the node  $x_k$ , we have  $token[k] = x_1 \otimes \dots \otimes x_{k-1} \otimes \Delta_k \otimes x_{k+1} \otimes x_n$ , and  $x_k$  sends this value to DWH as the next  $\Delta_F$ .

*Remark.* The protocol aggregates all the updates  $(\delta_{i1}, \delta_{i2}, \dots)$  on  $x_i$  that occur between two consecutive visits of the token, and considers them as a single update, namely  $\Delta_i$ . If we do not want to allow the gathering of consecutive updates  $(\delta_i)$  into a “big” update  $(\Delta_i)$ , each data node  $x_i$  can use a list recording its successive  $\delta_i$  updates, and define  $\Delta_i$  at line 13 as the first  $\delta_i$  of the list not yet considered. *End of remark.*

*Latency of the basic protocol:* We evaluate here the time latency of the protocol as the time that elapses between an update to the time when its effects are incorporated at DWH. We assume that each message takes one time unit, and that processing takes no time. Let us consider the two following extreme cases.

Case 1: a single data object has been updated. In that case, the token has to arrive at the corresponding node, then the token has to complete a turn of the ring. Moreover, an *incr* message has then to be sent. This means that, in the worst case,  $2n$  time units are required.

Case 2: each data object has been updated. It easy to see that in that case  $2n$  time units are also required. This means that, in that case, the average cost of an update is 2 time units.

Section 4.4 presents improvements that provide an efficient version of the protocol that reduces the cost to  $n$  time units in Case (1), while not increasing it in Case (2).

### 4.3 Correctness Proof

**Theorem 1.** *The protocol described in Figure 1 solves the view maintenance problem.*

**Proof** We have to show that the protocol satisfies the Validity, Order Consistency and Up-to-dateness properties described in Section 3. The proof uses the following notation:  $\forall i \in [1..n] : \forall k \in \mathbf{Z} : x_{i+kn} \equiv x_i$ .

Let us define a virtual time notion as follows: the time progress is measured as the number of steps performed by the token. Hence, at  $t = 1$  the token is in  $x_1$ 's possession, at  $t = 2$ , the token is held by  $x_2$ , at  $t$  the token goes for the  $((t \div n) + 1)th$  time to the site  $x_{(t-1) \bmod n + 1} \equiv x_t$ .

Let us rewrite  $(x_i^{[t]})_{t \geq 0}$  as the local history of  $x_i$  sampled every  $n$  time units ( $n$  is the time duration for a complete tour of the ring by the token). This means that the semantics of  $x_i^{[t]}$  is as follows: for any  $k \in \mathbf{N}$ ,  $x_i^{[i+kn]} = x_i^{[i+kn+1]} = \dots = x_i^{[i+kn+n-1]}$ , meaning that, from an external observer point of view,  $x_i$  is seen as constant during a tour, and  $x_i^{[i+kn]}$  includes all the updates made on  $x_i$  during the  $k$ th tour of the ring (these updates are locally kept at the node  $x_i$  in  $\Delta_i$ ). We also define for convenience of the proof  $x_i^{[t]} = x_i^{[0]}, \forall t < 0$ .

The proof basically follows from the following claim:

*Claim:* At step  $t$ ,

1. The value computed on the DWH based on all messages for steps up to  $t - 1$  satisfies:  $f^{[t-1]} = (x_1 \otimes x_2 \otimes \dots \otimes x_n)^{[t-1-n]}$ .
2. The *token* value received by the data manager of  $x_i$  (with  $i = (t-1) \bmod n + 1$ ) is the array:

$$\begin{aligned} token[i] &= (x_i^{[t-n]} \ominus x_i^{[t-2n]}) \otimes (x_1 \otimes \dots \otimes x_{i-1} \otimes x_{i+1} \otimes \dots \otimes x_n)^{[t-n]} \\ token[i+1] &= (x_{i+1}^{[t-n+1]} \ominus x_{i+1}^{[t-2n+1]}) \otimes (x_1 \otimes \dots \otimes x_{i-1} \otimes x_{i+2} \otimes \dots \otimes x_n)^{[t-n+1]} \\ &\dots \end{aligned}$$

