# Personalizing Information Gathering for Mobile Database Clients*

Susan Weissman Lauzac
Dept. of Mathematics and Computer Science
University of Puget Sound
Tacoma WA 98416, USA
slauzac@ups.edu

Panos K. Chrysanthis
Dept. of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260, USA
panos@cs.pitt.edu

## ABSTRACT

Mobile agents are ideal for mobile computing environments because of their ability to support asynchronous communication and flexible query processing since tasks can be delegated to mobile agents when a mobile client is disconnected. This paper explores the use of mobile agents in personalizing information gathering for mobile database clients. Personalized data take the form of materialized views and personalization is provided in the form of view maintenance options. These options, expressed using an extended SQL Create View command, offer a finer grain of control and balance between data availability and currency, the amount of wireless communication and the cost of maintaining consistency. The paper defines recomputational consistency and introduces new levels of materialized view consistency to better characterize the mobile client view currency customizations.

## Keywords

Mobile Computing, Mobile Agents, Materialized Views, Mobile Databases,Data Consistency and Currency

## 1. INTRODUCTION

The size of today's database and data warehousing environments as well as the Internet's ability to provide vast amounts of information has shown that today's users need better ways of handling what is available. *Customization or personalization* of information gathering for mobile clients is becoming increasingly important due to the computing, communication, and storage differences among mobile devices and the amount of information available.

In database systems, *views* provide a mean to present different users with different portions of the database based on the users' perspective. Within relational database systems, a view defines a function from a subset of base tables to a derived table and is *materialized* by physically storing the tuples in the derived table. In client-server configurations, *materialized views* can be stored at the clients to support local query processing and operate in a fashion similar to data caches [12]. In previous work, we explored the customization and localization properties of views in the context of mobile database environment to support disconnected query processing and developed a view maintenance mechanism called the *View Holder* [10, 11, 9].

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, requiring no modifications to be made to the existing data sources.

Because the View Holder combines and computes the necessary derived data, it is also able to offer different levels of view consistency between the data available and the derived data given to the clients. The contribution of this paper lies in providing an understanding of how view consistency is affected by mobile client view currency customization. Two types of view maintenance algorithmic approaches are examined, (1) *recomputational maintenance* that constructs an entirely new version of a materialized view, and (2) *incremental view maintenance* that allows updates to be slowly incorporated within an existing version based on the data warehousing Strobe algorithms [13]. The paper defines *recomputational consistency* and introduces new levels of materialized view consistency which correspond to specific view currency customizations. Furthermore, it explains how view maintenance is achieved by constructing a materialization program utilizing mobile agents [2, 5, 8].

The next section introduces our extension to the SQL **create view** statement for specifying user preferences. Section 3 examines the different choices for customizing view currency and realizing a materialization program using mobile agents. Section 4 formally discusses view consistency and Section 5 concludes with a summary.

*This work has been partially supported by NSF award IIS-9812532.


*ACM SAC* '02 Madrid, Spain

## 2. PERSONALIZED DATA ACCESS

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 accuracy of data, 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, for example, 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 *criteria for materialization* that describe which data changes should *invoke an update in the view holder*.

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 includes the view maintenance preferences of the submitting application 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). 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 change occurs at a specific DS or table: *DS1* or *DS1.Items*
- any change: ALL TABLES, ALL SOURCES
- a maintenance transaction commits at DW: DW.new_transaction.
- a given amount of time has passed: *10 minutes* or *DS1 10 minutes.* This supports plan disconnection.
- a specific number of versions: DW AFTER 3 versions.
- a logical combination of any of the above: E.g., *DS1.Items.price OR DW.new_transaction*; *DS1.Items.price AND DW.new_transaction.*

### 2.1 Customized SQL Statements

The extended SQL **create view** statement offers three additional but optional ([...]) clauses: **Update On**, **Role** 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] >]
[**ROLE** <Holder-as-Proxy, Holder-as-Buffer,
or Holder-as-Cache>]
[**MAINTENANCE** <Recomputational or Incremental>]

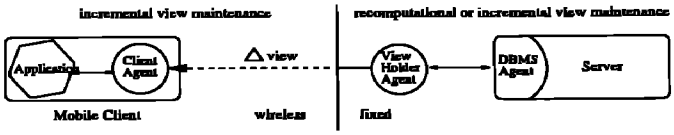


**Figure 1: View Maintenance in the Fixed and Wireless Networks**

The MAINTENANCE clause specifies the view maintenance strategy, either *recomputational* or *incremental*, that should be used by the view holder agent. The ROLE clause specifies how many versions and how much data must be maintained. The options range from *Holder-as-Proxy* where only the most recent changes are maintained, to *Holder-as-Buffer* where a larger portion of the data is prefetched for later use. If no role is specified then the default is the role Holder-as-Proxy since it is the least expensive option. Finally, as we discussed above, UPDATE-ON provides a logical expression of conditions for materialization. For incremental view maintenance only the conditions are required. For recomputational view maintenance, there is an additional option corresponding to either *full or partial* recomputational maintenance (e.g., *(DS1.current_price, partial) OR (DW.new_transaction, full)*). Both full and partial view maintenance will be defined in Sections 3.1 and 3.3, respectively. If no option is provided then full recomputational view maintenance is the default.

