Utilizing Versions of Views within a Mobile Environment

Susan Weissman Lauzac, *Panos K. Chrysanthis*
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, U.S.A.
{suew,panos}@cs.pitt.edu

Utilizing Versions of Views within a Mobile Environment

Abstract

Data caching and hoarding provide the only means to support disconnected mobile operations. In the context of mobile database applications, data cached can take the form of a materialized view. In this paper, we present a mechanism or *view holder* within the fixed network, whose job is to maintain versions of the views that are required by a particular mobile host. These views are very likely to be a small and specialized portion of the information found within the various data sources and, therefore, versions can be dynamically maintained by the view holder without incurring huge storage requirements. In addition, the view holder will respond to mobile host's queries by communicating only the differences between versions. Thus, a view holder mediates and exports the views needed by a mobile host, and updates are computed and delivered in a flexible and batch manner.

- *Keywords:* Mobile Computing, View Maintenance, Mobile Query Processing, Data Warehousing

1 Introduction

Currently, relational database management systems are the most widely used database management systems in supporting database applications. Within a relational database system, data are structured as a set of (base) tables whereas a *view* defines a function from a subset of base tables to a derived table. A view is *materialized* by physically storing the tuples of the view derived from the base tables. These *materialized views* save the tuples of important or frequent queries. In distributed environments with client-server configurations, materialized views can be stored at the clients to support local query processing. Materialized views operate in a fashion similar to data caches. Available data is accessed quickly through the materialized views without having to query a remote database server. The retrieval of data from materialized views, as in the case of tables at the server, can be made even faster through indices associated with the appropriate materialized views [ZGMHW95].

Research in mobile computers and wireless networks is allowing mobile clients to become an integrated part of distributed computing environments along with their stationary counterparts. Due to the communication costs and frequent disconnections of wireless networks, information stored within the mobile computer becomes crucial to maintaining productivity. If the data needed to complete a task are present on the mobile computer, remote access may be eliminated and processing may continue even though disconnection has occurred. Thus, the role of materialized views and view maintenance is becoming increasingly important in the context of mobile database applications because of their ability to support local data processing. Given that most of the transactions in a database environment are read-only transactions, in this paper, we focus on optimizing these read-only transaction on mobile computers by utilizing versions of materialized views. However, write transactions can still occur, but are only performed directly with the data sources and not through the materialized views stored on the mobile client. This work complements the existing research on mobile read-only transaction that has focused on the efficient dissemination of data to large number of mobile computers by exploring the broadcaset characteristics of the wireless communication [IVB97, AAFZ95, DCVKK97].

Within the context of a known group of application processes, a mobile computer will need a specific subset of the data available to be organized, summarized and gathered in a efficient manner from very different information systems. Therefore, there is a need for a mechanism within the fixed network that can reduce computation and communication costs by being able to: (1) buffer the data that cannot fit on the mobile host due to inadequate storage capabilities, (2) be a proxy for the mobile host in order to receive updates from the sources during periods of inadequate networking conditions, and (3) become a cache within the fixed network that represents the needs of a mobile host. In other words, we need to provide a *dynamic and customizable view maintenance mechanism* so that the *cache or view consistency* achieved between the data stored on the mobile host and the data sources match the availability or cost of the network and the capabilities of the mobile host. This mechanism is what we will call the *view holder*.

A view holder is similar to a *data warehouse* that stores materialized views from multiple base table sources found in various information systems. In contrast to view holders, the views in a data warehouse are often *large*, *static* and considered *generic* and *stateless* with respect to the individual clients. Hence, while data warehouses reduce the cost for answering repetitive queries, they do *not* reduce the cost for communicating these answers in the case that they are the same or have small differences. View holders, on the other hand, respond to a mobile host's queries by communicating only the differences between answers. A view holder is programmed to maintain multiple versions of a view in order to compensate for the data changes that occurred to the materialized views that were used during disconnection. In short, a view holder mediates and exports the views needed by a mobile host, and updates are computed and delivered in a flexible and batch manner. Thus, a view holder can be thought of as a customizable client-oriented data warehouse.

Within our example application environment there exists a data warehouse within a fixed network utilizing a versioning maintenance algorithm and multiple mobile hosts. The rest of this paper continues as follows: Section 2 presents a scenario for our example application and then describes its limitations within a mobile environment and explains the need for a view holder mechanism. Section 3 describes the view holder within the context of the example, and then presents algorithms for the view holder which allow for interactions with the data warehouse as well as the mobile hosts. Section 4 describes our prototype currently being developed that incorporates the concepts discussed in this paper. Finally, the conclusions and future work are presented in Section 5.

2 Motivating Example

In this section, we motivate further the need for a view holder by means of an example. Suppose that the database structure of our fixed network environment includes not only the database servers responsible for storing base relations, but also a *data warehouse* which stores materialized views derived from these base tables sources. It is very important to understand that this data warehouse holds the *static* views that contain useful summary information which must be maintained periodically by the execution of a *maintenance transaction* which is initiated by the data servers. Queries from a client or mobile host (MH) can be answered in the form of a *materialized view*, and throughout this paper a view will be considered materialized once it is defined within the *fixed network*.

Let us assume that the data warehouse is maintained with a versioning algorithm, such as the *"two version no locking" (i.e., 2VNL)* algorithm [QW97], where one or two versions may be available for the readers at any time without having them wait on any locks.