$$token[i-2] = (x_{i-2}^{[t-2]} \ominus x_{i-2}^{[t-n-2]}) \otimes (x_{i-1})^{[t-2]}$$

$$token[i-1] = (x_{i-1}^{[t-1]} \ominus x_{i-1}^{[t-n-1]}).$$

Using this claim, the proposed protocol trivially satisfies Validity, since any query returns an  $f^{[t]} = \bigotimes_{1 \leq i \leq n} x_i^{[t]}$  for some  $t$ .

Order Consistency is ensured because the DWH always updates its state from  $f^{[t]}$  to  $f^{[t']}$ , with  $t' \geq t$ . This means that if DWH processes the query  $q_1$  before the query  $q_2$  and if  $q_1$  returns  $f^{[t]} = (\bigotimes_{1 \leq i \leq n} x_i^{[t]})$ , then  $q_2$  will return  $f^{[t']} = (\bigotimes_{1 \leq i \leq n} x_i^{[t']})$  such that  $t' \geq t$ .

For Up-to-dateness, let us observe that, the token turning endlessly on the ring, any update is taken into account in the token during one of its turns on the ring, say during tour  $k$ . Then, this modification will be committed on DWH at the latest in the value  $f^{[(k+1)*n+1]}$ , using the first item of the claim.

*Proof of the claim:*

• *Initialization.* The token is initialized to  $[\perp, \dots, \perp]$ . Since  $\perp$  is the zero of the  $\oplus$  operation, during the first tour, the value on DWH remains unchanged. As it is initialized to  $\bigotimes_{1 \leq i \leq n} x_i^{[0]}$ , the first item of the claim holds for  $1 \leq t \leq n$ . The second item is satisfied for  $t = 1$  because there are no changes for  $x_i$  before  $t = 0$ .

• *Induction.* Let us suppose that the two items are satisfied until some  $t > 0$ , and let us show that they hold for  $t + 1$ . Let  $i = (t - 1) \bmod n + 1$ .

1. There are two cases corresponding to the test at line 3. If  $token[i]$  is equal to  $\perp$ , it means that  $p_i$  had no update to commit the previous time it got the token. Hence  $x_i$  was not modified at time  $t - n$ , and no update is needed on the DWH; therefore, the first item of the claim holds for  $t + 1$ .

If  $token[i]$  contains a value, the  $x_i$  data manager sends the value  $\Delta_F = token[i]$  to DWH (line 5) with an incremented sequence number. DWH adds it to its current view of the product at line 16. Due to the sequence number synchronization (line 15), and using the induction hypothesis on the token shape, the new value computed at line 16 is :

$$f' = f^{[t-1]} \oplus (x_i^{[t-n]} \ominus x_i^{[t-2n]}) \otimes (x_1 \otimes \dots \otimes x_{i-1} \otimes x_{i+1} \otimes \dots \otimes x_n)^{[t-n-1]}$$

and, by the induction hypothesis on  $f^{[t]}$  we get:

$$f' = (x_1 \otimes \dots \otimes x_n)^{[t-n-1]} \oplus (x_i^{[t-n]} \ominus x_i^{[t-2n]}) \otimes (x_1 \otimes \dots \otimes x_{i-1} \otimes x_{i+1} \otimes \dots \otimes x_n)^{[t-n]}.$$

But, for any  $x_j$  ( $j \neq i$ ) the values of  $x_j$  are equal at times  $t - n$  and  $t - n - 1$  (definition of  $x_i^{[t]}$ ), so the previous expression reduces to:

$$f' = ((x_1 \otimes \dots \otimes x_{i-1} \otimes x_{i+1} \otimes \dots \otimes x_n)^{[t-n]}) \otimes (x_i^{[t-n-1]} \oplus x_i^{[t-n]} \ominus x_i^{[t-2n]}).$$

Since  $x_i^{[t-n-1]} = x_i^{[t-2n]}$ , we conclude that the new value  $f'$  computed by DWH is equal to  $f^{[t]}$ , which proves the first item of the claim.

