# Efficient materialized view maintenance in mobile environments using interleaved Pull-Push algorithm

Noha Adly, Yousry Taha, and Magdy Nagi
*Computer Science & Automatic Control Dept.,Alexandria University, Egypt*
Nancy Samaan
*School of Information Technology and Eng., Ottawa University, Canada*

This paper presents a novel algorithm, *Interleaved Pull-Push*, for incrementally maintaining materialized views defined over information sources and stored in a Data Warehouse (DW), where both sources and DW could be mobile. In contrast to other approaches, the workload of updating the view is distributed among sources, thus alleviating the DW from receiving and handling unnecessary intermediate messages. The proposed algorithm ensures strong consistency of the view at the data warehouse in the presence of concurrent updates, while reducing the number of messages needed as well as the communication overhead. A simulation model based on the WHIPS architecture has been developed to analyse the performance of the proposed algorithm and to compare it with a previously proposed algorithm, Pure-Push, and the traditional Nested SWEEP algorithm. Several experiments have been conducted to quantify the costs and benefits of the proposed algorithm.

يقدم هذا البحث خوارزم جديد، يعتمد على الدفع والجذب، لتحديث شكل البيانات بالتزايد وذلك للبيانات المخزنة والمعرفة للعديد من مصادر المعلومات والمخزنة داخل مستودعات البيانات آخذا في الاعتبار إمكانية أن يكون كليهما متحرك. وبخلاف الأنظمة الأخرى فإن الخوارزم المقترح يوزع حمل تحديث شكل البيانات بين مصادر المعلومات وبالتالي يعفي مستودع البيانات من استقبال ومعالجة الرسائل الوسيطة الغير ضرورية. الخوارزم المقترح قام بتوفير التوافق القوي بين شكل البيانات داخل مستودعات البيانات في حالة التعديلات المتزامنة وفي نفس الوقت فهو يقلل من عدد الرسائل المطلوبة وحمل الاتصالات. وقد تم تنفيذ نظام محاكاة مبني على عمارة WHIPS لتحليل أداء الخوارزم المقترح ومقارنته بخوارزم آخر تم اقتراحه من قبل معتمد على الدفع فقط والخوارزم التقليدي Nested SWEEP. وقد قمنا بعمل عدة تجارب لقياس التكلفة والمكسب الناتج من الخوارزم المقترح.

**Keywords:** View maintenance, Data warehouse, Mobile environment

## 1. Introduction

With the huge growth in today's knowledge industry, data warehousing has become a well-established key technology for many applications spanning diverse domains such as business, science, and education.

Combining the technology of DWs and mobile computing can offer more flexibility, time saving and efficiency to current business applications. The introduction of the concept of mobility to the field of data warehousing plays a major key role to enhance the efficiency of these applications.

Several architectures for data warehousing in a mobile environment were suggested in [1]. Different degrees of mobility were introduced ranging from having only mobile clients, mobile sources, or mobile DWs at one end to a fully mobile environment at the other end in which clients, sources and the DW itself are all mobile.

One of the major problems in data warehousing is materialized view maintenance. Algorithms that compute changes to a view in response to changes in the base relations are called incremental view maintenance algorithms. Materialized view maintenance has been the focus of increasing research activities in recent years. Basic approaches for materialized view maintenance may be classified into two categories; self maintenance and query-compensation algorithms. In self-maintenance algorithms, [2–4], the view is maintained at the DW without accessing the sources. However, self-

maintenance is restricted to certain types of views and increases storage and maintenance complexity by storing additional information at the DW. On the other hand, query compensation algorithms focus primarily on the use of queries to sources. Query compensation algorithms, e.g., ECA, STROBE, SWEEP, and PSWEEP, can be found in [5-8].

Introducing mobility in data warehousing systems poses unique challenges [9, 10] to the above mentioned approaches. These challenges stem from the fact that these algorithms are based on sources sending their updates to the DW as they occur and from the assumption that the DW can query these sources at any time. This requires the DW to be connected during the whole process of calculating the view change. In mobile computing environments, however, sources and the DW are likely to be disconnected for long periods of time. In addition, these approaches relied on the assumption that the communication channels between the DW and sources are reliable and FIFO in nature; an assumption that obviously no longer holds for mobile environments.

Recently, several approaches for materialized view maintenance in mobile environments have been proposed [11–14]. Although these approaches were adaptable to the unique characteristics of the mobile environments, they assumed a single stationery database server, where the view change is calculated, and did not handle the case of distributed base relations. A detailed discussion of these algorithms can be found in [15].

In [16], a "Pure-Push" algorithm has been proposed to incrementally maintain materialized views stored in mobile DWs and defined over multiple mobile sources. The algorithm distributed the workload of the view update process between the DW and sources, tolerating periods of disconnection for both the sources and DW. Moreover, the cost of updating the view at the DW was reduced to $n$ messages, where $n$ is the number of sources contributing to the definition of the view.

This paper proposes a novel algorithm that requires only each source to contribute with two messages to perform the view updating process in response to update batches of $n$ sources. The proposed algorithm, termed as *Interleaved Pull-Push*, relies on having the DW initiate the update process by pulling updates, occurring during a certain period of time, from the sources via sending two request messages to only two sources. The two sources contacted by the DW respond by initiating two messages that propagate in the forward and reverse directions along a chain of sources contributing in the view definition. As the two messages propagate, each source appends its updates, to both messages, occurring during a certain period of time specified by the DW. Finally, the two messages reach the DW where they undergo a special join operation to calculate the view change.

In contrast to earlier approaches, the proposed algorithm distributes the workload between the sources and the DW and eliminates the need for compensation queries due to concurrent updates, which results in reducing the processing overhead for the mobile DW. Further, the number of messages needed to update the view is reduced to $2n$ and deferred updates are used rather than immediate updates, which is more realistic in mobile environment since it reduces the communication overhead. In addition, the DW and sources do not have to be connected all the time, and can alternate between the different modes of operation (disconnected, partially connected, fully connected). Finally, the algorithm does not require reliable communication channels supporting FIFO property, which is crucial for mobile systems. A simulation model based on the WHIPS introduced in [17,18] has been developed to analyze the performance of the proposed algorithm.

The remainder of this paper is organized as follows. Section 2 describes the system model. Section 3 presents an overview of the Pure-Push algorithm [16]. In section 4 the proposed Interleaved Pull-Push algorithm is described and a proof of its correctness is illustrated. A simulation model along with some experimental results and analysis are introduced in section 5. Finally section 6 concludes the paper.

