View Propagation and Inconsistency Detection for Cooperative Mobile Agents

Susan Weissman Lauzac
Dept. of Mathematics and Computer Science
University of Puget Sound
Tacoma WA 98416, USA
slauzac@ups.edu
and
Panos K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
panos@cs.pitt.edu

No Institute Given

Abstract. Mobile agents are autonomous programs that migrate from one machine to another within a network on behalf of a client, thus, they are ideal for mobile computing environments since tasks can be delegated to mobile agents when a mobile client is disconnected. This paper extends the traditional functionality of a mobile service agent with capabilities that facilitate asynchronous cooperation among mobile database clients. In the context of mobile client-server database applications, data cached to support disconnected operations can take the form of a materialized view. We design mobile agents to reduce computation and wireless communication costs, and use view versioning to cope with disconnected operations by allowing application sessions to access current data without invalidating work previously done. A data validation or results propagation process detects inconsistencies with newer versions of data upon reconnection. Essentially, these mobile agents will compute the period of time or *consistency window*, measured in versions, for which the results of a mobile client's application are consistent. We supply rules that govern the creation and sharing of results and show how inconsistencies can be detected to offer a higher availability of data while organizing and gracefully degrading the amount of consistency achieved between the mobile clients and the data sources.

1 Customized Query Processing for Mobile Agents

In previous work, we explored the customization and localization properties of views in the context of mobile database environments to support disconnected query processing and developed a view maintenance mechanism called the *View Holder* [8, 10]. The core of the View Holder is a versioning mechanism that can adjust the currency of the data stored on the mobile client, for example, by allowing a user who was disconnected during a plane flight to later read updated derived data without necessarily discarding work performed on older data during the flight. In addition, a view holder is dynamic and stateful with respect to an individual mobile client, and therefore, it can respond to a mobile client's queries for information by communicating only the differences between answers, thus reducing the cost of wireless communication. In contrast to the materialized views maintained by a large, static, and stateless data warehouse, the View Holder can be thought of as a customizable client-oriented data warehouse, and it does not require modifications to be made to the existing data sources.

Delivering the results of queries in a mobile environment is different than in a traditional distributed environment due to the rapidly changing conditions of the wireless communication network, the requirements of the user in terms of the data's accuracy, and the cost the user is willing to pay for communication. Traditional query processing facilities are generally concerned with minimizing response time. By contrast, in a mobile environment, a user may want to introduce delays or change data accuracy in order to save service charges or to minimize required resources. Clearly, there is a need for devising ways by which mobile users can specify their choices for view maintenance and communication, in particular a *criteria for materialization* that describe which data changes should *invoke an update in the view holder agent.*

Instead of using a generic profile, it seems more natural to specify user preferences along with the definition of the view to be customized. Thus, we propose to extend SQL so that the **create view** statement sent within the *create view message* includes the view maintenance preferences of a cache agent (CA) residing on the mobile device. Towards this, we introduce the *ON* condition that can specify which data should be monitored by the view holder agent and how often. Essentially, the *ON* condition creates the customizable data currency, and summary required by the mobile client's application sessions (ASs).

1.1 Customized SQL Statements

The extended SQL `create view` statement offers additional but optional ([...]) clauses such as: Update On, and Maintenance.

CREATE VIEW <name of view> **AS**
  SELECT <attribute list>
  FROM <table list>
  [**WHERE** <selection and join conditions>]
  [**GROUP BY** <grouping attribute(s)>]

[**HAVING** <group condition>]
[**ORDERED BY** <attribute list>]
[**UPDATE ON** logical expression of pairs:
<condition for materialization[,Full or Partial] >]
[**MAINTENANCE** <Recomputational or Incremental>]

The MAINTENANCE clause specifies the view maintenance strategy, either recomputational or incremental, that should be used by the view holder agent. Finally, UPDATE-ON provides a logical expression of conditions for materialization. This generic condition for determining materialization over data servers (DSs) and data warehouses (DWs) includes *Update ON*:

- an individual attribute at DS1: *DS1.Items.price*.
- a condition on an attribute: *DS1.Items.price* > *$15*.
- a maintenance transaction commits at DW: DW.new_transaction.
- a given amount of time has passed: *10 minutes* or *DS1 10 minutes*. This helps the CA when planning a disconnection.
- a logical combination of all the above: E.g., *DS1.Items.price OR DW.new_transaction*; *DS1.Items.price AND DW.new_transaction*.

Although incremental view maintenance (e.g.,[12]) can be specified, the View Holder environment is different from the data warehousing environment. The amount of customized data requested by mobile clients is orders of magnitude less than what is available from a data warehouse, however, there can be many more mobile clients and their view holder agents within the fixed network. Recomputational view maintenance methods are more suitable under this scenario, since we can perform recomputation without having to capture every update performed at all the data sources because we will not be computing the Δ view during reconstruction. Natural periods of disconnection or weak connectivity allow recomputated results to be stored within the view holder agent. Recomputational maintenance is easier to implement and incurs fewer latencies since several rounds of queries are no longer required. Since small amounts of data are requested, the storage space required for intermediate results will also be orders of magnitude smaller than what is required in data warehousing environments. Furthermore, subqueries will be requested once and only once from each data source during the recomputation. Therefore, recomputation of the view within the fixed network is more appropriate when using DBMS-agents, since these agents can travel once to each data source transporting results and reconstructing a materialized view without having to backtrack and perform compensatory queries at sites already visited.