2. Then, the data manager multiplies each other term by its old value of  $x_i$  (lines 7-8), that is the value that does correspond to dates  $t - 1, t - 2, \dots, t - n$ . Hence, the token is now made up of the following values:

$$token[i + 1] = (x_{i+1}^{[t-n+1]} \ominus x_i^{[t-2n+1]}) \otimes (x_1 \otimes \dots \otimes x_i \otimes x_{i+2} \otimes \dots \otimes x_n)^{[t-n+1]}$$

...

$$token[i - 2] = (x_{i-2}^{[t-2]} \ominus x_{i-2}^{[t-n-2]}) \otimes (x_{i-1} \otimes x_i)^{[t-2]}$$

$$token[i - 1] = (x_{i-1}^{[t-1]} \ominus x_{i-1}^{[t-n-1]}) \otimes (x_i)^{[t-1]}.$$

Finally, the data manager of  $x_i$  updates  $token[i]$  to  $(x_i^{[t]} \ominus x_i^{[t-n]}) = \Delta_i^{[t]}$  (line 6) before sending the token to  $x_{t+1}$  (that is  $x_{i+1}$ ). Since the time increases one tick at each token move, it follows that  $x_{t+1}$  will get a token of the same shape, thus proving the second part of the claim. *End of the proof of the claim.*

□ *Theorem 1*

#### 4.4 An Efficient Protocol

This section presents a version of the basic protocol in which (1) the data nodes are not required to know each other, and (2) each query/update gives rise to a finite number of control messages (this means that the protocol eventually stops sending message if there is no more query or update<sup>7</sup>).

So the goal is to suppress (1) the ring and (2) the perpetual motion of the token. Issue (1) can easily be realized by structuring the system as a star whose center is the DWH node: each data object  $x_i$  sends messages only to (and receives only from) DWH. (Interestingly, this makes useless the sequence numbers used in Figure 1.) Issue (2) deals also with efficiency. There are several possible approaches to address it. One is for a data node  $x_i$  to query the DWH node to get the token when  $x_i$  has been updated. According to the quality of service desired for the up-to-dateness of  $F$ , the data nodes can require the token each time their data is updated, or only after some “quantity” of updates have been done (“data update-driven” refreshing of  $F$ ). Another approach could be for the DWH node to entail a token motion periodically or each time the value of  $F$  is queried by its clients (“DWH-driven” refreshing of  $F$ ).

Here we describe a protocol (Figure 2) that implements the data-driven update approach. When compared with the basic protocol, the main modifications concern DWH.

*Behavior of a data node:* (Lines 1-12) When  $x_i$  is updated, the data node requests the token by sending a message to DWH (line 10). When it receives the token (that now no longer carries a sequence number generator), the data node  $x_i$  updates appropriately the token entries (i.e., as in the basic protocol, see lines 1-4), and sends back the token to DWH (line 5).

*Behavior of the DWH node:* (Lines 13-36) DWH manages the following local variables and uses two functions:

- *pending*[1.. $n$ ] is a boolean array, such that *pending*[ $i$ ] = *true* means that  $x_i$  has required the token (in order the increment  $\Delta_F$  due to the update  $\Delta_i$  be taken into account).
- *holder* contains the identifier of the current token owner ( $\in [1..n]$ ). When no token is running, *holder* =  $-1$ .
- *succ*( $i$ ) (line 30) is a function that returns the data node that follows the node  $x_i$  on the “ring”.
- *next\_holder*( $i$ ) (lines 31-36), assumes that  $x_i$  was the previous token holder, and delivers the next token holder (according to the position on data nodes on the “ring”), or  $-1$  if the token has not been requested by a data node.