## 2. System model

Consider a system consisting of $n$ information sources $S_1$, $S_2$,..., $S_n$ each of which could

be mobile. Each source $S_i$ contains a single base relation $R_i$, with the attributes $A_i = \{A_{i,1}, A_{i,2}, \ldots, A_{i,|A_i|}\}$. The data warehouse contains a materialized view $v = \Pi_{Av}(\sigma_p(R_1 \chi R_2 \chi R_3 \ldots\ldots R_n))$, where $P$ is a selection predicate and $A_v \subseteq U_i A_i$ is the set of attributes required in the view. It is also assumed that each source stores a copy of part of the view definition and that the sources and the DW have their clocks synchronized via approaches proposed in the literature (e.g. [19, 20]). In addition, each source has a reliable fixed storage that can be recovered in case of crashes.

The problem can be stated as giving a set of updates $\delta\Re=\{\delta R_1, \delta R_2, \ldots, \delta R_n\}$, where $\delta R_i$ is the set of updates of the relation $R_i$. It is required to update the materialized view $v$ at the data warehouse in presence of $\delta\Re$.

For each relation $R_i$, stored at source $S_i$, two time tags $cr_i$, $dr_i$ are used as in [13], $cr_i$ is a creation time tag, recording the creation time of the associated tuple in $R_i$, and $dr_i$ is a deletion time tag, marking the removal time of the associated tuple from $R_i$. Thus, $R_i$ has attributes $A_i = \{A_{i,1}, A_{i,2}, \ldots, A_{i,|A_i|}\} \cup \{cr_i\}$. In addition, for each relation $R_i$, a log, $\overline{R_i}$, that collects deleted tuples removed from $R_i$ is maintained, such that $\overline{R_i}$ has the attributes

$$\overline{A}_i = \{A_{i,1}, A_{i,2}, \ldots, A_{i,|A_i|}\} \cup \{cr_i\} \cup \{dr_i\}.$$

## 3. Pure-Push algorithm

The basic idea in the Pure-Push algorithm is to have the sources gather their updates in batches and send them in messages that propagate along a chain of sources. Each source computes part of the view change by joining the received message with its base relation. When the view change is finally calculated, the last source transmits the result to the DW, where the view is updated.

Denote the batch of updates occurring at source $S_i$ during the interval $]t^i_k, t^i_{k+1}]$ by $\delta R^i_{ub}(t^i_k, t^i_{k+1})$. Source $S_i$ selects source $S_j$ to contact and sends it the message $(\delta V, t^i_k, t^i_{k+1}, i)$ where $\delta V = \delta R^i_{ub}(t^i_k, t^i_{k+1})$. Upon receiving this message, source $S_j$ updates $\delta V$ such that $\delta V = \delta R^i_{ub}(t^i_k, t^i_{k+1}) \chi R_j(t^j_m)$, where $R_j(t^j_m)$ is the status of relation $R_j$ at time $t^j_m$, and $t^j_m$ is the time at which source $S_j$ transmitted its last batch update just before time $t^i_{k+1}$. This is done to avoid the effect of concurrent updates as explained in [16]. This process is continued as source $S_j$ forwards this message to its next source $S_h$. The message is propagated along the sources, gradually calculating $\delta V$ until the last source transmits to the DW the message $(\delta V, t^i_k, t^i_{k+1}, i)$, where,

$$\delta V = \Pi_{Av}(\sigma_p(R_1 \chi R_2 \chi \ldots \chi \delta R^i_{ub}(t^i_k, t^i_{k+1}) \chi \ldots R_n))$$

At the DW site, a temporary relation *Temp_View* along with an ordered list *Batches_List(i)* for each source $S_i$ are used to store the received $\delta V$'s and their corresponding time intervals, respectively. Each *Update_Window* time units, the DW updates its view content up to the maximum time where all the updates from the $n$ sources were received. This is easily achieved using the contents of Batches List and $cr_i$ and $dr_i$ attributes stored for each tuple in *Temp_View*. A detailed description of the algorithm and a proof of its correctness can be found in [16].

It can be seen that in the Pure-Push algorithm, the DW stays passive waiting for the view change to be calculated. This in fact gives each source the freedom to determine its suitable update periods and transmission time, giving more flexibility that suits the mobile environments. However, the cost of updating the view in response to an update batch occurring in one of the sources is $n$ messages, i.e., a total of $n^2$ messages are required in response to the update batches of the n sources. In addition, it requires the DW to keep track of update messages arriving from each source. The following section presents a novel algorithm that aims at overcoming these difficulties.

## 4. The interleaved Pull-Push algorithm

In the Interleaved Pull-Push algorithm, the DW is responsible for pulling sources' updates occurring during a certain common interval. The update process starts when the DW sends an update request to sources $S_1$ and $S_n$. In

response to the DW request, source $S_1$ initiates a message, *Forward Message*, that propagates along the $n$ sources in a forward direction until it finally reaches source $S_n$. In a similar manner, source $S_n$ initiates a second message, *Backward Message*, which is propagated along the $n$ sources in the reverse direction until it reaches source $S_1$. As the two messages propagate in the forward and reverse directions, each source appends its update information to both messages. Finally, the two messages reach the DW where the view is updated. The total number of messages needed for the batches of the $n$ sources during an update interval is $2n$ messages, i.e., each source needs only to transmit two messages instead of the $n$ messages required by the Pure-Push algorithm.

For simplicity, we assume that the next source is determined based on the view definition. More sophisticated mechanisms can be used to optimise the communication cost, and the message size. Elaboration on different methods for choosing the next source can be found in [16].

### 4.1. Algorithm description

Each *Update_Window* time units, the DW starts pulling the updates of the $n$ sources that occurred since the last time the DW triggered an update request. In other words, if the DW triggered an update request at time $t_k$, then the next update request will be triggered at $t_{k+1} = t_k + Update\_Window$. The DW first transmits an update request of the form $(t_k, t_{k+1})$ to two sources, $S_1$ and $S_n$. When $S_1$ receives the update request, it constructs a message, *frwrd_msg*, containing its updates, $\delta R^1_{ub}(t_k, t_{k+1})$, that occurred during the interval $]t_k, t_{k+1}]$. $\delta R^1_{ub}(t_k, t_{k+1})$ is calculated using the creation and deletion time tags, $cr_1$ and $dr_1$, respectively, in the following way,

$$\delta R^1_{ub}(t_k, t_{k+1}) = (\Delta R^1_{ub}(t_k, t_{k+1}), \nabla R^1_{ub}(t_k, t_{k+1})),$$

where $\Delta R^1_{ub}(t_k, t_{k+1})$ is the set of insertions of $R_1$ occurring during the interval $]t_k, t_{k+1}]$ and could be easily calculated as the set of tuples with the creation time tag belonging to that interval, i.e.,

$$\Delta R^1_{ub}(t_k, t_{k+1}) = \{r \in R_1 \;//\; r\{cr_1\} \in ]t_k, t_{k+1}]\}.$$