2.1 The data warehouse's view: DailySales

Our data warehouse in this example supplies summary information from base tables for a chain of sporting goods stores and will contain two separate versions of the data. The data warehouse has the following materialized view which totals daily sales by city and date:

DailySales(`tupleVN, operation, city, product, date, total_sales, pre_totalsales`)

`tupleVN` keeps the version number of the maintenance transaction that last updated this tuple, while `operation` keeps the last *logical* operation performed. The attributes `city, product,` and `date` are non-updatable attributes that do not change, whereas `total_sales` must be periodically updated by a maintenance transaction. Since two copies or versions of this updatable attribute will be made available, the most current version is held in `total_sales` while the previous version is within `pre_totalsales`. When a maintenance transaction is currently updating the tuples, only the *current* version is available to readers while the new version is being created. When there is no maintenance transaction executing, both the previous and current versions of the attribute can be read [QW97].

Now suppose that a MH starts an application which will allow the user to see and perform some rough calculations regarding the sales of racquetball equipment. This MH will request a view from the data warehouse's **DailySales** inquiring about the day to day sales of racquetball equipment by stores in each city.

Query: RqballSales

```
SELECT city, date, product, total_sales
FROM DailySales
WHERE product = “rqball”
GROUP BY city, date
```

We want this query to be processed within the fixed network to save both the energy and resources of the MH. The results are then communicated to the MH's as a materialized view. The MH may keep this view for some time and may *not* receive the current day's sales figures due to traveling or communication delays. Eventually, another application such as a spreadsheet and graphing tool could be started that would allow the user to create slides for an upcoming presentation. At this point, the most recent results may be available, or communication conditions may have improved (e.g., the user is dialing up from a hotel room after work). Within the new application, the most recent sales figures can be incorporated into the spreadsheet.

Typically, once the MH receives the most recent version of the data, the user will be running two applications and accessing two separate versions of the view *at the same time*. This is in contrast to a single *reader session*, in a data warehousing environment, where each consecutive query comes from the same version. Once many maintenance transactions occur and the data is considered too old, the user must end its current work and gather a new version because the older one has *expired*. Therefore, view maintenance is achieved by forcing the client or MH to receive a new version of the materialized view. However, such a view maintenance strategy is not suitable and potentially very expensive for a MH. We elaborate on this next.

2.2 Problems

Although it may appear that directly accessing the data warehouse for data is acceptable, there are actually a wide range of difficulties. Some of these problems stem from the limited resources of the MH and wireless networking conditions, while others come from the static nature of the data warehouse.

From the viewpoint of a mobile host these problems include:

- It is *not* always convenient or even possible for a MH to receive a new version of the view at the time of expiration, especially during a period of disconnection or under poor networking conditions.
- If the MH can not receive a new version then the work done during a period of disconnection with an expired version may have to be discarded upon reconnection.
- The MH may not have enough *adequate storage* to hold several versions of a materialized view or even one entire version.

The data warehouse and data servers are stateless and do not keep track of who is interested in their ongoing changes. They will *not contact a mobile host* and inform them when updates are to be performed. From this we can see the following difficulties:

- To update the MH, the data warehouse will have to send the entire new version of the view each time since it can not compute the difference between versions. Communicating this entire recomputed version is *costly*, especially if there are very few differences between versions.
- Since data warehouse's views are *static*, the query **RqballSales** will *not* be maintained by the data warehouse. It would be a waste of space for the data warehouse to build separate views for only a couple of users.
- Each time a MH's materialized view is to be updated the data warehouse will have to reconstruct the materialized view from scratch.
- The data warehouse will *not* create or maintain an *indexing* structure for the query **RqballSales** that can be communicated to the MH.

- It would be difficult for the data warehouse to combine data from several heterogeneous sources if it did not have all the data being requested by the MH.
- The data warehouse becomes a centralized source, and therefore, a bottleneck of the system.

2.3 Our Solution

The above discussion clearly pointed out that direct interaction between the MH and the data sources cannot be supported in a flexible manner. To alleviate the problems discussed, *without* requiring modifications to the existing databases and data warehouses (i.e., unlike solutions presented in [BDMW95]), the mechanism we created for maintaining the materialized views requested by a mobile host is called the *view holder*. The goal of the view holder is to bring the data closer to the mobile hosts and control the interactions between the mobile hosts and the data sources.

Every application of a mobile host can only use a subset of the data that exists in the sources or, in our example, the data warehouse DW. We say the application *superset or superview SV* contains all the information that will be used by an application. The superview is really just a materialized view defined as a query *Q* applied to the data warehouse (*SV = Q(DW)*).

For a specific application environment, MH can request that a view holder maintains this superview *SV* of the data that it could possibly need. Then the MH can cache or hoard[1] a subset of this superview *SV* before a period of disconnection or a weak connection. In the example given, the mobile machine may not have the ability to store all the data regarding every store that sells racquetball equipment in every city from the query **RqballSales**. However, if the user is traveling to Pittsburgh, only the data concerning stores in this city will be downloaded to the mobile machine. The request for the query **RqballSales** is what we call the *initiation message* and forms the superview, while, the request for all information concerning Pittsburgh from the view holder is called the MH's *cache message*.

