Programming Views for Mobile Database Clients*

Susan Weissman Lauzac, Panos K. Chrysanthis
Department of Computer Science
University of Pittsburgh, Pittsburgh, PA 15260

Abstract

Within a database mobile environment, cached data on mobile clients can take the form of materialized views. In order to efficiently maintain such materialized views while taking into consideration disconnected operations, in this paper, we present the view holder, *a mechanism within the fixed network that maintains versions of views. Further, we propose an extension to SQL that enables the programming of the view holders by the mobile clients based on their preferences and capabilities and discuss their implementation.*

1 Introduction

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. The information stored can take the form of materialized views that operate in a fashion similar to data caches in distributed systems. Thus, the role of materialized views and view maintenance as a caching scheme is becoming increasingly important in the context of mobile database applications [4].

Specifically, there is a need for 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. In this paper, we present such a mechanism in the form of a proxy within the fixed network that can assume different roles in order to provide a customizable client-oriented "data warehouse" mechanism which we will call the *view holder* [13].

In contrast to its warehouse counterpart, a view holder is not as static or generic, and maintains a state with respect to the individual mobile clients it supports. When a view holder is required to maintain a particular view, the view specification can be seen as a program specification [11]. Further, a view holder can be programmed to maintain multiple versions of a view in order to compensate for the data changes that occured to the materialized views that were used during disconnection.

In the next section, by means of an example, we motivate the need of view holders. Section 2 describes the view holder within the context of the example, and then presents how the view holder allows for efficient interactions with the data sources as well as the mobile hosts. In Section 4, we extend the SQL `create view` statement and show how it can be used to program the view holders. In addition, we discuss their implementation.

2 Motivating Example

For the sake of generality, suppose that the database structure of our fixed network includes not only relational database servers responsible for storing base tables, but also a *data warehouse* which stores materialized views derived from these base table sources. It is very important to understand that this data warehouse holds *static* views that contain useful summary information which must be maintained periodically by the execution of a *maintenance transaction*. Queries from a client or mobile host (MH) can be answered with 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 by a versioning algorithm, such as *"two version no locking"* [10], where one or more versions may be available for the readers at any time. However, our data servers do *not* support any versioning mechanism.

2.1 The Data Servers and Warehouse

Our data servers contain base tables regarding some sporting goods stores (Table 1). The tables shown include `Items`, `Stores` and `Sales`, where `Sales` gives

*This work is supported in part by the National Science Foundation under grant IRI-95020091

Items			
itemid	iname	line	current_price
2	12" racquet	rqball	$30
3	instr. video	golf	$40

Stores			
sid	sname	city	manager
11	Dunham's	Pittsburgh	Ms. Balon
12	Dunham's	Erie	Mr. Dill
13	MC Sport	Pittsburgh	Mr. Patrick

Sales				
sid	itemid	quantity	current_price	date
11	3	10	$40	2/5/98
12	2	20	$30	2/5/98
13	2	40	$30	2/6/98
12	2	20	$30	2/20/98
12	3	10	$40	2/20/98

Table 1. Tables from the Data Servers

individual item transaction information. Our data warehouse, in this example, supplies summary information from the base tables and contains several separate versions of the data. One materialized view, called `TotalSales`, periodically totals the sales by store and item:

TotalSales(`tVN,sid,sname,itemid,line,Tsales`)

`tVN` keeps the version number of the maintenance transaction that last updated this tuple, since the data warehouse maintains multiple versions of the view there will be multiple versions of each tuple. The attributes `sid, city` and `itemid` are non-updatable attributes that do not change, whereas `Tsales` must be periodically updated by a maintenance transaction and will have different values among the versions. We will assume that the tuples with the largest `tVN` numbers belong to the most *current* version of the view. Table 2 shows a possible materialization for this view where two versions are available.

The maintenance transaction that created version 3 of the view does *not* include the last two sales transactions made by the store with `sid` 12 on 2/20/98. However, version 4 was created at a later time and includes these last two sales transactions.