When DWH receives a request for the token from  $x_i$  (line 13), it creates a token and sends it back to  $x_i$  if there is no token (lines 14-16). Otherwise, it records the fact that  $x_i$  has required the token (line 17).

---

<sup>7</sup> The property for a protocol to eventually stop sending control messages when they are no more request (update/query) is sometimes called *quiescence* [5].

```

(1) when token_sn(token) is received by  $x_i$ :
(2)    $token[i] \leftarrow \Delta_i$ ;
(3)    $\forall j \in [1..n], j \neq i$  : do  $token[j] \leftarrow (token[j] \otimes (x_i \ominus \Delta_i))$  enddo;
(4)    $\Delta_i \leftarrow \perp$ ;
(5)   send token_sn(token) to DWH;
(6)    $sent\_request_i \leftarrow false$ 

(7) when update( $\delta_i$ ) is received by  $x_i$ :
(8)    $x_i \leftarrow x_i \oplus \delta_i$ ;
(9)    $\Delta_i \leftarrow \Delta_i \oplus \delta_i$ 
(10)  if  $(\neg sent\_request_i)$  then send request(i) to DWH;
(11)                                      $sent\_request_i \leftarrow true$ 
(12)  endif

(13) when request(i) is received by DWH:
(14)  if  $(holder = -1)$  then  $holder \leftarrow i$ ;
(15)                                      $token \leftarrow [\perp, \dots, \perp]$  % create a token %
(16)                                     send token_sn(token) to  $x_i$ 
(17)                                     else  $pending[i] \leftarrow true$ 
(18)  endif

(19) when token_sn(token) is received by DWH from  $x_i$ :
(20)  let  $\Delta_F = token[succ(i)]$ ;
(21)  if  $(\Delta_F \neq \perp)$  then  $f \leftarrow f \oplus \Delta_F$ ;
(22)                                      $token[succ(i)] \leftarrow \perp$ 
(23)  endif;
(24)  if  $(token \neq [\perp, \dots, \perp])$  then  $holder \leftarrow succ(i)$ 
(25)                                     else  $holder \leftarrow next\_holder(i)$ 
(26)  endif;
(27)  if  $(holder \neq -1)$  then  $pending[holder] \leftarrow false$ ;
(28)                                     send token_sn(token) to  $x_{holder}$ 
(29)  endif

(30)  function  $succ(i)$  returns  $((i \bmod n) + 1)$ 

(31)  function  $next\_holder(i)$ :
(32)     $y \leftarrow i$ ; %  $x_i$  was the previous token holder %
(33)    repeat  $y \leftarrow succ(i)$  until  $(y = i \vee pending[x])$  endrepeat;
(34)    if  $(y = i)$  then returns  $(-1)$ 
(35)    else returns  $(y)$ 
(36)  endif

```

**Fig. 2.** A Quiescent Single Term View Maintenance Protocol

When DWH receives the token from  $x_i$  (line 19), it first incrementally updates  $f$  with the appropriate value  $\Delta_F$  if necessary (lines 20-23), exactly as in the

basic protocol. Then, DWH determines the next holder of the token (lines 24-26). If  $token \neq [\perp, \dots, \perp]$ , then the token has to complete its current turn of the ring (line 24); otherwise, the token skips to a requesting data node (if there is no requesting data node we have  $holder = -1$ ). Finally, according to the value computed for  $holder$ , DWH sends the token to  $x_{holder}$  if  $holder \in [1..n]$ , or destroys it if  $holder = -1$ .

It is easy to see that the protocol is quiescent. The proof that it solves the view maintenance problem is similar to the proof of the basic protocol.