The view holder can keep track of the updates performed to the query **RqballSales**, as well as, the specific changes to the data from Pittsburgh. In other words, the superview will be incrementally maintained (SV' = *SV* + Δ), and only data from the Δ will need to be communicated to the mobile host. In this way, the view holder will be able to reduce the amount of wireless communication required to update a MH when it is possible. Other structures, such as indices, can be built and maintained by the view holder and later communicated to the MH.

[1] The term *hoarding* was introduced in [KS92] to identify the state a MH is in when it is collecting useful data in anticipation of disconnection.

As shown in Figure 1, we propose a layered system architecture where the data servers and data warehouse are more closely coupled within the fixed network than the materialized views maintained by the view holders. The data server layer is responsible for periodically constructing a maintenance transaction in order to update the data warehouse. A data warehouse, using a versioning maintenance algorithm, is created where the views are static and the number of consecutive versions of each view also remains static. Hence, the amount of space made available for versions of a particular attribute is known and fixed.

In contrast, a view holder will maintain a version of a view requested by a MH for as long as the host needs it. So, the view holder can be seen as a *buffer*, holding versions of a specialized view for a particular MH. Space allocated for the updated attributes of a view must be done dynamically since it is not known beforehand how many versions will be maintained. The idea here is that the views requested by an MH are very likely to be a small and specialized amount of the information from within the data servers and/or data warehouses. The current size of many data warehouses and other decision support databases (up to 4.6 TB [WA97, WA98]) precludes the simple solution of duplicating the available data on the mobile machine. To avoid these huge storage requirements, versions of the requested data are dynamically maintained by the view holder.

It is possible some of the data sources including data warehouses may *not* support explicit versions of data. In such a case, the view holder will query the source in order to extract the data at a given moment. A timestamp for this implicit "version" could be the last time the tuple, attribute, or table was modified and found by querying the catalog of the data source.

For the rest of this paper, we assume the mobile internetworking environment proposed in [KVP96], which contains a static, *fixed* network of hosts and data servers. Each MH retains its network connection while moving by remaining in contact with special *stationary* hosts equipped with wireless interfaces, called the *mobility support stations* (MSS). Each MSS is responsible for an area called a *cell*. A MH can only communicate with one MSS at any given time, and therefore, be in only one cell. A view holder can reside anywhere in the fixed network, for example, on a MSS or within a location management server. The view holder could then migrate as part of a hand-off procedure based on the MH's movement within the system. However, this is beyond the scope of this paper.

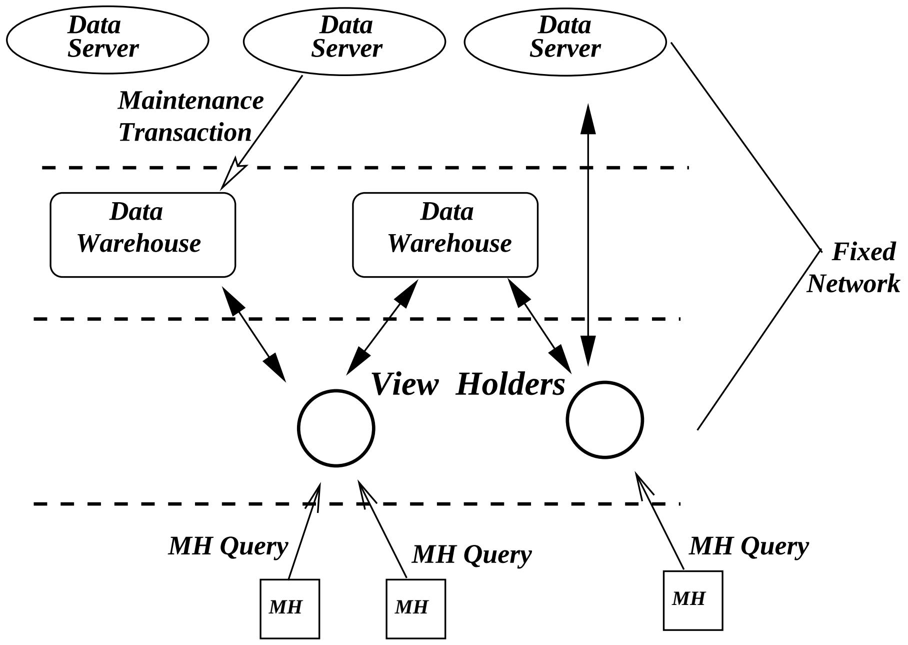


Figure 1: Overview of Architecture

3 View Holders

A view holder can potentially support many mobile clients. However, for clarity let us assume one view holder per MH in order to understand the interaction between the system's layers. Essentially, under this scenario, the view holder behaves as a *proxy* for the MH since the fixed network has greater storage capabilities and does not incur expensive communication costs.

The data structure used by the view holder is called the *Tuple Version List* (TVL) (see Figure 2). Each TVL maintains all the data necessary to keep available several versions of a tuple from the data warehouse. A TVL contains the key attributes and other non-updatable attributes of a tuple. It also has a linked list of versions for the updatable attributes used by the MH (i.e., the list Used) as well as the most current version of the tuple (Latest). The most current version number is kept in the view holder's local integer variable, Latest_Version, while an ordered list of available version numbers is also maintained by the view holder. The latest version can be prefetched (i.e., before it is needed) or materialized on demand. If it is prefetched then the MH does not have to wait for the TVLs to be updated before receiving the latest version. Any additional indices can still be built for the TVLs if desired to facilitate faster access during repetitive updates and queries.