Now suppose that a mobile host (MH) starts an application which will allow the user to see and perform some rough calculations regarding the racquetball equipment sales. In order to inquire about equipment sales made by stores in each city, the MH must request information from both the data warehouse and the data sources. So that each possible source is uniquely identified within the **RqballSales** query, *DW* refers to the data warehouse while *DB* refers to our source containing the base tables.

TotalSales					
tVN	sid	sname	itemid	line	Tsales
3	11	Dunham's	3	golf	$400
3	12	Dunham's	2	rqball	$600
3	13	MC Sport	2	rqball	$1200
4	11	Dunham's	3	golf	$400
4	12	Dunham's	2	rqball	$1200
4	12	Dunham's	3	golf	$400
4	13	MC Sport	2	rqball	$1200

Table 2. Data Warehouse's TotalSales View

Query: RqballSales
CREATE VIEW RqballSales **AS**
  SELECT DW.sname, DW.itemid, DW.line,
    DW.Tsales, DB.current_price
  FROM DW.TotalSales, DB.Items
  WHERE (DW.line = "rqball") **AND**
    (DW.TotalSales.itemid = DB.Items.itemid)

This global query will be sent to a query processor within the fixed network which will decompose it into component queries for both the data warehouse and the database. We want this query to be processed within the fixed network to save both the energy and resources of the MH. The results will then be communicated to the MH as a materialized view.

Suppose that when the MH requested this view it was materialized with respect to version 3 of the data warehouse's `TotalSales` view. The MH may keep its materialization of `RqballSales` for some time and may *not* receive the most current sales figures (i.e., from version 4) 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 their hotel room after work). Within the new application, the most recent sales figures can be incorporated into the spreadsheet. Once the MH receives the most recent version of the data, the user will be running two *application sessions* 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 [10].

Another difference to consider is that, in a data warehousing scenario, 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

The above example clearly illustrated the need to support versions in order to cope with (partially) disconnected operations. Even if the sources support versions, directly accessing the data servers and data warehouse poses a wide range of difficulties. Some of these problems stem from the limited resources of the MH and wireless networking conditions:

- In some versioning algorithms, once many maintenance transactions occur within the data warehouse 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* [10]. However, it is *not* always convenient or even possible for a MH to receive a new version of the view.
- 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.
- The MH may not have enough *adequate storage* to hold several versions of a materialized view or even one entire version.

Other problems stem from the static nature of the data warehouse and data servers:

- To update the MH, the data warehouse will have to send an entire new version of the view each time since it can *not* compute the difference between versions.
- Each time a MH's materialized view is to be updated the data warehouse and the data sources will have to be queried in order to reconstruct the materialized view from scratch.
- The data warehouse and data sources will *not* create or maintain a particular indexing structure, for example, for the query **RqballSales** that can be communicated to the MH.
- Each MH will be requesting information from the data warehouse and sources. This creates a great deal of traffic within the fixed network, especially if many MHs are requesting the same view query.

3 View Holders

To alleviate the problems discussed, *without* requiring modifications to the existing databases and data warehouses (i.e., unlike solutions presented in [1]), we developed a versioning mechanism, called the *view holder*, for maintaining the materialized views requested by a mobile host.

Every application on a mobile host uses only a subset of the data that exists in the data sources DS including our data warehouse. 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 sources ($SV = Q(DS)$).

For a specific application environment, the 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 [6] 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 ($SV' = SV + \Delta$), 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.