## 5 The General Case

This section addresses the case where the view  $F$  is composed of several terms, i.e.,  $F(x_1, \dots, x_n) = \bigoplus_x term_x$  where each  $term_x$  is a term. The following view will be used in the following as an example to illustrate the underlying principles of the proposed solution:

$$F(x_1, \dots, x_5) = (x_1 \otimes x_2 \otimes x_3 \otimes x_5) \oplus (x_1 \otimes x_2 \otimes x_4) \oplus (x_3 \otimes x_4).$$

When the view  $F$  includes a single term, as we have seen, the main problem that has to be solved is the management of concurrent updates. Basically, this was a *parallelism management* problem. The basic protocol (and its improvement) described in the previous section addressed this issue with an appropriate pipelining technique implemented by a token moving on a ring.

When the view is composed of several terms, the new problem that appears is the following. Considering the previous view  $F$  (the initial value of which is  $f_0$ ), let us look at the data  $x_4$ , and assume it is the only data that is updated, namely to  $x_4 \oplus \delta_4$ . We get:

$$F(x_1, x_2, x_3, x_4 \oplus \delta_4, x_5) = (x_1 \otimes x_2 \otimes x_3 \otimes x_5) \oplus (x_1 \otimes x_2 \otimes (x_4 \oplus \delta_4)) \oplus (x_3 \otimes (x_4 \oplus \delta_4)),$$

i.e., from an incremental computation point of view:

$$F(x_1, x_2, x_3, x_4 \oplus \delta_4, x_5) = f_0 \oplus (x_1 \otimes x_2 \otimes \delta_4) \oplus (x_3 \otimes \delta_4) = f_0 \oplus \Delta 0_F \oplus \Delta 1_F.$$

This means that, not only a single update of a data entails more than one  $\Delta_F$  computation (namely, here  $\Delta 0_F$  and  $\Delta 1_F$ ), but their results have to be *atomically added* to  $f_0$  (adding them separately would produce an intermediate value of  $f$  that does not correspond to a valid view with respect to the given specification). Basically, this is a *synchronization* problem (in some sense, this problem is dual with respect to the parallelism management problem due to the concurrent updates).

The solution we propose is the following. It actually generalizes the basic protocol described in Figure 1.

- First, a ring and a token are associated with each term of the view. Hence, a token-based protocol is associated with each term (if the view is made up of a single term, the protocol simplifies and becomes the basic protocol described in Figure 1).

- If a data object  $x_i$  belongs to several rings (i.e., it appears in several terms), its updates  $\Delta_i$  not yet taken into account in the computation of  $F$ , require all the corresponding tokens to be simultaneously present at  $x_i$  in order the corresponding  $\Delta_F$  increments (one for each term to which  $x_i$  belongs) be computed. In the previous example, both the token associated with the ring including  $x_1, x_2$  and  $x_4$  (second term) and the token associated with the ring including  $x_3$  and  $x_4$  (third term) have to be simultaneously present on the node  $x_4$  for  $\Delta_4$  to be taken into account.

Moreover, as previously, each token carries a sequence number generator to allow DWH to correctly add the  $\Delta_F$  increments to the current value  $f$ , i.e.:

- in the correct order for each ring considered separately, and
- at the same time (i.e., atomically) for the  $\Delta_F$  increments that come from different rings but are due to the same update  $\Delta_i$  of a data object  $x_i$ .

A protocol that follows these principles can be found in [3].

## 6 Concluding Remarks

This paper has developed a formal model for maintaining views at data warehouses in a distributed asynchronous system. The view maintenance problem associated with distributed data warehouses has been investigated primarily by the database community. In general, the focus has been directed towards an appropriate storage model for supporting fast OLAP queries as well as rapid integration of updates at the data sources into the data warehouse. There is relatively little emphasis to provide a formal definition of the view maintenance problem and formally develop a correct solution. In this paper, we provided a formulation of the view maintenance problem in terms of abstract update and data integration operations and defined the notions of correctness associated with data warehouse views. We also introduced a basic protocol and established its proof of correctness. Then, we presented an efficient version of the proposed protocol by incorporating several improvements. We also included the principles of the solution to a more general view formulation which currently does not have a similar semantics in database systems. However, this general framework could be particularly promising to integrate semi-structured data sources.