The main purpose of the view holder is to always maintain any version being used by a MH application as well as the latest version. Every time there is a new version committed by the data

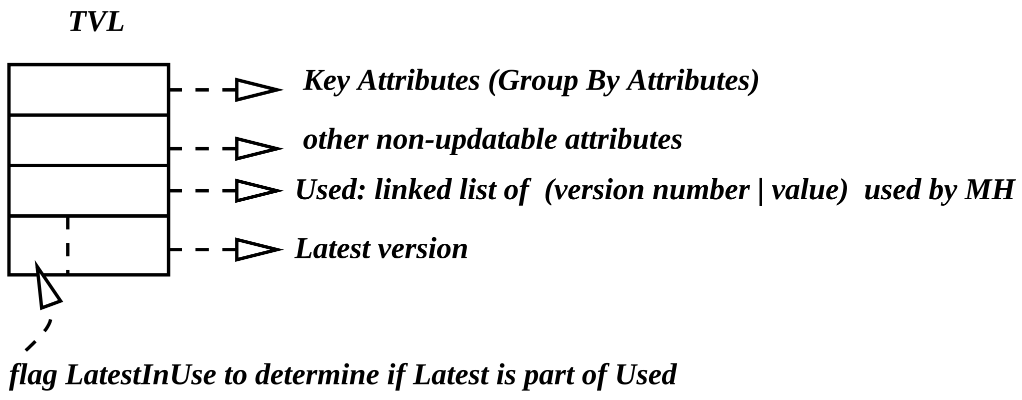

TVL
Key Attributes (Group By Attributes)
other non-updatable attributes
Used: linked list of (version number | value) used by MH
Latest version
flag LatestInUse to determine if Latest is part of Used

Figure 2: One Tuple Version List

tupleVN	operation	city	product	date	total_sales	pre_totalsales
2	update	Pittsburgh	golf	2-3-97	10,000	1,000
1	insert	Erie	golf	2-4-97	2,000	null
1	insert	Pittsburgh	rqball	2-5-97	3,000	null
2	update	Pittsburgh	rqball	2-6-97	40,000	4,000
1	insert	Erie	rqball	2-5-97	5,000	null
2	update	Lancaster	rqball	2-6-97	60,000	6,000

Table 1: Warehouse Materialized View DailySales

warehouse, the space allocated to the latest version is overwritten if the MH is not using it. Thus, anytime the MH starts a new application and requests the latest version, this is added to the linked list of versions currently being used preventing it from being overwritten.

Suppose that version 1 of the data warehouse's **DailySales** has been created and that our MH was interested in the smaller view regarding only racquetball equipment as seen in Section 2. After the MH contacted its view holder with its query, the view holder supplied version 1 from the data warehouse. If a maintenance transaction decides to update the view **DailySales**, then two versions of the view become available. Table 1 shows our materialized view within the data warehouse where both versions are committed and no new maintenance transaction is currently executing.

The `operation` attribute of a tuple stored in the data warehouse reflects the *net effect* of all operations performed on this particular tuple during the execution of a maintenance transaction. In a similar manner, the newest version of an attribute held by the latest pointer of the TVL shows the net effect of all the versions that took place between MH requests.

The view holder can access either version 1 or 2 from the data warehouse by looking at the `tupleVN` and `operation` attributes of each tuple. These attributes will determine if the `total_sales` or the `pre_totalsales` attribute should be read for each tuple. Since the view holder must maintain the version needed by the MH and the latest version, our view holder currently looks like the left-hand side of Figure 3. Recall that the flag LatestInUse (marked U) indicates that the latest version of this tuple is currently being used by the MH. We can see in the first and third TVLs that there has been no changes in these tuples between version 1 and version 2.