Essentially, the View Holder approach as shown in Figure 1 allows either recomputational or more complex incremental view maintenance to occur in the *fixed* network while only the Δ view (i.e., incremental maintenance) is communicated across the expensive *wireless* links to the cache agent residing on the

mobile device. A view holder agent computes the variations, or Δ view, between any two and possibly non-consecutive versions of a materialized view. The next section will provide an example of how the distributed query processing library routines can construct a materialization program to perform recomputational view maintenance when monitoring loops are used to evaluate the *ON* condition.

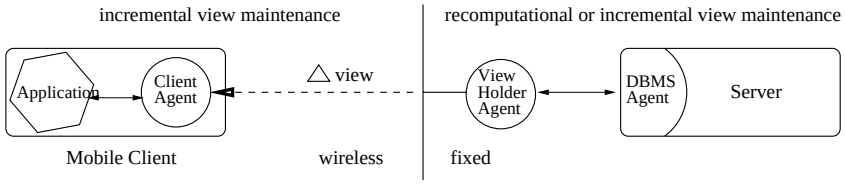

Fig. 1. View Maintenance in the Fixed and Wireless Networks

1.2 The View Holder's Materialization Program

How does the View Holder learn about relevant changes that occur at the data sources? There are three possible solutions:

- **Monitor Data:** Have the view holder agent's materialization program periodically query the relevant data sources to discover when updates or new versions have been created.
- **Monitor Catalog:** Have the materialization program query the database's catalog to determine from the last time a tuple, attribute, or table was updated if a relevant change has been made.
- **Trigger:** Build a trigger within the data warehouse and server so that the data sources notify the query processing facility of relevant changes.

Let us start by considering a query that reflects the join of data from tables r_1, r_2, r_3 of data servers $DS1, DS2, DS3$ respectively and requires monitoring at only one data source.

CREATE VIEW OneMonitor **AS**
  SELECT $DS1.a$, $DS2.b$, $DS3.c$
  FROM $DS1.r_1$, $DS2.r_2$, $DS3.r_3$
  WHERE $(DS1.r_1.b = DS2.r_2.b)$ **AND** $(DS2.r_2.c = DS3.r_3.c)$
    AND $(DS1.r_1.a < 5)$
  UPDATE ON $(DS2.r_2$, full)
  MAINTENANCE Recomputational

Table r_1	
A	B
1	2
7	2

Table r_2	
B	C
-	-

Table r_3	
C	D
3	4

Table 1. Tables for Query `OneMonitor`

Table 1.2 shows the state of the base tables that will be used to construct a materialized view from the query `OneMonitor`. Once this query reaches the view holder agent, if the MetaData maintained by the view holder agent does not contain information about the tables r_1, r_2, r_3, then this information must be obtained from the individual data sources. Once this information is gathered and stored, the query must be processed and a materialization program formed.

In this example, two DBMS-agents are necessary for the view recomputation. The *condition evaluation* DBMS-agent performs monitoring at $DS2$. The *view evaluation* DBMS-agent recomputes the view whenever a change to table r_2 occurs. To create this materialization program, a query that is expressed in a high-level query language such as SQL is represented internally as a structure known as the *query tree*. This query tree is used to create an execution strategy for accessing the data and obtaining the results. When the execution strategy is combined with a monitoring loop and condition evaluation, we call this a *materialization program strategy*. Once the execution strategy has been determined the *code generator* generates the code for executing the plan. It is this code that can be placed within a view evaluation DBMS-agent for execution at any data server.

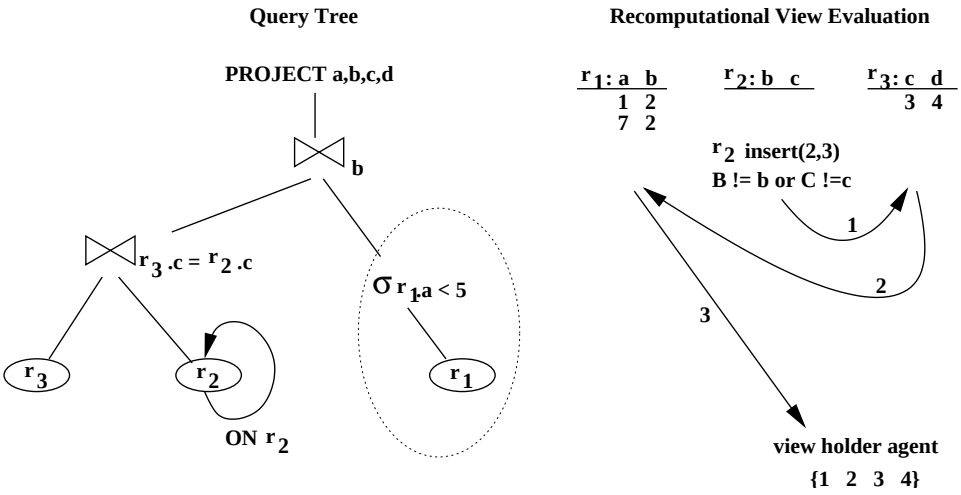

Fig. 2. Query Tree for Query `OneMonitor`

For our example view `OneMonitor` the query tree will appear as in Figure 2 (left-side). The query tree reflects the query trip plan used by the view evaluation DBMS-agent in order to perform the joins at the data servers. Once the *ON* condition is satisfied at $DS2$, the data of $DS3.r_3$ and $DS2.r_2$ will be joined (e.g., using a hash-join [2]) before this result is then later joined with the data selected from table $DS1.r_1$.