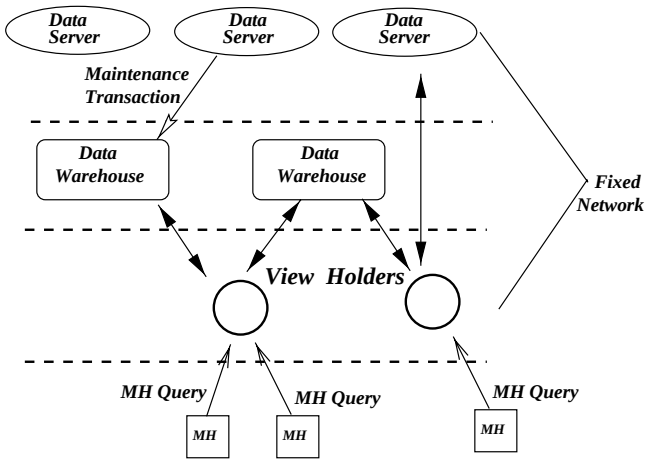

Figure 1. Overview of Architecture

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 hold-

ers. The data server layer is responsible for periodically constructing a maintenance transaction which updates a data warehouse where the views are static and the number of consecutive versions of each view also remains static.

In contrast, a version in the view holder will *never expire*, even if the data warehouse stops maintaining it, since the view holder must maintain a version for as long as a MH needs it. So, the view holder can be seen as a *buffer*, holding versions of a specialized view for a particular MH. This is done in order to compensate for the data changes that will occur to the materialized views being used during disconnection. Therefore, 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. Many versions can be dynamically maintained without incurring huge storage requirements because 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. View holders can be implemented via a data structure called the *tuple version list (TVL)*. These TVLs and the algorithms that maintain them and compute Δ are described in [13].

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 create an implicit "version" by querying the source in order to extract the data at a given moment. Often, by querying the catalog of a data source, we can determine the last time a tuple, attribute, or table was modified and use this as a timestamp to be stored along with the data. In addition, other structures, such as indices, can be built and maintained by the view holder and later communicated to the MH along with the materialization of the cache message. The MH would then be able to reconstruct an indices tree within its own memory.

3.1 View Holder Roles

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. *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 view and build any required indices.
- **Holder-as-Buffer:** There is full replication of the views currently being used on both the MH and view holder. Whenever a MH exhausts its resources while disconnected, it can now 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 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.

In the first case, *Holder-as-Proxy*, the view holder's state can be made small and possibly migrated as part of the hand-off between cells. In the case of *Holder-as-Buffer*, all the previous versions used by a MH are maintained as well as the latest in order to provide data replication, whereas in the third case, *Holder-as-Cache*, an even larger superview that may be used by the MH is prefetched and maintained. In the last case, the view holder must provide more functionality including some query processing and will most likely exist as a host integrated within the fixed network.

4 The Programming of Views

An unmaterialized view can be seen as a program specification. Every time the view "program" is executed, the view is re-materialized [11]. It is the view holder's responsibility to *monitor* the sources and rematerialize the data when changes have occurred, since the MH does not have the resources to do this work. The MH is only given the *pure data* that is not maintainable and does not contain derivation or maintenance procedures. As a result, the view holder must contain and execute a *materialization program*. However, this program does not have to be constructed without any input from the MH. Any additional information the MH can give the view holder regarding the host's work (e.g., important data to monitor, planned disconnections) can help customize and reduce the cost of materialization.

Therefore, in addition to allowing the user to state, within a select statement, the data it wishes to have maintained by the view holder, we want the user to be able to make very specific requests regarding how this data is maintained and communicated. These MH's preferences include: (1) what *role* the view holder will play in its interactions with the MH and, hence, the number of versions the view holder will have to maintain (2) which constraints determine how often view maintenance occurs or is communicated, and (3) which specific data changes are most important to the MH.

4.1 Mobile Host Preferences

Our goal is to extend SQL so that the **create view** statement sent within the initiation message includes the preferences of a MH with respect to issues (2) and (3) above. For our example in Section 2.1, we showed how a MH can request the services of a view holder by describing the data required and specifying from which sources the data should originate. Now we want to include the criteria for materialization that will describe which data changes should invoke the update of a MH. Towards this, we introduce the *ON* condition that can specify which data should be monitored by the view holder and how often. In our example, suppose that the mobile user wanted its view (within the view holder) to be updated only when there was a new version available from the data warehouse. That is, if the `current_price` attribute is updated a new version is *not* generated. Thus, the view holder only has to monitor the data warehouse for the release of new versions.