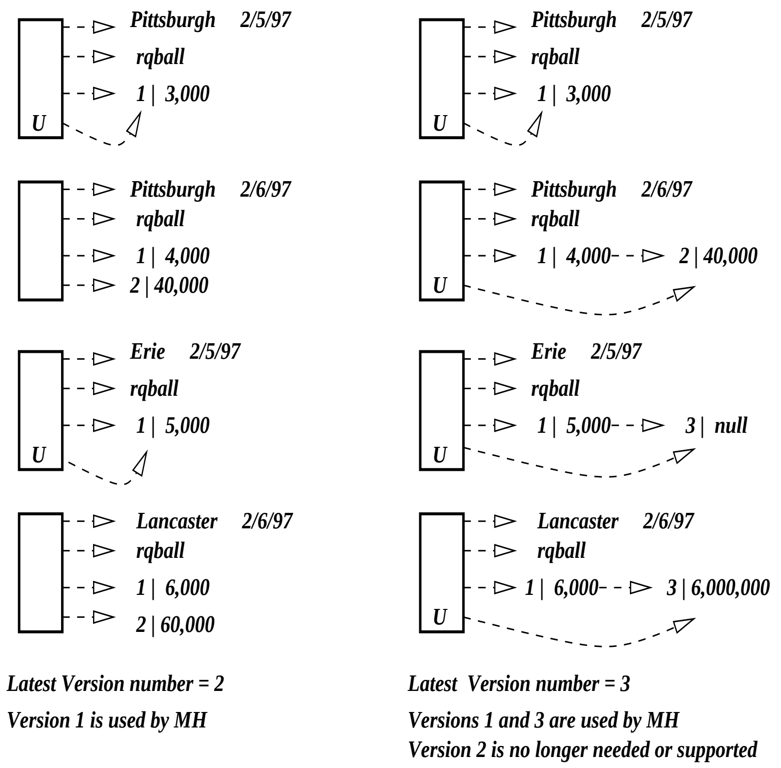


Figure 3: TVL before and after version 3 is stored

3.1 Updating the View Holder

So far, two maintenance transactions originating from a data server have executed within a data warehouse creating the two versions of tuples we see in our TVL structures. Now suppose a third maintenance transaction goes into effect at the data warehouse. Once the third maintenance transaction commits there are only two versions available within the data warehouse, versions 2 and 3. Any tuple labeled version 1 which has not been touched since its creation, still belongs to the other versions, although, version 1 is now considered *expired*.

tupleVN	operation	city	product	date	total_sales	pre_totalsales
2	update	Pittsburgh	golf	2-3-97	10,000	1,000
1	insert	Erie	golf	2-4-97	2,000	null
1	insert	Pittsburgh	rqball	2-5-97	3,000	null
2	update	Pittsburgh	rqball	2-6-97	40,000	4,000
3	delete	Erie	rqball	2-5-97	5,000	5,000
3	update	Lancaster	rqball	2-6-97	6,000,000	60,000

Table 2: Updated Materialized View DailySales

Once the maintenance transaction releases version 3 of the view, the view holder is allowed to read this version and store it as the latest version available. It is the job of the view holder to periodically monitor the data servers and/or data warehouses to know when updates or new versions have been created. Recall that the data warehouse and data servers are stateless and do not keep track of who is interested in their ongoing changes. The view holder can request a version of a view from the data warehouse by supplying a version number. The algorithm *UpdateVH* shown below describes how the TVLs are updated when a new version is released.

```
Algorithm UpdateVH ( )

Let VN = Latest_Version +1
SELECT operation, city, date, product, total_sales
FROM DailySales version VN
WHERE product = “rqball”
GROUP BY city, date

For each tuple T
    Find the TVL that matches the Group By
        attributes of T
    If found
        If TVL.LatestInUse = TRUE
            append TVL.Latest to Used
            set TVL.LatestInUse = FALSE
        End If
        If operation = “update”
            let Latest = (VN | total_sales)
        Else operation = “delete”
            let Latest = (VN | null)
        End if
    Else
        create new TVL
        insert key and non-updatable attributes
        let Latest = (VN | total_sales)
    End If
End for
Update the Latest_Version number to VN
```

When a MH starts an *application session* it must keep track of which version it is using for that particular session. If our MH now starts a new application such as the spreadsheet and graphing tool discussed, then it may want the *Latest* version (i.e., version number 3) maintained by its view holder. When this happens the latest version of each tuple becomes part of the *Used* linked list so that they are saved for future use and not overwritten. The flag *LatestInUse* associated with each *Latest* field in the TVL can be marked used (*U*) as each tuple is given to the MH if it is not already set. Therefore, the next time an update happens, new space must be allocated dynamically for the new *Latest* version of a tuple. Our TVLs now look like the right-hand side of Figure 3. We can see that the first TVL was not changed and, therefore, both version 1 and version 3 will see the same value for this tuple.

Notice that only versions 1 and 3 are maintained by the view holder, and version 3 is the latest. Version 2 can *not* be constructed using this view holder even though it is *still available* through the data warehouse. Version 2 was not needed by the MH and, therefore, its space was reclaimed within the view holder to hold and save version 3. Therefore, the view holder is a buffer for the MH and does *not* always store the versions found in the data warehouse. The last TVL of the right-hand side of Figure 3 demonstrates this since version 2's data, (2 | 60,000), is now no longer available to the MH.

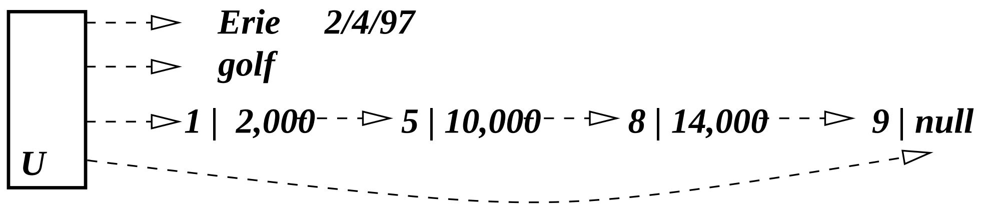
Whenever the MH wants the tuples for a particular application it must supply the version number (VN) it is interested. Then the linked list of *Used* is followed for each tuple until the $MaximumVersionNumber \leq VN$ is found. As a result, the *null* values obtained when a tuple is deleted must be explicitly placed within the TVL, otherwise, a previous version will be mistaken for the deleted tuple.

3.2 Communicating Changes in Versions

A MH can describe the view it is interested in by sending a select statement, such as the one used in the example, within an *initiation message* to its MSS. The fixed network will then assign this request to a view holder where the current version of this view will be gathered from the data warehouse and/or data servers. After the initiation, the MH can then perform queries by communicating with the view holder which will now buffer any versions necessary for as long as the MH may need them. The result of these queries or *cache messages* will be stored on the MH.