Often a query tree is built or modified to supply a more efficient strategy for executing a distributed query. One possible modification would be to know the approximate tables sizes from the MetaData maintained by the view holder agent. If the table size of $DS1.r_1 < DS3.r_3$ then the code generator would have wanted to perform the join between $DS1.r_1$ and $DS2.r_2$ first, and this strategy would have been reflected in the query tree in order to reduce the amount of data transferred across the fixed network.

Since each recomputed version of the materialized view is associated with the launching of a view evaluation DBMS-agent, we want these particular mobile agents to be processed by the data sources and view holder agent in the order they are launched. Therefore, we can associate a version number with each view evaluation DBMS-agent. At the data sources, mobile agents are buffered and executed in the order of their version number. Note that this method does not require the notion of a global time (unlike methods from [3]). since the version number is a locally maintained variable.

The condition evaluation part of the materialization program may require multiple condition evaluation DBMS-agents. For example, $cond_1$ *OR* $cond_2$ where $cond_1$ and $cond_2$ must be examined at separate sites, $DS1$ and $DS2$ respectively, would require a *coordinating* DBMS-agent to launch condition evaluating DBMS-agents that reside at $DS1$ and $DS2$. and supply the with the current version number.

When one of the conditions is satisfied, say at $DS1$, then the appropriate results from $DS1$ should be sent to the coordinating DBMS-agent with the current version number. The version number prevents the same materialized view version from being recomputed twice in the case where multiple condition evaluating DBMS-agents send their results to the coordinating DBMS-agent. In other words, although $cond_1$ and $cond_2$ require the use of two mobile agents, both mobile agents work toward the creation of the same view version.

Storing the results at the time the condition becomes true at $DS1$ allows the condition evaluation DBMS-agent to take a "snapshot" of the data server at a time when the *ON* condition was satisfied. For example, consider the *ON* condition $cond_1$ *AND* $cond_2$. Figure 3 shows which states of the data servers are combined to form a new state of the materialized view. Each time the condition is satisfied at a data server (for example, at $DS1$) a new version of the materialized view will be started and completed later by the coordinating DBMS-agent. However, only once the condition is also satisfied at the remaining site ($DS2$) can a new version of the materialized view be created by a view evaluation DBMS-agent. It is important to remember that there may be other remote data servers that are not part of the *ON* condition and yet are still part of the re-

computation of a materialized view. The view evaluation DBMS-agent may still have other remote data servers to visit according to its query trip plan. In [9], we defined *recomputational consistency* based on the snapshots incorporated from the data sources and introduced new levels of materialized view consistency to better characterize the mobile view currency customizations available.

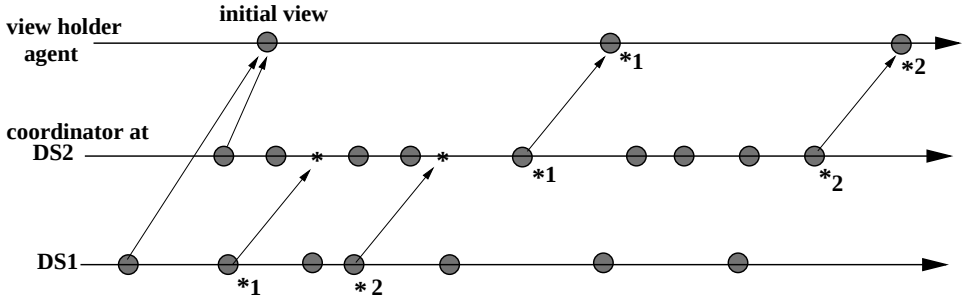

Fig. 3. Evaluating the Conjunction of Two *ON* Conditions

2 Materialized Views for a Data Processing Chain

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. Crampton
12	Dunham's	Erie	Mr. Prunty
13	REI Sport	Pittsburgh	Mr. Atkins

Sales

sid	itemid	quantity	sales_price	date
11	3	10	$40	2/5/01
12	2	20	$30	2/5/01
13	2	42	$30	2/6/01
12	2	20	$30	2/20/01
12	3	10	$40	2/20/01

Table 2. Tables from the Data Servers

Our data servers contain base tables regarding some sporting goods stores (Table 2). The tables shown include `Items`, `Stores` and `Sales`, where `Sales`

gives individual item transaction information. Now suppose that a user are going to begin a business activity (e.g., rough calculations, contract, graph). Recall that our view holder agent will contain materialized views derived from the base tables, and maintain several separate versions of each view [10]. 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 transaction that last updated this tuple. The attributes `sid, city` and `itemid` are non-updatable attributes that do not change, whereas `Tsales` must be periodically updated and will have a different value among the versions. We will assume that the tuples with the largest `tVN` numbers belong to the most *current* version of the view. Table 3 shows a possible materialization for this view where two versions are available since the *ON* condition was satisfied and either recomputational or incremental view maintenance was performed.