Similarly, $\nabla R^1_{ub}(t_k, t_{k+1})$ is the set of deletions of $R_1$ occurring during the interval $]t_k, t_{k+1}]$ and is calculated by;

$$\nabla R^1_{ub}(t_k, t_{k+1}) = \{r \in \bar{R}_1 \;//\; r\{dr_1\} \in ]t_k, t_{k+1}]\}.$$

$S_1$ forwards the message $(\delta V_{frwrd\_msg}, t_k, t_{k+1})$ to source $S_2$ where $\delta V_{frwrd\_msg} = \delta R^1_{ub}(t_k, t_{k+1})$, i.e., $\delta V_{frwrd\_msg}$ is part of view change that is gradually calculated as the message propagates in the forward direction.

To avoid the effects of concurrent updates, instead of using the current status of $R_2$ in performing the join, $S_2$ restores the status of $R_2$ at time $t_k$, $R^2(t_k)$. This is achieved using $cr_2$, $dr_2$, and, $\bar{R}_i$. Source $S_2$ then updates $\delta V_{frwrd\_msg}$ such that it becomes

$$\delta V_{\text{frwrd\_msg}} = (\delta R^1_{ub}(t_k, t_{k+1}) \chi R^2(t_k)) \cup \delta R^2_{ub}(t_k, t_{k+1}),$$

and then forwards the message $(\delta V_{frwrd\_msg}, t_k, t_{k+1})$ to the next source where the same process is repeated with each source $S_i$ performing both a join operation on $\delta V_{frwrd\_msg}$ with its base relation restored at time $t_k$, $R^i(t_k)$, and then appending its own update batch $\delta R^i_{ub}(t_k, t_{k+1})$ in the interval $]t_k, t_{k+1}]$, and transmitting the resultant message to its next source $S_{i+1}$. When *frwrd_msg* finally reaches the last source, $S_n$, it is sent directly to the DW. The contents of $\delta V_{frwrd\_msg}$ as it is received in the DW is shown in eq. (1);

$$\delta V_{\text{frwrd\_msg}} = \bigcup_{i=1}^{n-1} \delta R^i_{ub}(t_k, t_{k+1}) \chi R^{i+1}(t_k) \chi \ldots \chi R^n(t_k). \quad (1)$$

In a similar manner, another message propagates simultaneously in the reverse direction with source $S_n$ receiving an update request from the DW in the form $(t_k, t_{k+1})$ indicating that the DW is pulling for updates

occurring during the interval $]t_k, t_{k+1}]$. As the message travels in the reverse direction, each source performs a join operation and appends its own update in a similar manner to the forward message. The contents of the reverse message upon arrival at $S_1$ is shown in eq. (2)

$$\delta V_{bckwrd\_msg} = \bigcup_{j=2}^{n} R^1(t_{k+1}) \chi \ldots \chi R^{j-1}(t_{k+1}) \chi \delta R^j_{ub}(t_k, t_{k+1}). \quad (2)$$

where $\delta V_{bckwrd\_msg}$ is part of view change that is gradually calculated as the message propagates in the reverse direction.

When $\delta V_{frwrd\_msg}$ and $\delta V_{bckwrd\_msg}$ reach the DW, a special operator is applied to them yielding the view change, $\delta V$, in response to the update batches at the $n$ sources. This operator is referred to as special join operator and denoted by $\otimes$. The special join operation is performed by joining the individual terms of eq. (1) with their corresponding parts in eq. (2) such that $i = j$, in the following manner:

$$\begin{aligned}
\delta V &= (\delta V_{bckwrd_msg}, t_k, t_{k+1}) \otimes \\
&\quad (\delta V_{frwrd_msg}, t_k, t_{k+1}) \\
&= (\delta R^1_{ub}(t_k, t_{k+1}) \chi R^2(t_k) \chi \ldots \chi R^n(t_k)) \\
&\cup (R^1(t_{k+1}) \chi \delta R^2_{ub}(t_k, t_{k+1})) \chi (\delta R^2_{ub}(t_k, t_{k+1}) \chi \\
&\quad R^3(t_k) \chi \ldots \chi R^n(t_k)) \\
&\cup \quad \ldots. \\
&\cup (R^1(t_{k+1}) \chi \ldots \chi \delta R^i_{ub}(t_k, t_{k+1})) \chi (\delta R^i_{ub}(t_k, t_{k+1}) \\
&\quad \chi R^{i+1}(t_k) \chi \ldots \chi R^n(t_k)) \\
&\cup \quad \ldots. \\
&\cup (R^1(t_{k+1}) \chi \ldots \chi \delta R^n_{ub}(t_k, t_{k+1})) \\
&= (\delta R^1_{ub}(t_k, t_{k+1}) \chi R^2(t_k) \chi \ldots \chi R^n(t_k)) \\
&\cup (R^1(t_{k+1}) \chi \delta R^2_{ub}(t_k, t_{k+1}) \chi R^3(t_k) \chi \ldots \chi R^n(t_k)) \\
&\cup \quad \ldots. \\
&\cup (R^1(t_{k+1}) \chi \ldots \chi \delta R^i_{ub}(t_k, t_{k+1}) \chi R^{i+1}(t_k) \chi \ldots \chi R^n(t_k)) \\
&\cup \quad \ldots. \\
&\cup (R^1(t_{k+1}) \chi \ldots \chi \delta R^n_{ub}(t_k, t_{k+1})) \\
&= \left( \bigcup_{i=1}^{i=n} (R^1(t_{k+1}) \chi R^2(t_{k+1}) \chi \ldots \chi \delta R^i_{ub}(t_k, t_{k+1}) \right. \\
&\quad \left. \chi R^{i+1}(t_k) \chi \ldots \chi R^n(t_k)) \right). \quad (3)
\end{aligned}$$

For simplicity, in eq. (3) no duplicate rows are assumed, however, the case of duplicate rows can be handled by adding a row identifier to eliminate producing the same row twice.