The view holder can build an indices tree structure for the TVLs attributes to provide faster access during repetitive updates and queries. If the MH also uses TVL structures to store cache messages it is possible to transfer any indices built in a view holder along with the cache message. This allows the MH to reconstruct the indices tree within its own memory.

Whenever the MH asks for the storage of a new version, the view holder can then in turn ask the data warehouse for its latest version if it has not already prefetched it. During the process of updating the view holder's TVLs, the exact differences between versions will be implicitly calculated since the TVL will contain a new tuple entry if there has been an update, and a null value if the tuple has been deleted. Therefore, at any time, the differences or Δ between any two versions contained within a view holder can be computed easily.



Latest Version number = 9
Versions 2, 4, 5, 6, 8, 9 are used by MH and kept by the view holder
Oldest available version is 2

Figure 4: One TVL storing Multiple Versions

For example, take the TVL shown in Figure 4. From the example we see that the difference between version 2 and version 8 is an update from the value $2,000$ to the value $14,000$. However, there is no difference in values between version 2 and version 4 since both of them use the tuple $(1 \mid$

2,000). The difference between version 2 and version 9 would be to completely delete the tuple and its contents from the view. Whenever the view holder needs to send a version or part of a version to the MH, it will use the algorithm *SendLatest* described below. When there is no change between the latest version of a tuple and the version already stored within the MH, a predetermined *no-change* token is sent.

Algorithm SendLatest (Latest_Version)

```
For each TVL
    If Latest.value $\neq$ the last element of Used.value
        let TVL.LatestInUse = TRUE
        send latest.value to MH
    Else
        send the no-change token
    End If
End for
append Latest_Version number to the
    list of available versions
send Latest_Version number to the MH
```

In the mobile environment, communicating only the changes between versions saves resources, especially when changes can be computed and incorporated easily, as in our case where we communicate tuples. If few changes have happened between versions, only the values for the tuples that have experienced change will be communicated to the MH. This is also useful upon reconnection when a MH may want to know the difference between the currently latest version of the view and the data it was able to work with while being disconnected, for example, as part of a data validation procedure.

Although we have concentrated on the view holder prefetching and maintaining data used by the MH, there are other possibilities to consider when deciding the exact role the view holder plays within the fixed network. These possibilities, which we alluded to in the introduction, arise from matching the placement and functionalities provided by the view holder with the capabilities of both the MH and the static hosts within the fixed network. *Essentially, the view holder can support any of these possible roles:*

- **Holder-as-Proxy:** The view holder only stores the latest version found in both the fixed network and MH in order to compute the Δ and build any required indices. All other versions are on the MH who has the adequate storage capabilities. In this case, the view holder's state can be made small within a MSS and possibly migrated as part of the hand-off between MSSs.

- **Holder-as-Buffer:** The view holder provides the same features as above. In addition, there is a full replication of the views currently in use (except the latest) on both the MH and view holder. Whenever a MH exhausts its resources while disconnected, it can suspend one or more of its active applications and reclaim the space of the respective versions. Later, when reconnected, these processes can finish with the view holder's copy of the required data.
- **Holder-as-Cache:** The view holder maintains a superview *SV* of the information required by the MH's application sessions. The MH can then hoard a subset of this data for use when disconnected. The MH has limited storage capabilities. Here the view holder must provide more functionality including some query processing and will most likely exist as a host integrated within the fixed network.

Which role the view holder assumes is decided by the preferences and capabilities of the MH. Thus, the view holder becomes a customizable client-oriented view maintenance mechanism. The MH specifies these customizations by using an extension to SQL that we proposed in [WLC98].

3.3 Deleting a Version from a View Holder

Once a MH completes the application sessions utilizing a particular version of a view, that view can be removed from the view holder. However, as we have seen, tuples from older versions that are not touched are used by the newer versions (see Section 3.2). Using the following proposition, we can develop a *garbage collection* procedure for deleting tuples from a TVL that are no longer useful.

Proposition 1 *If a tuple does* not *belong to a version i then it will* not *belong to any version j for all $j > i$.*

This proposition is useful when implementing the garbage collection routine since the versions of a view holder are placed in the linked list *Used* in order of their creation. Once we find a version number that does *not* use the tuple we want to delete, then we know it can be deleted safely since it will not exist in any other later versions.

As an example, consider the TVL of Figure 4. Notice that versions 2,4,5,6,8, and 9 are currently being used by the MH making version number 2 the oldest available version. Now let's suppose that the MH no longer requires version number 2. As a result, we see that version 4 will now become the oldest available version. The question for the garbage collection routine is whether or not it is safe to delete tuple (1 | 2,000). Recall that whenever we want to access a particular version of a tuple (VN) we have to make sure that the tuple with a $MaximumVersionNumber \leq VN$ is found. In order to find out if the tuple in question can now be deleted from the Used list, we first have to find the $MaximumVersionNumber \leq 4$. The result in this case is a $MaximumVersionNumber = 1$. Therefore, we can *not* delete the tuple (1 | 2,000) since the information it contains is still required by version 4 and possibly other versions. However, if the tuple had *not* been a part of version 4, the tuple could have been safely deleted since, by the proposition, we would be certain that no other later versions of this view would ever require this tuple.

4 An Implementation

Our prototype, currently developed in collaboration with the University of Cyprus, implements the concepts of view holders. This prototype supports database applications within a mobile environment by utilizing mobile Java agents, called *DBMS-aglets* developed at the University of Cyprus [PPS99].