TotalSales					
tVN	sid	sname	itemid	line	Tsales
3	11	Dunham's	3	golf	$400
3	12	Dunham's	2	rqball	$600
3	13	REI Sport	2	rqball	$1260
4	11	Dunham's	3	golf	$400
4	12	Dunham's	2	rqball	$1200
4	12	Dunham's	3	golf	$400
4	13	REI Sport	2	rqball	$1260

Table 3. View Holder's TotalSales View

Note that the view 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 2/20/01. Suppose a MC requested a query regarding racquetball equipment sales and the result was materialized with respect to version 3 of the `TotalSales` view. The MC may keep its materialization of the view for some time and may *not* receive the most current sales figures (i.e., from version 4) due to traveling or communication delays. Later, another application such as a spreadsheet and graphing tool could be started. 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). Now, within the new application, the most recent sales figures can be incorporated into the spreadsheet. This shows how versions help cope with disconnected operations, by allowing applications to access more *current* data without invalidating work previously done.

Each *application session (AS)* running on a MC only reads view data derived from the same consistent state at the data sources. In other words, each AS is associated with one view state vs_i from the view holder agent's view state sequence, VS_{seq}. However, once the MC receives the most recent version of the

data, the client will be running two ASs and accessing two separate versions (or view states) of the view *at the same time*.

3 Results in a Data Processing Chain

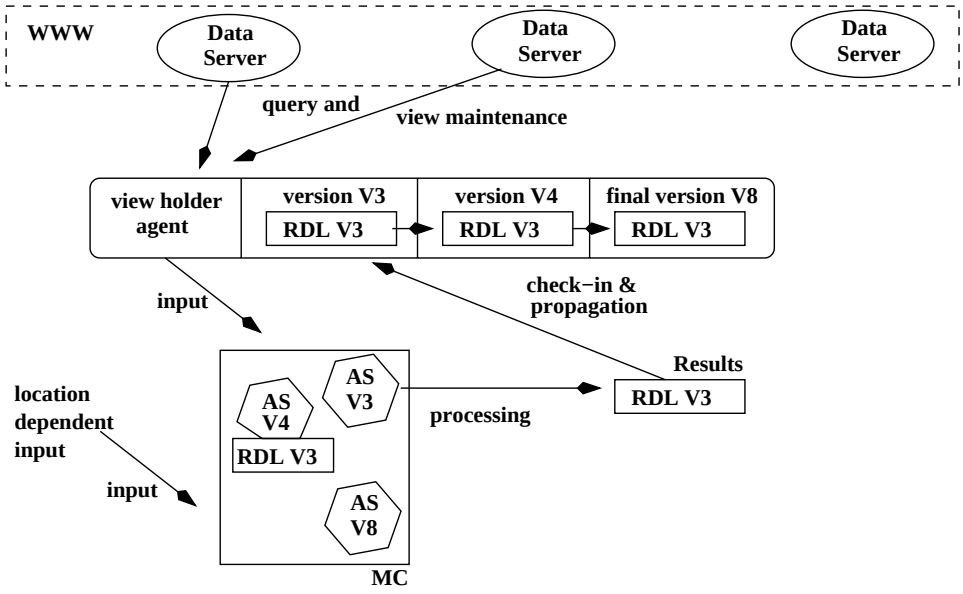

Fig. 4. Mobile Client Data Processing Chain

During disconnection and MC can execute ASs that perform work on a subset of the data cached from the view holder agent's view without having to lock data at the sources or even at the view holder agent. While working a MC can try various computations or solutions involving the data and then later, upon reconnection or improved communication conditions, it can attempt to integrate the results of the ASs within the view holder agent.

Figure 4 shows how this data processing chain works and is created within the view holder agent. Version V_3 of the materialized view was requested as input to an AS. This input was possibly combined with information from the geographical location of the mobile device in order to produce the *result data layer* RDL_V_3. By caching the input, the mobile client was able to do this work even while disconnected. Later, during periods of quality connectivity, this RDL can be stored in the view holder and *coupled with the derived materialized data* used as input in producing these results, while additional application sessions can be started using newer versions or view states as their input.

Furthermore, the integration process will allow RDL_V_3 to *propagate* further and become associated with successive view states that are consistent with the

results. An RDL is considered consistent with a successive view state, vs_i if vs_i could have been as input to create the RDL. The versions over which a result data layer can be integrated is called the *consistency window* of a RDL. Eventually these results can be sent to a results database or a results process such as a graphing tool, and then archived or stored within the remote data servers for future access.

Although we have discussed the reading of data, write transactions on base data could still originate from an AS, but these transactions are *only* performed directly with the data sources and not through the materialized views stored on the mobile client and the view holder. Changes to the base data will be inconsistent with the current version of the derived data used by an AS, therefore, an AS should write to the base sources only if these inconsistencies can be tolerated.

As stated earlier, a version of the data in the view holder agent will *not expire* even if the data sources stop maintaining it. Instead the view holder must maintain a version for as long as an AS needs it. So, the view holder agent can be seen as a *buffer*, holding versions of a specialized view for a particular AS and its results (RDLs). Therefore, even if a RDL does not propagate it can still be consistently read along with the view state used.