CREATE VIEW RqballSales **AS**
  SELECT DW.sname, DW.itemid, DW.line,
    DW.Tsales, DB.current_price
  FROM DW.TotalSales, DB.Items
  WHERE DW.line = "rqball" **AND**
    (DW.TotalSales.itemid = DB.Items.itemid)
  UPDATE ON DW.new_version

Besides the condition above which will update the view whenever the data warehouse generates a new version, this generic condition for determining materialization could also include:

- updating upon changes to a specified table or attribute. *(e.g., UPDATE ON DB.current_price, UPDATE ON DB.current_price > $15)*
- updating with changes to a specific data source. *(e.g., UPDATE ON DB)*
- updating with given keywords such as *ALL TABLES, ALL SOURCES.*
- updating after a given amount of time or after a specified number of versions have been released. This helps when planning a disconnection.
- updating with any combination of sources, attributes, and conditions. *(e.g., UPDATE ON DB.current_price > $15 OR DW.new_version)*

4.2 Building a View Maintenance Mechanism

There is a point, in our example in Section 2.1, where a maintenance transaction released version 4 of the data warehouse's view `TotalSales`. Once this occurred, the view holder was allowed to read this version and prefetch it for later use by the MH. But how did the view holder learn about the release of a new version? In order to answer this question there are several possible solutions to consider:

- **Monitor Data:** Make the view holder periodically monitor the data servers and data warehouses to known when updates or new versions have been created.
- **Monitor Catalog:** Have the view holder query the catalog and determine the last time a tuple, attribute, or table was updated. After comparing this with the timestamps of the data stored in the TVLs, the view holder can determine if there has been any changes.
- **Trigger:** Built a trigger within the data warehouse and server so that the sources notify a view holder when changes have occurred.

If the data server's constraint base can be manipulated, then once the view holder receives a request from a MH, a *trigger* can be constructed that will inform the view holder of any changes to the relevant data. On the other hand, if our data warehouses and servers are stateless and can not keep track of who is interested in their ongoing changes, then the triggering option can not be utilized. In this case, the two monitoring options still become valid possibilities that can be initiated by a view holder which creates a *loop* to monitor the data sources by periodically checking for updates. The choice is usually dependent on the type of permission or authorization granted by the data sources.

Figure 2 shows our options for view maintenance. Essentially, the *ON* condition given by the MH can be incorporated into a triggering or looping condition. Sometimes there is a *direct* translation from the *ON* condition to a condition for view maintenance. For example, if the MH only wants to be updated when there are changes to the attribute `current_price` then this condition can be *directly* incorporated into the *triggering* constraint. In the case of monitoring, a general timing condition could be used so that the attribute would be checked periodically. However, if the user wanted its materialization to be updated every 10 minutes independent of the changes to `current_price`, this type of condition could be *directly* integrated into the *monitoring* loop condition. In both cases, once updates had occurred, a select statement would be used to notify the view holder.

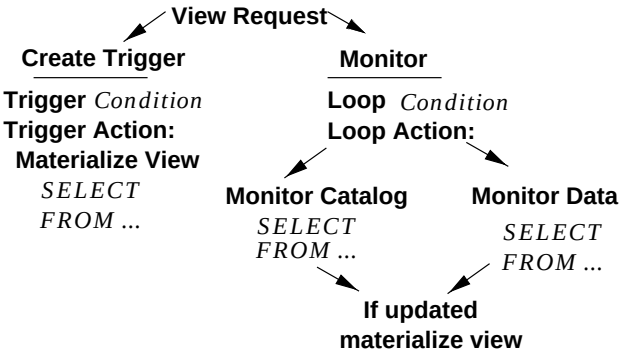

Figure 2. Programming View Maintenance

4.3 Implementation Issues

Our implementation approach for executing the monitoring program at the data sources is to utilize an *Aglet*. Aglets are mobile Internet agents in Java [3] that can be dispatched from the view holder and sent to a remote host for execution. If the aglets are equipped with database capabilities they become *mobile database agents* [8] that can connect to a remote data server, execute the monitoring program, and fetch any updates (see Figure 3). After receiving updates from an issued aglet, it is the view holder's responsibility to process the updates and present the MH with specific view data changes.