4.1 Aglet Technology

An *Aglet* (agile applet) is a lightweight mobile Java object (approximately 2 kilobytes) developed at IBM Japan [AGLET98]. An aglet carries along its program code, state, unique identification, and query trip plan as it moves from host to host. If there are any communication problems, such as a host failure, the trip plan allows the aglet to try alternate hosts and solutions. A host interacts with an aglet by utilizing an aglet server program. This Java program listens for incoming aglets and provides a context in which they can resume their suspended execution. Figure 5, shows several aglets operating between two hosts A and B. Aglets can be created, talk to one another, move from host to host, and be disposed of at any time.

4.2 DBMS-Aglets: Mobile Java Agents for Database Access

If the aglets are equipped with database capabilities they become *DBMS-aglets* [PPS99], that can connect to a remote data server and collect data updates. Our DBMS-aglets are dispatched from the view holder in order to obtain new versions from the data sources. Once the DBMS-aglet arrives at

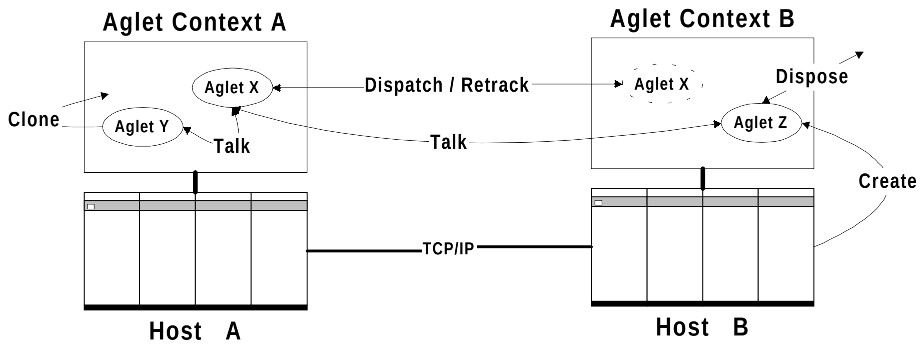


Figure 5: Aglet Life Cycle

the data server and passes any security checks, it then attempts to connect to the database. To do this it must load the correct Java Database Connectivity (JDBC) Driver. Currently, there are JDBC drivers for most desktop and distributed systems including drivers created specifically for Microsoft and Sun systems [H97]. With the correct driver the DBMS-aglet can then connect to the database and obtain the version requested by the view holder. The results can be dispatched back to the view holder while the DBMS-aglet continues its trip to the next host until released by the view holder (see Figure 6). At any time the view holder can dynamically adjust the DBMS-aglet's trip plan. After receiving updates from an issued DBMS-aglet, it is the view holder's responsibility to process the updates and present the MH with specific view data changes (i.e., the Δ view). By utilizing DBMS-aglets, a view holder can either move with the MH or remain stationary within the fixed network.

In a distributed database environment, processing queries reliably among various heterogeneous databases becomes a problem. To alleviate this problem, a CORBA system is usually invoked. Our prototype uses a CORBA system enhanced for mobile clients, called *Voyager*, developed in Java by

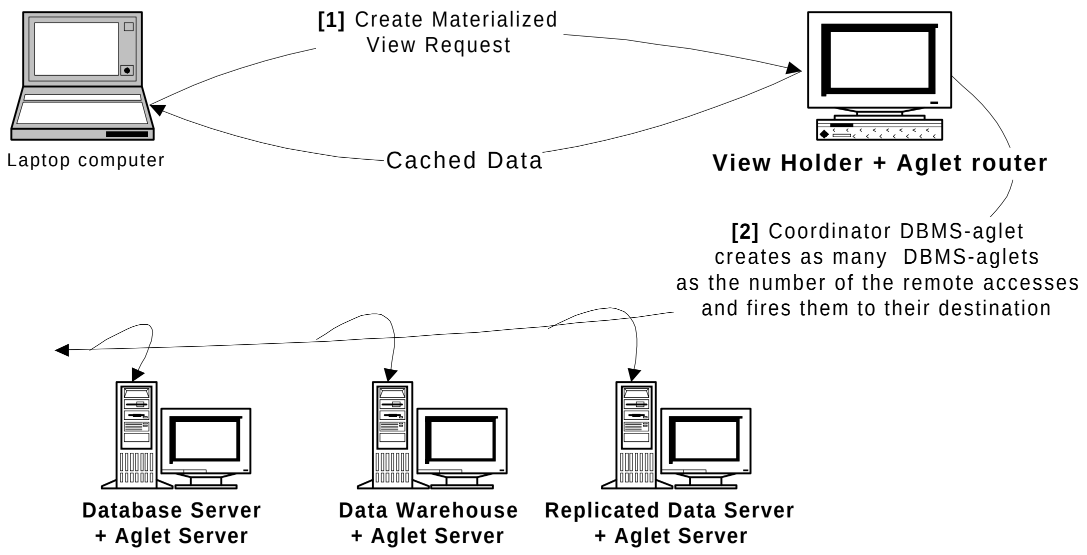

[1] Create Materialized View Request
Laptop computer
Cached Data
View Holder + Aglet router
[2] Coordinator DBMS-aglet creates as many DBMS-aglets as the number of the remote accesses and fires them to their destination
Database Server + Aglet Server
Data Warehouse + Aglet Server
Replicated Data Server + Aglet Server

Figure 6: Aglets for View Holders

ObjectSpace [OS98]. In addition, we are working on implementing view holders in conjunction with PRO-MOTION, our mobile transaction infrastructure, also under development using Java [WC97].