4 Result Propagation and Inconsistency Detection

So far, we have described how view holders agents maintain and communicate multiple versions of materialized views to overcome the limitations of MC disconnection from the data sources, and thereby, increase the concurrency of mobile client reading. In addition, versions were created whenever *ON* conditions are satisfied to prefetch updates that are of interest to a user and thus customize the currency of the AS reads. Essentially, the data currency and consistency for one AS is provided by the view holder without a MC having to stay in contact with the stateless data sources. This gives the MC more options when disconnected, for example, whenever a MC exhausts its resources, it can now suspend one or more of its active applications and reclaim needed space, then later when reconnected, these ASs can finish with the view holder's copy of the *results* stored along with the original version of the derived data used to create these results. If a new AS is started then this AS can still see the results or RDLs stored, as long as these results are consistent with the new version of the derived view being used as input to the new AS. In this way, ASs can propagate their work to new sessions. In this section, we will show how the $\Delta view$ between versions of the materialized tuples read by the MC can be used to detect inconsistencies.

4.1 The View Holder Results Propagation Model

Figure 5 shows conceptually how the view holder agent maintains the results for a MC. When a MC begins its work it requests a specific amount of data from the sources to be maintained by the view holder agent as a materialized view.

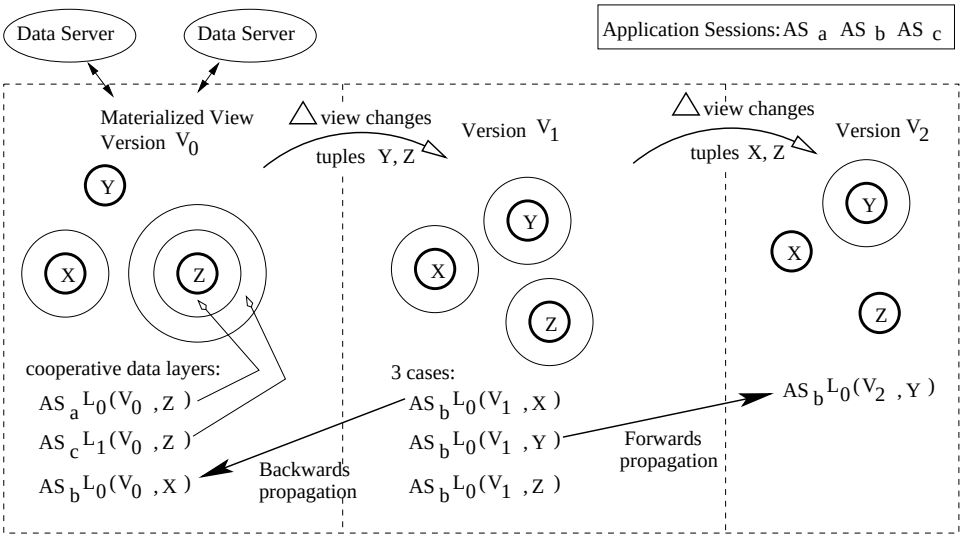

Fig. 5. The Result Data Layers

Subsets of this view can be cached, although how much data is stored depends on the storage capacity of the mobile machine. Individual ASs will read tuples from this cached subset as they perform their work, and the set of *core* tuples an AS reads from a view state of the materialized view is what we call the read set C.

In addition, each AS executing on a MC has the ability to produce results that will become part of the view state. The data one AS produces is the AS's RDL. For an AS_a operating on a MC and reading from version V_0 of the view holder agent's materialized view, we denote the RDL L_i produced as, $AS_a L_i(V_0, C)$, where C is the set of tuples read from V_0 that were used to create the AS's results. When a MC reconnects, the data layers created during disconnection are submitted for integration. A layer is integrated along with the version of data read and used to produce the layer. Whether or not these integrated results are finally committed and passed on to a results process or database is discussed in Section 4.2. We can think of a RDL as the *creation of a materialized view derived from the tuples of one version* of a view holder agent's view.

In the figure, version V_0 is created from the data sources and contains three tuples `X, Y, Z`. Application sessions, AS_a and AS_c operating on the MC request data from tuple `Z`, and create results that will be stored in the RDLs, $AS_a L_0(V_0, Z)$ and $AS_c L_1(V_0, Z)$ where AS_a integrated its results first. At this point, an *ON* condition was satisfied, and a new version of the data appears in the view holder as version V_1. The $\Delta view$ will contain the set of tuples that are different between versions V_0 and V_1. It will show what changes were made to `Y` and `Z`. AS_b can read tuples from version V_1, but what happens during integration

of the results depend on the contents of the $\Delta view$ between versions. Consider the following three cases from Figure 5:

- AS_b reads `X`: In this case the resulting RDL exists in both version V_0 and V_1 since `X` is not a part of the $\Delta view$, in other words, `X` has not changed. We call this the *backward propagation* of data layers.
- AS_b reads `Y`: In this case the resulting RDL exists in version V_1 and will exist in V_2 when it is created since the tuple `Y` is not modified between version V_1 and version V_2. This is called *forward propagation*.
- AS_b reads `Z`: In this case the resulting RDL exists only for version V_1 and does not propagate because tuple `Z` is part of every version's $\Delta view$.

4.2 Types of Consistency Windows

Optimistic replication of data from the view holder agent to the mobile machines offers a high degree of data availability. It is this availability that becomes crucial in allowing a mobile client to work individually, especially during periods of disconnection. However, the *cost* of availability in the mobile environment comes in the form of data consistency. We can no long guarantee that work performed by a mobile client will always be consistent with later versions of a materialized view. As new data and versions of a materialized view are created, we make more current data available to the application sessions, but we do so at the expense of creating possibly even more inconsistencies.