The DW stores the received $\delta V_{frwrd\_msg}$ and $\delta V_{bckwrd\_msg}$ of different time intervals in two temporary relations *Temp_frwrd* and *Temp_bckwrd*, respectively, in order to handle messages received in different orders. The DW then applies $\delta V_{frwrd\_msg}$ and $\delta V_{bckwrd\_msg}$ to the materialized view only when both $\delta V_{frwrd\_msg}$ and $\delta V_{bckwrd\_msg}$, of the time interval following the last time the view has been updated, have arrived and applied to the materialized view. That is when both $\delta V_{frwrd\_msg}$ and $\delta V_{bckwrd\_msg}$ of the time interval $t_k$ , $t_{k+1}$ are received and stored, the DW will apply those messages to the view if it has already been updated in response to $\delta V_{frwrd\_msg}$ and $\delta V_{bckwrd\_msg}$ of the time interval $t_{k-1}$ , $t_k$ . $\delta V_{frwrd\_msg}$ and $\delta V_{bckwrd\_msg}$ are then discarded from the temporary relations.

Each source $S_i$ handles lost messages and disconnection of next or previous source by waiting $\Delta t$ for an acknowledgement, if not received, $S_i$ retransmits the same message.

To avoid infinite growth of sources' logs, the DW periodically broadcasts a message to all $n$ sources containing a time stamp $t_v$, where $t_v$ is the ending time interval of the last applied view change to the materialized view at the DW. When source $S_i$ receives $t_v$, it can delete from its log file all tuples deleted before $t_v$.

In contrast to the "Pure-Push" algorithm, the creation and deletion time tags need not be stored in each tuple in the received messages. They are only used locally at each source during the calculation of $\delta Rs$. The only overhead encountered in the message size in both forward and backward messages is the time stamps $t_k$, $t_{k+1}$. Figs. 1 and 2 provide a pseudocode description for the Interleaved Pull-Push Algorithm at source $S_i$ and the DW, respectively.

**Algorithm 1: MODULE SOURCE(i)**

```
procedure RECEIVE_FORWARD_MESSAGE()
  repeat
    Receive frwrd_msg(δV_frwrd_msg, t_k t_k+1)
    Compute R^i(t_k),Compute  δV_frwrd_msg ← δV_frwrd_msg χ R^i(t_k)
    Append δR^i_ub(t_k, t_k+1) to δV_frwrd_msg
    repeat
     Send frwrd_msg (δV_frwrd_msg , t_k, t_k+1) to next source
     Wait Δt for an ACK
    until an ACK is received
  until Forever
procedure RECEIVE_BACKWARD_MESSAGE()
  repeat
    Receive bckwrd_msg(δV_bckwrd_msg , t_k; t_k+1)
    Compute R^i(t_k+1), Compute δV_bckwrd_ msg ← R^i(t_k+1) χ δV_bckwrd_msg
    Append δR^i_ub(t_k, t_k+1)to δV_bckwrd_msg
    repeat
      Send bckwrd msg(δVb_ckwrd_msg , t_k, t_k+1) to previous source
      Wait Δt for an ACK
    until an ACK is received
  until Forever
procedure EMPTY LOG()
  repeat
    if a broadcast message (t_v) is received from DW
     then  Delete all time stamps t_k < t_v
    Delete from R_i tuples with dr_i ≤ t_v
  until Forever
main
  repeat
    Start procedure RECEIVE_FORWARD_MESSAGE
    Start procedure RECEIVE_BACKWARD_MESSAGE
    Start procedure EMPTY_LOG
  until Forever
```

Fig. 1. Interleaved Pull-Push algorithm at source $S_i$.

**Algorithm 2: MODULE DATA WAREHOUSE()**

```
procedure REQUEST_UPDATES()
  repeat
   Each Update_Window Do
    send request msg (t_k, t_k+1) to S_1, S_n
  until Forever
procedure RECEIVE_UPDATES()
    Receive frwrd_msg(δV_frwrd_msg, t_k; t_k+1)
    Temp_frwrd ←Temp_frwrd + δV_frwrd_msg
    Receive bckwrd_msg(δV_bckwrd_msg, t_k, t_k+1)
    Temp_bckwrd ← Temp_bckwrd + δV_bckwrd_msg
    if (frwrd_msg(δV_frwrd_msg, t_k, t_k+1) and bckwrd_msg(δV_bckwrd_msg, t_k, t_k+1) are received )
      then
        Compute δV ← δV_bckwrd_ msg ⊗ δV_frwrd_msg
        Update the view v ← v + δV
        Temp_frwrd ← Temp frwrd - δV_frwrd_msg
        Temp_bckwrd ← Temp_bckwrd -δ V_bckwrd_ msg
        t_v ← t_k+1
        Broadcast t_v to all sources S_i
main
    repeat
        Start procedure REQUEST_UPDATES()
        Start procedure RECEIVE_UPDATES()
    until Forever
```

Fig. 2. Interleaved Pull-Push algorithm at the DW.

Since only two messages propagate along all sources, the total number of messages needed to update the view in response to the update batches of $n$ sources, is $2n$. In other words, each source contributes with only two messages.

Although, in this algorithm, the update message size can be larger, collecting updates of $n$ sources in two messages has the advantage of alleviating each source from the overhead of calculating the view change and constructing a separate message in response to each update of other sources.

### 4.2. Correctness

In this section, we develop a proof of correctness for the "Interleaved Pull-Push" algorithm. Consider a view, $v$, defined over $n$ sources $S_1$, $S_2$, ... , $S_n$, such that, $v = R_1 \chi R_2 \chi \ldots \chi R_i \chi \ldots \chi R_n$. Assume that at time $t_1 > t_k$ the content of the DW is $v(t_1) = R^1(t_k) \chi R^2(t_k) \chi \ldots \chi R^i(t_k) \chi \ldots \chi R^n(t_k)$. At time $t_{k+1}$ the DW transmits an update request $(t_k, t_{k+1})$ to $S_1$ and $S_n$, and the process continues as described before. When the DW receives $(\delta V_{frwrd\_msg}, t_k, t_{k+1})$ and $(\delta V_{bckwrd\_msg}, t_k, t_{k+1})$ it can first compute the result of joining both messages using eqs. (1) and (2) resulting in eq. (3). Applying the content of eq. (3), at time $t_2 > t_{k+1}$ to the view $v$, results in:

$$v(t_2) = v(t_1) + (\delta V_{bckwrd\_msg}, t_k, t_k + 1) \otimes (\delta V_{frwrd\_msg}, t_k, t_k + 1)$$