Aglets carry along their program code, state, unique identification, and query trip plan as they move from host to host. If there are any communication problems, such as a host failure, the trip plan allows them to try alternate hosts and solutions. Despite their multiple functionalities, aglets are small lightweight objects with a binary code of approximately 2 kilobytes. Once the aglet arrives at the data server and passes any security checks, it then attempts to connect to the database by loading the appropriate Java Database Connectivity (JDBC) Driver. Once the aglet is connected to a data server, it executes its monitoring of the database or catalog. The results can be dispatched back to the view holder while the aglet continues its monitoring until released or redirected to another data source by the view holder.

5 Conclusions

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 the *view holder*, a mechanism that maintains customizable versions of cached views specified by an extension of SQL. Currently, we are working on implementing view holders using Java agents in conjunction with PRO-MOTION, our mobile transaction infrastructure [12].

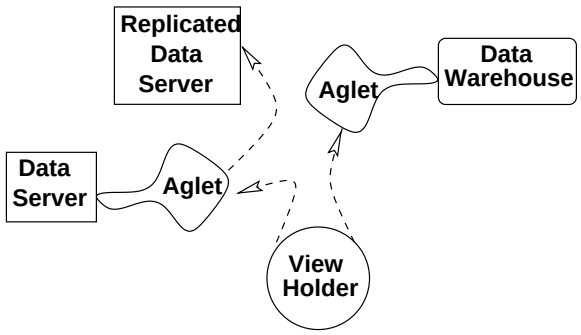

Figure 3. Monitoring Aglets

References

[1] J. Bailey, G. Dong, M. Mohania, X. Wang. Efficient Incremental View Maintenance Using Tagging in Distributed Databases. TR 95-37, U. of Melbourne, 1995.

[2] L. S. Colby, A. Kawaguchi, D. F. Lieuwen, I. Mumick. Supporting Multiple View Maintenance Polices. *the ACM SIGMOD Conf.*, pp. 405–416, 1997.

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

[4] T. Imielinski, S. Viswanathan, B. R. Badrinath. Querying in Highly Mobile Environments. *the 18th VLDB Conf.*, pp. 41–52, Aug. 1992.

[5] A. Kawaguchi, D. Lieuwen, D. Mumick, D. Quass, K. A. Ross. Concurrency Control Theory for Deferred Materialized Views. *the 1997 ICDT*, Jan. 1997.

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

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

[8] S. Papastavrou and G. Samaras. The Development of DBMS Client/Server Applications on the Web using Java Mobile Agents. TR 98-5, U. of Cyprus, 1998.

[9] A. P. Sistla, O. Wolfson, S. Chamberlain, S. Dao. Modeling and Querying Mobile Objects. *the 13th ICDE*, pp. 422–431, Apr. 1997.

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

[11] N. Roussopoulos. Materialized Views and Data Warehouses. *SIGMOD record*, 27(1):21–26, Mar. 1998.

[12] G. Walborn and P. Chrysanthis. Pro-motion: Management of Mobile Transactions. *the 11th ACM SAC*, pp. 171–181, Mar. 1997.

[13] S. Weissman Lauzac and P. K. Chrysanthis. Utilizing Versions of Views within a Mobile Environment. *the 9th ICCI*, Jun. 1998.

[14] J. E. Widom. Special Issue on Materialized Views and Data Warehousing. *IEEE DE Bulletin*, 18(2), Jun. 1995.

[15] Y. Zhuge, H. Garcia-Molina, J. Hammer, J. Widom. View Maintenance in a Warehousing Environment. *the ACM SIGMOD Conf.*, pp. 316–327, 1995.