The goal of our results propagation model is to be able to get the most we can from the results generated by a mobile client, in other words, to offer a higher availability of data while organizing and gracefully degrading the amount of consistency achieved between the results created by the mobile client and the data sources. Here, we are referring to the consistency between a version *already* constructed and stored within the view holder agent and the results produced by a mobile client. How much consistency is desirable depends on the purpose of the work performed. How well a MC may be able to achieve a desirable consistency and store its results depends on the availability and cost of the wireless network and the capabilities of the mobile client. We divide the creation of RDLs within a view holder agent into two consistency categories:

Application-specific consistency window: Work performed while disconnected on older data is still valid as long as it is consistent with a specific amount of data from within the view holder agent. This scenario is useful whenever results need to be generated over a period of time. For example, if each grouping of versions represented a month of sales figures (e.g., September, October, November), then results integrated for each month could still be used to generate earnings graphs and analysis. In this scenario, the *application's consistency window* would be the versions spanning one month of sales figures, and we would want a RDL to integrate over all versions within the window. If a RDL *cannot* be integrated over the span of the application's consistency window

then it must be *aborted* and recalculated. As soon as an RDL integrates over the application's consistency window, it is considered committed.

Final version consistency window: This is the scenario where *all* RDL consistent windows must include the final version. When the mobile client has completed its individual work, it must be able to store its results as RDLs that are part of the *final* version of the view holder agent's view. As long as a MC's results can be forward propagated to the final version, it will not need to have its application session aborted or be forced to run a reconciliation procedure. Therefore, when a MC integrates its work, it does so as a *conditional commit* until the final version of the view holder agent is created. However, the further these results are able to propagate towards the final version the less reconciliation would be required. In this scenario, the consistency window for a RDL spans the initial version used by the AS to the final version stored for the MC. This is useful for any data processing chain where data computed at a mobile location can be integrated at the view holder agent, and if found to be valid used for final processing and storage.

4.3 Rules for Result Propagation

Given our results propagation model, we will now provide four rules for sharing, and propagating the RDLs created by the mobile client's ASs. Since the results can be considered a materialized view derived from a view holder agent's tuples, the granularity for detecting inconsistencies happens at the tuple level, so first we must define the set of tuples from which the results are derived. This set of tuples for a RDL is called the *core dependency set:*

Definition 1. Core Dependency Set: *An AS can read: (1) a set C of core tuples from version V_j of the view holder agent and/or, (2) a set of result data layers, where each layer L_i is derived from a core tuple set C_i of version V_j. Therefore, the set of core tuples that an AS's results or RDL are derived from is the set*
$CoreD = C \cup \bigcup C_i$.

- **1. Forward Propagation:** Suppose an AS's results are derived from the set $CoreD$ from version V_j of the view holder agent. The RDL produced by the AS can be forward propagated to version V_k $(k > j)$ if for all versions i, where $k > i \geq j$, $CoreD() \cap \Delta view_i = \emptyset$.
- **2. Backward Propagation:** Suppose an AS's results are derived from the set $CoreD(V_j)$ of the view holder agent. The RDL produced by the AS can be backward propagated to version V_k $(j > k)$ if for all versions i, where $j > i \geq k$, $CoreD() \cap \Delta view_i = \emptyset$.

Now we need two additional rules for integrating a RDL:

- **3. Integration:** When a new RDL is integrated, there should be an attempt to forward and backward propagate this layer to all other versions of the view.

– **4. New View Version:** When a new version V_k is created, there should be an attempt to forward propagate all RDLs associated with V_{k-1} to V_k.

What happens when a new RDL can *not* forward propagate to the latest version of the view depends on what kind of work is being done and what type of consistency window is required:

– **Application-Specific Commit Rule:** A RDL commits if it successfully forward and backward propagated over the application's consistency window, otherwise it must abort or be reconciliated.
– **Final Version Conditional Commit Rule:** Let V_k be the final version of the view holder agent's materialized view after the mobile client has integrated its RDLs. A RDL commits if it successfully forward propagated to V_k, otherwise it must abort or be reconciliated.

4.4 LTVs and Propagation

The Logical Tuple Versions (LTVs) of a view holder agent operate at the granularity of the materialized view's tuples. Figure 6 shows the scenario where an AS has requested version 3 of the materialized view while later two additional ASs are started using the latest version 4. In order to read a version V of the materialized view, the cache agent residing on the mobile device will receive the largest version number available that is less than or equal to V for each tuple in the view holder agent.

View Holder for VN = 3

Key	Non−updatable	Updatable	Count
Dunham's itemid 2	rqball $30	3 | $600	1
REI Sport itemid 2	rqball $30	3 | $1,260	1

View Holder for VN = 4

Key	Non−updatable	Updatable	Count
Dunham's itemid 2	rqball $30	3 | $600 4 | $1,200	1 2
REI Sport itemid 2	rqball $30	3 | $1,260	3

Fig. 6. Logical Tuple Versions (LTVs)

Storing the RDLs within the view holder agent's LTVs allows layers to be automatically propagated forwards or backwards between versions of tuples. LTVs do this by implicitly calculating from the $\Delta view$ which versions of a view's tuple are consistent with a particular AS's RDL. For example, Figure 7 shows a

possible scenario for the building of RDLs from the LTVs previously shown. As shown in the LTVs for version 3, suppose a MC request this version, and its AS is started. If this application session reads the `REI Sport` tuple and creates some sales results, then this work, $AS_a L_0(V_3, REISport)$, will be associated with the tuple `REI Sport` within version 3.