$$= (R^1(t_k) \chi R^2(t_k) \chi \ldots \chi R^i(t_k) \chi \ldots \chi R^n(t_k)) \cup \left( \bigcup_{i=1}^{i=n} (R^1(t_{k+1}) \chi R^2(t_{k+1}) \chi \ldots \chi \delta R^i_{ub}(t_k, t_{k+1}) \chi R^{i+1}(t_k) \chi \ldots \chi R^n(t_k)) \right) \quad (4)$$

$$v(t_2) = (R^1(t_k) \chi R^2(t_k) \chi \ldots \chi R^i(t_k) \chi \ldots \chi R^n(t_k)) \cup (\delta R^1_{ub}(t_k, t_{k+1}) \chi R^2(t_k) \chi \ldots\ldots \chi R^n(t_k)) \cup \left( \bigcup_{i=2}^{i=n} (R^1(t_{k+1}) \chi R^2(t_{k+1}) \chi \ldots \chi \delta R^i_{ub}(t_k, t_{k+1}) \chi R^{i+1}(t_k) \chi \ldots \chi R^n(t_k)) \right)$$

$$v(t_2) = (R^1(t_k) \cup (\delta R^1_{ub}(t_k, t_{k+1})) \chi (R^2(t_k) \chi \ldots \chi R^i(t_k) \chi \ldots \chi R^n(t_k)) \cup \left( \bigcup_{i=2}^{i=n} (R^1(t_{k+1}) \chi R^2(t_{k+1}) \chi \ldots \chi \delta R^i_{ub}(t_k, t_{k+1}) \chi R^{i+1}(t_k) \chi \ldots \chi R^n(t_k)) \right)$$

$$v(t_2) = R^1(t_{k+1}) \chi (R^2(t_k) \chi \ldots \chi R^i(t_k) \chi \ldots \chi R^n(t_k)) \cup \left( \bigcup_{i=2}^{i=n} (R^1(t_{k+1}) \chi R^2(t_{k+1}) \chi \ldots \chi \delta R^i_{ub}(t_k, t_{k+1}) \chi R^{i+1}(t_k) \chi \ldots \chi R^n(t_k)) \right)$$

$$v(t_2) = R^1(t_{k+1}) \chi [ (R^2(t_k) \chi \ldots \chi R^i(t_k) \chi \ldots \chi R^n(t_k)) \cup \left( \bigcup_{i=2}^{i=n} (R^2(t_{k+1}) \chi \ldots \chi \delta R^i_{ub}(t_k, t_{k+1}) \chi R^{i+1}(t_k) \chi \ldots \chi R^n(t_k)) \right) ]. \quad (5)$$

From eq. (4 and 5), by induction, eq. (5) can reduce to,

$$v(t_2) = R^1(t_{k+1}) \chi R^2(t_{k+1}) \chi \ldots \chi R^i(t_{k+1}) \chi \ldots \chi R^n(t_{k+1}). \quad (6)$$

It is clear that eq. (6) represents the correct content of the view after applying effects of all update batches of the $n$ sources during the interval $]t_k, t_{k+1}]$. Since the state transformation of the view occurs in the order of the updates, the proposed algorithm ensures strong consistency. Moreover, since each source $S_i$ participates in $\delta V_{frwrd\_msg}$ and $\delta V_{bckwrd\_msg}$ depending only on the time stamps, $t_k$ ,$t_{k+1}$, contained in the received messages, the result of the view change $\delta V$ is not affected by the order or the time $\delta V_{frwrd\_msg}$ and $\delta V_{bskwrd\_msg}$ were received at each source $S_i$.

In addition, due to the retransmission used in the algorithm, i.e., missed messages are retransmitted periodically until an acknowledgment is received, both forward and backward messages are guaranteed to be propagated along the sources and reach the DW, which ensures that the algorithm is live.

Further at the DW, time stamps stored with messages can be used to detect lost messages. This allows the FIFO assumption for the communication channel between the sources and DW to be relaxed.

## 5. Performance analysis

A simulation model, based on an adapted version of the WHIPS (WareHouse Information Prototype at Stanford) model introduced in [17, 18], has been developed. The performance of the Interleaved Pull-Push is analyzed and compared to the performance of both the Pure-Push [16] and Nested SWEEP algorithms [7]. The following subsections discuss the modified WHIPS architecture along with the experiments results and their analysis.

### 5.1. Modified WHIPS architecture

Several modifications were introduced to the WHIPS system model [17,18], as shown in fig. 3. The WHIPS architecture is divided into three layers: The information sources, integration modules, and the DW.

In the original WHIPS model, data sources consisted of sources' *wrappers* and *monitors*. In the modified architecture, a *meta-data store* is added in each source to store part of the view definition, and information for how to contact a certain set of other sources. *Wrappers* are responsible for translating single source queries from the internal WHIPS representation into queries in the native language of its source.

In the proposed algorithm, since information sources communicate directly, an *update manager* and a *query manager* are added in each source. A source *update manager* is responsible for the creation of the query needed to join the received part of the view with the base relation of the source. Meanwhile, a *query processor* is responsible for evaluating a query received from the *update manager* of each source.

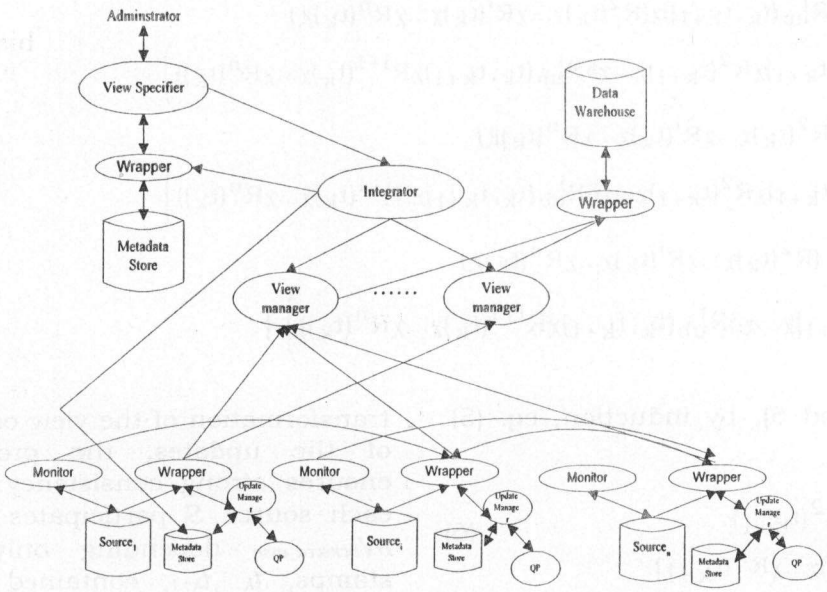


Fig. 3. Modified whips system architecture.