To conclude, we would like to point out that this paper is an outcome of interactions between researchers working in the areas of distributed computing and databases, respectively. We believe that such interactions are highly beneficial to both communities. Interestingly, when looking in the past, we can observe such beneficial influences. We cite two: the area of epidemic communication [7, 16] that first appeared in the distributed computing area and has then been successfully applied to databases (e.g., [4]), and the domain of atomic broadcast/multicast and group communications [1, 12, 13].

## References

1. Agrawal D., Alonso G., El Abbadi A. and Stanoi I., Exploiting Atomic Broadcast in Replicated Databases. *Proc. of the International Conference on Parallelism (EUROPAR)*, pp. 496-503, August 1997.
2. Agrawal D., El Abbadi A., Singh A. and Yurek T., Efficient Data View Maintenance Warehouses. *Proc. ACM SIGMOD*, pp. 417-427, 1997.
3. Agrawal A., El Abbadi A., Mostéfaoui A., Raynal R. and Roy M., The Lord of the Rings: Efficient Maintenance of Views at Data Warehouses. *IRISA Research Report #1441*, IRISA, Rennes, France. Available at: <http://www.irisa.fr/bibli/publi/pi/2002/1441/1441.html>.
4. Agrawal D., El Abbadi A. and Steinke R.C., Epidemic Algorithms in Replicated Databases. *Proc. ACM PODS*, pp. 161-172, 1997.
5. Aguilera M.K., Chen W. and Toueg S., On Quiescent Reliable Communication. *SIAM Journal of Computing*, 26(6):2040-2073, 2000.
6. Colby L.S., Griffin T., Libkin L., Mumick I.S. and Trickey H., Algorithms for Deferred View Maintenance. *Proc. ACM SIGMOD*, ACM Press, pp. 469-480, 1996.
7. Demers A. *et al.*, Epidemic Algorithms for Replicated Database Maintenance. *Proc. ACM PODC*, ACM Press, pp. 1-12, 1987.
8. Gupta A. and Mumick I.S., Maintenance of Materialized Views: Problems, Techniques and Applications. *IEEE Bulletin of the Technical Committee on Data Engineering*, 18(2):3-18, June 1995.
9. Gupta A., Mumick I.S. and Subramanian V.S., Maintaining Views Incrementally. *Proc. ACM SIGMOD*, pp. 157-166, 1993.
10. Héлары J.-M., Mostéfaoui A. and Raynal M., A General Scheme for Token and Tree-Based Distributed Mutual Exclusion Algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 5(11):1185-1196, 1994.
11. Hull R. and Zhou G., A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches. *Proc. ACM SIGMOD*, pp. 481-492, 1996.
12. Kemme B. and Alonso G., A New Approach to Developing and Implementing Eager Database Replication Protocols. *ACM Transactions on Database Systems*, 25(3):333-379, 2000.
13. Powell D. (Guest Editor). Special Issue on Group Communication. *Communications of the ACM*, 39(4):50-97, 1996.
14. Rundensteiner E.A., Koeller A. and Zhang X., Maintaining Data Warehouses over Changing Information Sources. *Communications of the ACM*, 43(6):57-62, 2000.
15. Stanoi I., Agrawal D. and El Abbadi A., Modeling and Maintaining Multi-View Data Warehouses. *Proc. 18th. Int. Conference on Conceptual Modeling*, Paris, France, pp. 161-175, 1999.
16. Wu G. T. and Bernstein A. J., Efficient Solutions to the Replicated Log and Dictionary Problems. *Proc. ACM PODC*, pp. 233-242, 1984.
17. Zhuge Y., Garcia-Molina H., Hammer J. and Widom J., View Maintenance in a Warehousing Environment. *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 316-327, 1995.
18. Zhuge Y., Garcia-Molina H. and Wiener J.L., The Strobe Algorithms for Multi-Source Warehouse Consistency. *Proc. Int. Conference on Parallel and Distributed Information Systems*, 1996.