Once changes have occurred at the base table in the data sources, the view holder agent will build version 4 of the view incorporating the changes to the `Dunham` tuple by increasing the value of `Tsales` from \$600 to \$1,200. After version 4 is created, two more ASs are started, AS_b and AS_c. However, since the tuple `REI Sport` was *not* part of the $\Delta view$, a MC reading version 4 of this tuple from the LTVs will actually be reading the unchanged tuple from version 3 (i.e., the largest version number less than or equal to 4 for tuple `REI Sport` is version 3). Conceptually, this implies that any data layers associated with the tuple `REI Sport` are automatically propagated forward to version 4.

From Figure 7 we see that within version 4, AS_b integrates data layer $AS_b L_0$ $(V_4, Dunham's)$, while AS_c reads the tuple `Dunham's` and the propagated RDL $AS_a L_0$ $(V_4, REISport)$ in order to produce its own result data layer, $AS_c L_0(V_4,$ $\{Dunham's, REISport\})$. Note that since AS_c reads from AS_a its $CoreD$ set must now contain both the `Dunham's` and `REI Sport` tuples. This happens by the definition of the *Core Dependencies Set*, AS_c's results are derived from its own core set C and the core set of the RDL $AS_a L_0(V_4, REISport)$. Therefore, the set $CoreD =$ { `Dunham's` } $\cup$ $C_{AS_a L_0}$, where $C_{AS_a L_0} =$ { `REI Sport` }. This ensures that if work done by AS_a does not propagate forward, then neither will AS_c's RDL.

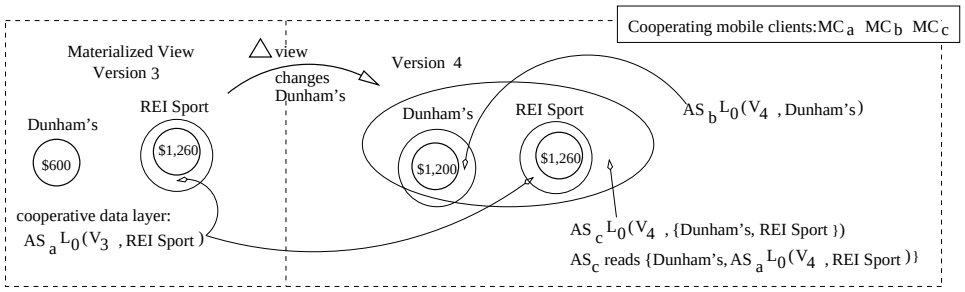

Fig. 7. Result Data Layers for Figure 6

5 Utilizing RDLs for Asynchronous Cooperation

The DVS prototype as described in [7] is a multi-tiered system architecture where the View Holders' components mediate between the data sources and the mobile clients. The **Dictionary_Vagent** keeps track of all the materialized views being maintained by the **View_Agents** and their locations. The **View_Agents** create

materialized views from the various data servers, perform view maintenance, and execute queries on these views. By having a **View_Agent** also store the results created from the mobile clients utilizing a view as described in this paper, the **View_Agent** becomes a cooperative work repository and *cooperation facilitator* so that ASs executing on *different* mobile clients can share the RDLs produced.

Some asynchronous cooperative environments that have been described in the literature, such as CoAct [4] or Coda [5], employ optimistic replication strategies where each client has their own copy of the shared data they require. Later, a synchronization process allows the client's work to be integrated with the work of others while providing conflict detection and/or the reconciliation of conflicts that occurred due to concurrent accesses done to the various replicas. In the View Holder approach, the shared data is the materialized view created from the remote databases for the mobile clients. Optimistic replication happens when a mobile client requests a copy of a version of the materialized view. Once RDLs are created they can be sent to the view holder agent (i.e., **View_Agent**) in order to validate the RDLs and detect inconsistencies. If an RDL becomes integrated and coupled with version(s) of the materialized view, then another mobile client can read these results along with a version from the RDL's consistency window.

Producing Cooperative Work

Asynchronous cooperative work varies between periods of individual MC processing and periods of joint work [4]. During periods of individual work, MCs can run ASs that perform work on a subset of the data cached from the view holder agent's view without having to lock data at the sources or at the view holder. During these individual work periods, users can try out various computations or solutions involving the data, whereas during joint work the users make their results available to the other cooperative users [4].

Figure 8 shows how cooperative work is created within the view holder agent. During a period of individual work, our mobile client may cache one or more versions of a materialized view as described throughout this paper and produce results such as graph data points. Later, during periods of joint work, these results can be stored in the view holder agent *and coupled with the derived materialized data* used in producing these results (RDLs). These results can become shared among the users and eventually sent to a results database or a results process such as a graphing tool. The cooperative clients do *not* update the remote databases themselves, but perform all integration and sharing through the view holder agent.