The *integrator module* is responsible, at initialization time, for creating sources' *query processors* and the *meta-data* stored at both the DW and the sources. When a new source becomes available, the *integrator* receives a notification from the source's *monitor* and *wrapper*. The *integrator* then stores the sources' information in the *meta-data* of the DW and in the necessary other *meta-data stores* in other sources.

In contrast to the original WHIPS model, the *view manager* has its work distributed among the *update managers* of the sources. Thus, it is no longer responsible for the creation of the global query. It can only receive the calculated view change and sends the appropriate view change to the DW manager for execution. It also manages the temporary relations, *Temp_View* in the Pure-Push algorithm, and *Temp_frwrd* and *Temp_bckwrd* in the Interleaved Pull-Push algorithm.
Since no global queries are generated by the *view manager*, the job of the *query processor* will be distributed among the *query processors* stored at each source. As in the WHIPS architecture, the DW *wrapper* receives view update changes and applies them to the DW views. Like source *wrappers*, the DW *wrapper* also translates the view changes to specific syntax of the DW database. A detailed discussion of modules functionalities along with model verification and validation can be found in [15].

### 5.2. Experiments results and analysis

In this section, the performance of the Interleaved Pull-Push is analyzed and compared to the performance of both the Pure-Push [16] and Nested SWEEP algorithms [7]. Nested SWEEP has been chosen for comparison as the SWEEP family [7, 8] generally outperforms the previous families of algorithms, ECA [5], and Strobe [6]. In addition, Nested SWEEP is selected specifically from the SWEEP family because the SWEEP algorithm follows immediate updating while the Nested SWEEP uses batch updating and achieves the same degree of consistency, strong consistency, as the two proposed algorithms. The Nested SWEEP algorithm was implemented using the WHIPS model, while both the Pure-Push and Interleaved Pull-Push algorithms were implemented using the modified WHIPS model.

Four base relations *R(A,B,RX)*, *S(B,C,SX)*, *T(C,D,TX)*, and *U(D,E,UX)* were chosen. *A*, *B*, *C*, *D*, and *E* are key attributes of their corresponding relations, and are 8 bytes each. *RX*, *SX*, *TX*, and *UX* are 80 bytes each. Data Warehouse environment settings are similar to that used in the WHIPS model [17,18]. Experiments were carried out in both stationary and mobile environments, considering views defined over different number of sources. Experiments duration was set to 10,000 units of time, and a confidence interval of 0.95 was obtained. Table 1 defines the four views used in the experiments and summarizes the experiments parameters and their settings. Values of input parameters and distributions have been inherited from the WHIPS model.

A detailed description of the experimentation environment and more experiments results can be found in [15].

#### 5.2.1 . Experiment 1: update propagation time

This experiment was set to measure the Update Propagation Time *UPT*, the total time needed for an update batch to reach the DW. The parameters used in this experiment were fixed to the following values: $F$ = 0.1 updates/sec for all sources, *Update_Interval* = 0.3 sec for all sources, probability of DW disconnection ($s$) = 0.1, probability of sources disconnection ($s'$) = 0.1. The *UPT* can be divided into four components: $t_{network}$, the total time needed per batch to propagate through the network, $t_{computation}$, the total time needed per batch in computations, $t_{query}$, the total time needed per batch to perform queries and $t_{inner\_communication}$ the total time needed per batch in communication between the WHIPS's inner modules. The experiment was run under two scenarios: non-interfering updates, where updates from different sources do not intersect, and interfering updates between different sources, where updates of different sources occur simultaneously, with a probability of interference = 0.65. Results for both cases are shown in figs. 4 and 5, respectively of the four views $V_1$, $V_2$, $V_3$ and $V_4$ comparing the three algorithms.

Table 1
Parameters settings

| Parameter | Description | Base setting |
|---|---|---|
| $n$ | Number of sources | 4 |
| $V_1$ | $= \Pi_{A,B}(R)$ | |
| $V_2$ | $= \Pi_{R.A,R.B,S.B,S.C}(R \chi S)$ | |
| $V_3$ | $= \Pi_{R.A,R.B,S.B,S.C,T.C,T.D}(R \chi S \chi T)$ | |
| $V_4$ | $= \Pi_{R.A,R.B,S.B,S.C,T.C,T.D,U.D,U.E}(R \chi S \chi T \chi U)$ | |
| $\|\|R\|\|$ | Base relation cardinality | 1000 tuples |
| $\sigma$ | selectivity | $0 \le \sigma < 1$ |
| $J$ | Join Factor | 1 |
| $S$ | the fraction of the total time spent by a mobile DW in the sleep mode. | 0~0.9 |
| $\omega$ | $1/\omega$ is mean value of the exponentially distributed interval t after which the data warehouse changes its status | 1800sec |
| $s'$ | The fraction of the total time spent by a mobile source in the sleep mode. | 0~0.9 |
| $\omega'$ | $1/\omega$ is mean value of the exponentially distributed interval t' after which a source changes its status | 1800 sec |
| $\Delta t$ | time to wait for an acknowledgement | 0.2 |
| $F$ | Updates arrival distribution | 0.1 ~1 per sec |
| *Update_Interval* | Time between transmitting update batches | 0.1~1 sec |
| *Update_Window* | DW update window | 0.1~1 sec |

For the case of non-interfering updates, it was observed that both $t_{inner\_communication}$ and $t_{computation}$ increase with the number of sources participating in the view definition for both Pure-Push and Interleaved Pull-Push algorithms. However, they remain almost independent of the number of sources in the case of the Nested SWEEP algorithm. This is due to the fact that in the first two algorithms, both the update manager and the query processor modules are located at each source rather than at the DW, as in the case for the Nested SWEEP. As the number of sources

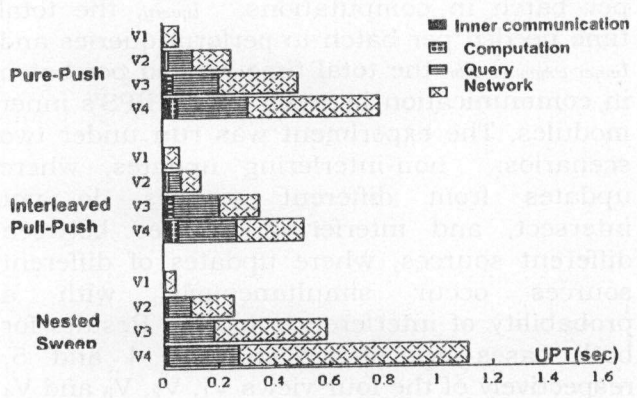


Fig. 4. UPT with non-interfering updates.