5 Conclusions and Future Work

This paper addresses the problem of caching/hoarding and maintaining data within a mobile environment in the form of a materialized view. Our main contribution is the development of a mechanism or *view holder* that maintains customizable versions of cached views. The view holder reduces the communication between the fixed network and the MH. This happens while increasing the functionality of view maintenance, for example, a MH now has the option to reclaim memory needed while disconnected, and when reconnected, allow processes to finish with the view holder's copy of the required data. Finally, an implementation using Java based mobile DBMS-aglets was described.

An extension of this work was the introduction of an extended form of SQL that allows the MH to specify (1) what *role* the view holder will play in its interaction with the MH, (2) which constraints determine how often view maintenance occurs or is communicated, and (3) which specific data changes are most important to the MH [WLC98]. Our future work involves allowing write transactions through the MH's materialized views. This must be done while maintaining consistency across multiple versions among many mobile clients.

Acknowledgments

We would like to thank Stavros K. Papastavrou and George K. Kyrou for their help with the implementation of this project.

References

[AAFZ95] S. Acharya, R. Alonso, M. Franklin and S. Zdonik. Broadcast Disks: Data Management for Asymmetric Communication Environments. In *Proc. of the 1995 ACM SIGMOD Int'l Conf. on Management of Data*, pp. 199–210, 1995.

[IVB97] T. Imielinski, S. Viswanathan, and B. R. Badrinath. Data on Air: Organization and access. In *IEEE Transactions on Knowledge and Data Engineering*, Vol. 9, No. 3, pp. 353–372, May/June 1997.

[AGLET98] Aglets Workbench. by IBM Japan Research Group.
Web site: http://aglets.trl.ibm.co.jp

[BDMW95] J. Bailey, G. Dong, M. Mohania, and X. Wang. Efficient Incremental View Maintenance Using Tagging in Distributed Databases. Technical Report 95-37, Univ. of Melbourne, 1995.

[CKLM97] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, and I. Mumick. Supporting Multiple View Maintenance Polices. In *the ACM SIGMOD Conf.*, 1997.

[DCVKK97] A. Datta, A. Celik, D. VanderMeer, J. Kim and V. Kumar. Adaptive Broadcast Protocols to Support Power Conservant Retrieval by Mobile User. In *Proc. of the 13th IEEE International Conference on Data Engineering*, pp. 124–133, 1997.

[H97] B. V. Haecke JDBC: Java Database Connectivity. chapters 1,2,3, pages 3-33. IDG Books Worldwide, 1997.

[IVB92] T . Imielinski, S. Viswanathan, and B. R. Badrinath. Querying in Highly Mobile Environments. In *the 18th VLDB Conf.*, pages 41-52, Aug. 1992.

[KLM$^+$97] A. Kawaguchi, D. Lieuwen, D. Mumick, D. Quass, and K. A. Ross. Concurrency Control Theory for Deferred Materialized Views. In *the 1997 ICDT Conf.*, Jan. 1997.

[KS92] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Sys.*, 10(1):3–25, Feb. 1992.

[KVP96] P. Krishna, N. H. Vaidya, and D. K. Prodhan. Static and Adaptive Location Management in Mobile Wireless Networks. *Jour. of Computer Comm.*, 19(4), Mar. 1996.

[MWC96] A. Massari, P. K. Chrysanthis, and S. Weissman. Supporting Mobile Database Access through Query by Icons. *Distributed and Parallel Databases Jour.*, 4(3), Jul. 1996.

[OS98] Voyager(tm) Technical Overview. by ObjectSpace.
Web site: http://www.objectspace.com/voyager

[PPS99] S. Papastavrou, E. Pitoura, and G. Samaras. Mobile Agents for WWW Distribued Database Access. In *the 15th Int'l Conf. on Data Engin.*, Mar. 1999.

[PSWCD97] A. Prasad Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Mobile Objects. In *the 13th Int'l Conf. on Data Engin.*, Apr. 1997.

[QW97] D. Quass and J. Widom. On-Line Warehouse View Maintenance for Batch Updates. In *the ACM SIGMOD Conf.*, pages 147-158, May 1997.

[WA97] R. Winter and K. Auerbach. Giants walk the Earth: the 1997 VLDB Survey. *Database Programming and Design*, 10(9):S2-S9+, Sept. 1997.

[WA98] R. Winter and K. Auerbach. The Big Time: the 1998 VLDB Survey. *Database Programming and Design*, 11(8), Aug. 1998.

[WC97] G. Walborn and P.K. Chrysanthis. Pro-motion: Management of Mobile Transactions. In *the 11th ACM Annual Symp. on Applied Computing*, Mar. 1997.

[Wid95] J. Ed. Widom. Special Issue on Materialized Views and Data Warehousing. *IEEE Data Engin. Bulletin*, 18(2), Jun. 1995.

[WLC98] S. Weissman Lauzac and P. K. Chrysanthis. Programming Views for Mobile Database Clients. *Proceedings of the Nineth International Workshop on Database and Expert Systems and Applications*, pages 408–413, Aug. 1998.

[ZGMHW95] Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *the ACM SIGMOD Conf.*, 1995.