View Holders maintain versions of views to allow for greater flexibility and customization in the amount of data currency and consistency achieved between the views cached on the mobile computer and the data formed within the cooperative work repository. Essentially, we extend the traditional functionality of a server-side proxy with capabilities that facilitate cooperation among the users by providing:

- **Flexible Data Currency:** The view holder agent is a mechanism where we can adjust the currency of the data stored on the mobile client, for example,

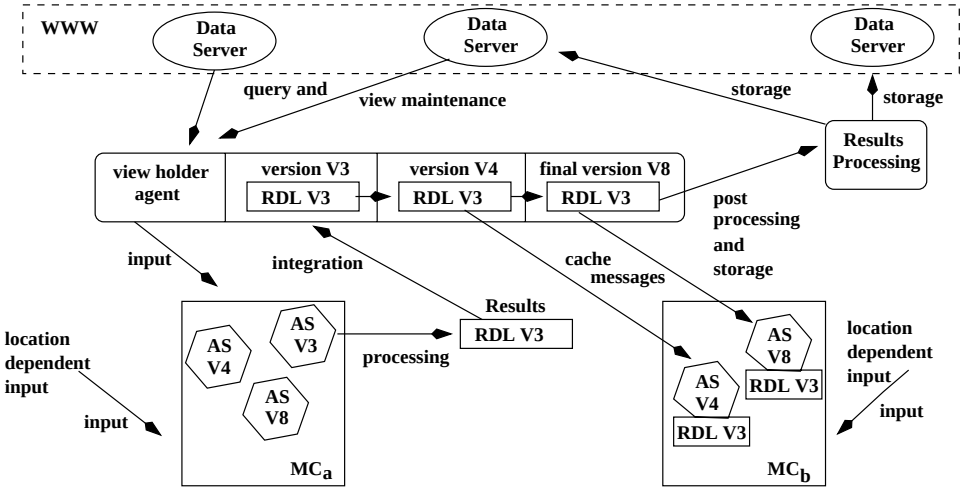

Fig. 8. Integrating and Sharing RDLs During Cooperation

so that an already working client or a new client joining the cooperative effort will be able to read newly updated derived data and results previously produced (i.e., a new version of the view and its respective RDLs). We use versioning not only to create an optimistic asynchronous cooperative strategy that allows individual work to be performed while disconnected, but also to supply changes in data currency.

- **Cooperative Work Consistency:** As individual results of the newly synthesized cooperative work are created, there is a need for providing a process that can integrate the work done by mobile clients and check for data inconsistencies. Essentially, we will be adding individual results to the cooperative work repository only if the states of the cooperative work or RDLs are consistent with the original derived version used as input, and the RDLs can propagate throughout the consistency window. Looking at our example 7 again, we can now consider AS_a, AS_b, and AS_c as application sessions executing on different MCs. The integration process within the LTVs and the rules for integration and sharing remain the same.

6 Conclusion

Within a mobile environment, we have shown how data cached can take the form of a materialized view and described a server-side agent mechanism for the fixed network or *view holder agent* that maintains versions of the views that are required by a particular mobile client's application sessions. View Holders are designed to reduce computation and wireless communication costs, and use view versioning to cope with disconnected operations, by allowing application sessions to access more *current* data without invalidating work previously done. Result propagation and inconsistency detection allow work performed during

disconnection to be integrated within a view holder agent along with the original data used as input to create the results. The results are considered valid as long as they can be integrated within a consistency window of versions (i.e., view states) as required by the application. Rules were supplied that govern the creation and sharing of these result data layers among a group of application sessions possibly executing on different mobile clients and showed how inconsistencies can be detected within the LTVs of the view holder agent before sharing is allowed to proceed.

References

1. B.R. Badrinath, A. Fox, L. Kleinrock, G. Popek, P. Reiher, and M. Satyanarayanan. A Conceptual Framework for Network Adaptation. *IEEE Mobile Networks and Applications*, 5(4):221–231, 2000.
2. R. Elmasri and S. B. Navathe. Fundamentals of Database Systems. chapters 18-21, pp. 501–633. Benjamin-Cummings, 1989.
3. R. Hull and G. Zhou. A Framework for Supporting Data Integration using the Materialized and Virtual Approaches. In *the ACM SIGMOD Conf.*, pp. 481–492, Jun. 1996.
4. J. Klingemann, T. Tesch, and J. Wasch. Enabling Cooperation among Disconnected Mobile Users. In *the 1997 COOPIS Conf.*, Sept. 1997.
5. J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Trans. on Computer Sys.*, 10(1):3–25, Feb. 1992.
6. D. Quass and J. Widom. On-Line Warehouse View Maintenance for Batch Updates. In *the ACM SIGMOD Conf.*, pp. 393–404, May 1997.
7. C. Spyrou, G. Samaras, E. Pitoura, S. Papastavrou, and P. K. Chrysanthis. The Dynamic View System (DVS): Mobile Agents to Support Web Views. In *the 17th Int'l Conf. on Data Engineering*,pp. 30–32, 2001.
8. S. Weissman Lauzac and P. K. Chrysanthis. Programming Views for Mobile Database Clients. In *Proc. of the Nineth Int'l Workshop on Database and Expert Sys. and Applications*, pp. 408–413, 1998.
9. S. Weissman Lauzac and P. K. Chrysanthis. Personalizing Information Gathering for Mobile Database Clients. In *Proc. of the 10th ACM Symp. on Applied Computing*, Mar. 2002.
10. S. Weissman Lauzac and P. K. Chrysanthis. Utilizing Versions of Views within a Mobile Environment. *Journal of Computing and Information*, 1998.
11. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View Maintenance in a Warehousing Environment. In *the ACM SIGMOD Conf.*, 1995.
12. Y. Zhuge, H. Garcia-Molina, and J. Wiener. Consistency Algorithms for Multi-Source Warehouse View Maintenance. *Distributed and Parallel Databases Jour.*, 4(4), 1997.