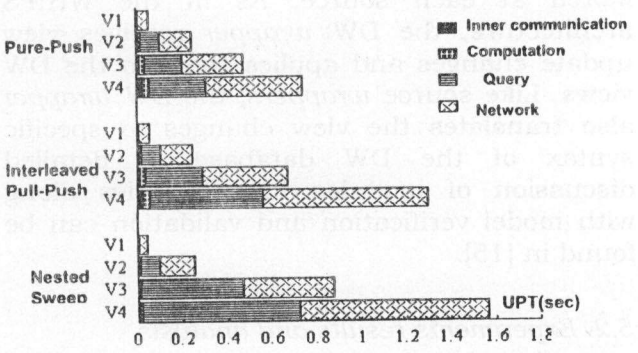


Fig. 5. UPT with interfering updates.

increases, $t_{network}$ and $t_{query}$ dominate *UPT* with $t_{query}$ almost identical for the three algorithms. However, $t_{network}$ is significantly smaller in the case of the Interleaved Pull-Push algorithm where it becomes roughly 25% of that of the Nested SWEEP and almost 50% of the Pure Push algorithm. Comparing the UPT of the three algorithms in this case shows that the Interleaved Pull-Push algorithm has the least UPT. The superiority of the Interleaved Pull-Push in that case is due to the relatively small size of both forward and backward messages, as at the end of the update batch process, each of them contains almost half of the view

change in comparison with the other two algorithms

It is observed that Nested SWEEP performs poorly in the case of interfering updates, where $t_{query}$ increases sharply by almost 160% for $V_3$ and 170% for $V_4$ over the case of non-interfering updates. This is due to the local compensation performed at the DW to eliminate the effect of concurrent updates. Interleaved Pull-Push shows an increase in both $t_{network}$ and $t_{query}$ due to the increase in the size of both the forward and backward messages which both carries a partially calculated view change in response to several update batches from different sources. However it is better than Nested SWEEP by 15%. It is noticed that the Pure-Push algorithm outperforms both algorithms, where all components of *UPT* remain the same as in the case of non-interfering messages. This is because as an update batch propagates along the sources, the view change is calculated in the same way regardless of update interference.

*5.2.2 . Experiment 2: throughput*

In this experiment, the throughput of the three algorithms is demonstrated under various batches update intervals. Figs. 6 and 7 show the average throughput for the three algorithms in case of view $V_3$ under, non-interfering and interfering updates.

It is observed that for non-interfering updates, the Interleaved Pull-Push outperforms both the Pure Push and Nested SWEEP algorithms by achieving the best throughput, especially for high update rates. This is mainly due to the fact that the Interleaved Pull-Push reduces the number of messages, while the increase in the message size is not significant. As the update interval increases beyond 0.8 sec, the time interval between two consecutive updates reaching the DW is sufficient for the completion of the update process in the three algorithms. Therefore, as observed, the throughput of the three algorithms becomes almost identical.

In case of interfering updates, Nested SWEEP shows the least throughput. This is due to the local query compensation which causes $t_{query}$ to increase which in turn reduces the throughput. The Interleaved Pull-Push shows better throughput than Nested SWEEP, but lower throughput than Pure-Push. This is due to the increase in the size of both forward and backward messages, which in turn increase $t_{network}$. It is observed that the performance of the three algorithms becomes almost identical for update intervals greater than 0.7 seconds.

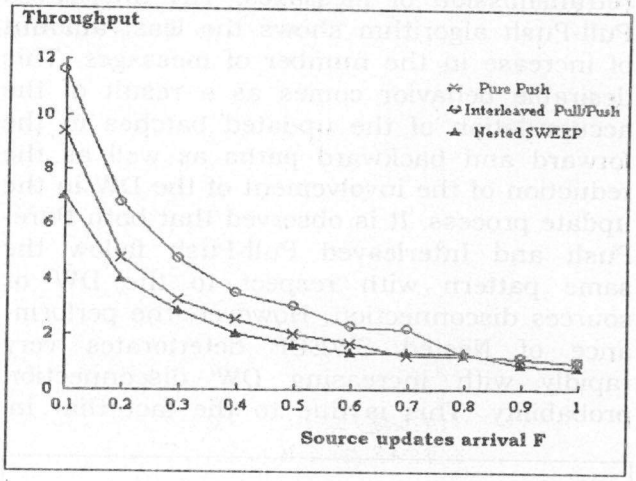


Fig. 6. Throughput (# of updates per second) with non-interfering updates (view $V_3$).

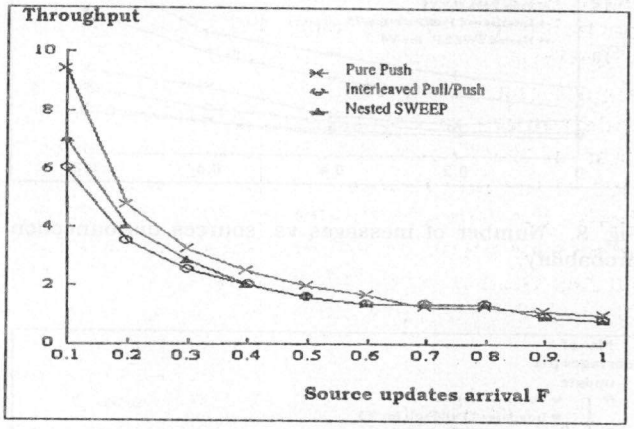


Fig. 7. Throughput (# of updates per second) with interfering updates (view $V_3$).

*5.2.3. Experiment 3: number of messages*

This experiment demonstrates the behavior of the three algorithms based on the number of messages needed per an update batch versus different probabilities of disconnection for both the sources and the DW. Figs. 8 and 9 display the number of

messages needed per an update batch versus disconnection probability at the sources and the DW, respectively for the case of interfering updates.

It is observed that as the rate of disconnection for the sources or DW increases, the number of messages needed for update propagation increases, due to the retransmission of messages. The interleaved Pull-Push algorithm shows the least amount of increase in the number of messages. This desirable behavior comes as a result of the accumulation of the updated batches in the forward and backward paths as well as the reduction of the involvement of the DW in the update process. It is observed that both Pure-Push and Interleaved Pull-Push follow the same pattern with respect to the DW or sources disconnection. However, the performance of Nested SWEEP deteriorates very rapidly with increasing DW disconnection probability. This is due to the fact that in Nested SWEEP the DW assumes a major role in sending and receiving intermediate messages while calculating the view change.

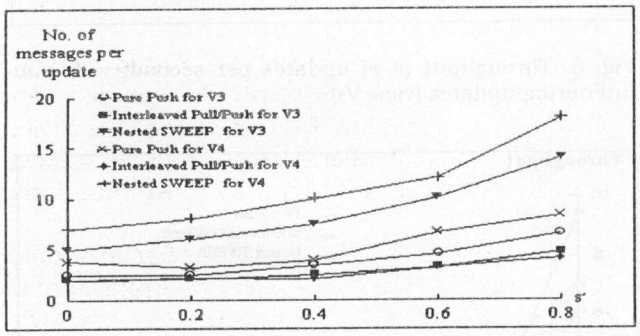


Fig. 8. Number of messages vs. sources disconnection probability.

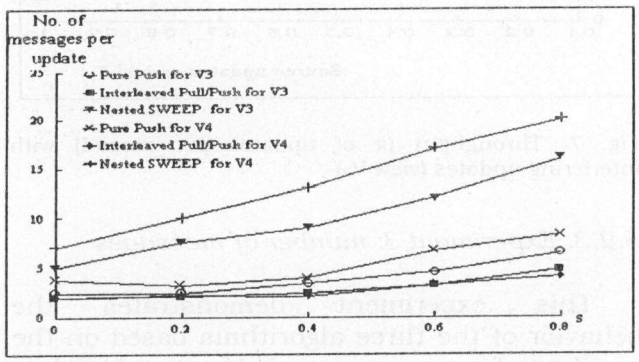


Fig. 9. Number of messages vs. DW disconnection probability.

## 6. Conclusions

In this paper we presented a novel algorithm, Interleaved Pull-Push, for incrementally updating materialized views defined over sources and stored in a DW, where both the DW and sources could be mobile. The total number of messages needed to update the view in response to the update batches of $n$ sources, is $2n$, i.e., each source contributes with only two messages. Query compensation messages used by previous approaches have been eliminated. In addition, having the update messages propagate along the sources eliminates intermediate results from reaching the DW and thus eliminating the time a message spends in the network.

To analyze the performance of the proposed algorithm, a simulation model based on an adapted version the WHIPS architecture was implemented. The experiments performed showed that the update propagation time, the time needed for an update to reach the data warehouse is less than the time needed in traditional view maintenance algorithms. In addition, better throughput is obtained specially in cases of higher frequencies of updates in sources.

## References

[1] D. Stanoi, S. Agrawal, A. Abbadi, Phatak and B. Badrinath, "Data Warehousing Alternatives for Mobile Environments", in Proc. Of MobiDE'99, Seattle, Washington, pp. 110–115 (1999).

[2] N. Huyn, "Efficient self-Maintenance of Materialized Views", in Proc. of the ACM Workshop on Materialized Views: Techniques and Applications, Montreal, Canada (1996).

[3] D. Quass, A. Gupta, I. Mumick and J. Windom, "Making Views Self Maintainable for Data Warehousing", in Proc. Fourth Intl. Conf. On Parallel and

Distributed Information Systems (PDIS) (1996).

[4] R. Hull and G. Zhou, "A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches", in Proc. ACM SIGMOD '96, pp. 481–492 (1996).

[5] Y. Zhuge, H. Garrcia-Molina, J. Hammer and J. Windom., "View Maintenance in a Warehousing Environment", in Proc. ACM SIGMOD '95, pp. 316–327 (1995).

[6] Y. Zhuge, H. Garrcia-Molina and J. Wiener, "The Strobe algorithms for multisource warehouse consistency", in Proc. Fourth Intl. Conf. on Parallel and Distributed Information Systems (PDIS) (1996).

[7] D. Agrawal, A. Abbadi and A. Singh, "Efficient view maintenance at data warehouses", in SIGMOD Conference, pp. 417–427 (1997).

[8] X. Zhang and E. Rundensteiner, PSWEEP: Parallel View Maintenance Under Concurrent Data Updates of Distributed Sources, Tech. Report WPI-CS-TR-99-14, Worcester Polytechnic Institute, Dept. of Computer Science (1999).

[9] T. Imilienski and R. Badrinath, "Mobile Wireless Computing: Challenges in Data Management", Communication of ACM, Vol. 37 (10) (1994).

[10] G. Forman and J. Zahorjan, "The Challenges of Mobile Computing", IEEE Computer Journal, Vol. 27 (4), pp. 38–47 (1994).

[11] S. Lauzac and P. Chrysanthis, "Programming views for mobile database clients", in Database and Expert Systems Applications( DEXA) workshop, Vienna, Austria, pp. 408–413 (1998).

[12] O. Wolfson, P. Sistla, A. Dao, K. Narayanan and R. Raj, "View Maintenance in Mobile Computing", in SIGMOD Record 24(4), pp. 22–27 (1995).

[13] H. Chung and H. Cho, "Maintenance of Materialized Views in Mobile Computing Environments", in Intl. Tech. Conf. On Circuits/Systems, Computers and Communications ITC-CSCC, pp. 617–620 (1996).

[14] K. Lee, A. Si and H. Leong, "Incremental View Update for a Mobile DataWarehouse", in ACM Symposium on Applied Computing (SAC), Atlanta, Georgia, USA, pp. 394–399 (1998).

[15] N. Samaan, Incremental View Maintenance Algortihms for Data Warehouses in Mobile Environments, M.Eng. Thesis dissertation, Alexandria Univ., Egypt (2001).

[16] N. Adly, Y. Taha, N. Samaan and M. Nagi, "An Incremental View Maintenance Algortihm for Data Warehouses in Mobile Environments", in IEEE International Symposium on Signal Processing and Information Technology(ISSPIT), Cairo, Egypt (2001).

[17] J. Wiener, H. Gupta, W. Labio and Y. Zhuge, "A System Prototype for warehouse View Maintenance", In Proc. of the ACM Workshop on Materialized Views: Techniques and Applications (1996).

[18] Y. Zhuge and H. Molina, "Performance Analysis of WHIPS Incremental Maintenance.", Technical report, Stanford University, http://www-db.stanford.edu/pub/papers/zgwhipsperf.ps (1998).

[19] D. Mills, "Internet time synchronization: the network time protocol.", IEEE Trans. on Communications, Vol. 39 (10) (1991).

[20] E. D. Kaplan, Understanding the GPS: Principles and Applications, Artech House, Boston, MA, ISBN:0-89006-793-7 (1996).

Received August 6, 2002
Accepted October 26, 2002