## 3. CUSTOMIZING VIEW CURRENCY USING MOBILE AGENTS

The view holder is a middleware component developed to provide beneficial options for communication and computation in *both* the fixed and wireless networks. Within the wireless network, a view holder acts as the data source for the view materialization by a mobile client whereas within the fixed network, it acts as a (customized client-oriented) data warehouse integrating data from multiple data sources. Thus, a view holder combines two view evaluations when executing a customized create view statement.

The best choice for a materialization mechanism across the expensive wireless network is clearly incremental view maintenance. The view holder as the only data source computes and communicates to the mobile client the variations, or Δ view, between any two and possibly non-consecutive versions of a materialized view. On the other hand, the choice of the view maintenance mechanism in the fixed network that involves multiple data sources is not obvious. In particular, we are interested in better understanding the choices in a Java-based mobile agent infrastructure [7] as shown in Figure 1. DBMS-agents are mobile-agents that can connect to a remote data server and invoke database operations [6].

With incremental view maintenance algorithms, such as the Strobe algorithms presented in [13], *all updates* performed at the data sources are reported to a view maintenance mechanism. This mechanism is then responsible for querying other data sources and learning which corresponding changes must be made to the materialized view. In order to avoid *multiple source anomalies* due to the latency inherent to receiving answers, once the computation of the view changes begins, new updates that occur at the data sources must be taken into account and compensated.

Compensatory actions may lead to additional queries.

Recomputational view maintenance takes a view's specification and completely recreates the view from scratch. A relevant subquery is performed *once* at each data source and the results are combined in order to re-build the materialized view. Once a new materialized view is created it can be compared with the older materialization in order to learn what exact changes occurred during view maintenance. Since recomputational maintenance does *not* require the exact changes from the data sources and does *not* compute the $\Delta$ view during reconstruction, multiple source anomalies *cannot* occur.

Incremental view maintenance is more suitable for a data warehouse environment where the volume of data is large (several terabytes) and there is limited off-line time that prohibits the running of a very long recomputational view maintenance transaction. Incremental view maintenance allows a data warehouse to slowly absorb incoming updates and incrementally modify its materialized views without having to block readers for long periods of time. Its major cost is in the requirement of possibly several rounds of compensating queries.

The View Holder environment is different from the data warehouse environment. The amount of customized data requested by a mobile client is orders of magnitude less than what is available from a data warehouse. Further, off-line times are longer because each view holder supports typically a single user and because of the natural periods of mobile users' disconnections. Thus, in contrast to data warehouses, recomputational view maintenance transactions are expected to be of short duration, have small storage requirements for intermediate results and execute within long off-line time. Under these circumstances, recomputational view maintenance methods are more suitable. These same circumstances hold when using mobile agents to implement view materialization. Further, recomputational maintenance leads to mobile agents with small footprint since it is easy to implement and incurs fewer latencies since mobile 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.

But how does the View Holder agent learn about relevant changes that occur at the data sources to perform recomputation of a view? An SQL query expressing a customized materialized view specification produces a materialization program with two basic components: *view evaluation* that computes the new view and *condition evaluation* that triggers a new view materialization. The selection of the method for evaluating an *ON condition expression* is not limited by the use of DBMS-agents. All three possible methods can be used with mobile agents and their applicability only depends on the data source capabilities.

- **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 view holder agent's 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 when relevant changes have occurred.

| *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`**

In the next section, we will provide an example of how the distributed query processing library routines can construct a materialization program to perform recomputational view maintenance when monitoring data are used to evaluate the *ON* condition.

## 3.1 View Holder's Materialization Program

Let us start by first 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. Table 1 shows the state of the base tables that will be used to construct a materialized view from the query `OneMonitor`.

**CREATE VIEW** OneMonitor **AS**
**SELECT** $DS1.a$, $DS2.b$, $DS3.c$
**FROM** $DS1.r_1$, $DS2.r_2$, $DS3.r_3$
**WHERE** $(DS1.r_1.a < 5)$ **AND**
$(DS1.r_1.b = DS2.r_2.b)$ **AND** $(DS2.r_2.c = DS3.r_3.c)$
**UPDATE ON** $(DS2.r_2$, Full)
**ROLE Holder-as-Cache**
**MAINTENANCE Recomputational;**

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. There are three types of DBMS-agents that can be used when constructing a materialization program:

- **coordinating DBMS-agent:** Both parts of the materialization program can be contained within one coordinating DBMS-agent. The coordinating DBMS-agent will reside at a data source that needs to be monitored, and dispatch other condition evaluation DBMS-agents to the data sources that need to be monitored. Once the *ON* condition is satisfied, the coordinating DBMS-agent launches a view evaluation DBMS-agent. For every version that is created, a coordinating DBMS-agent can only dispatch **one** view evaluation DBMS-agent.
- **view evaluation DBMS-agent:** Whenever a new version of a materialized view must be constructed, the view evaluation DBMS-agent is responsible for traveling to each data source, executing the appropriate subquery, and performing any necessary joins.
- **condition evaluation DBMS-agent(s):** For every data source that must be monitored for the *ON* clause, a condition evaluating DBMS-agent can be dispatched to reside at the data source and notify the coordinating DBMS-agent when the *ON* condition has been satisfied and a new version must be constructed.

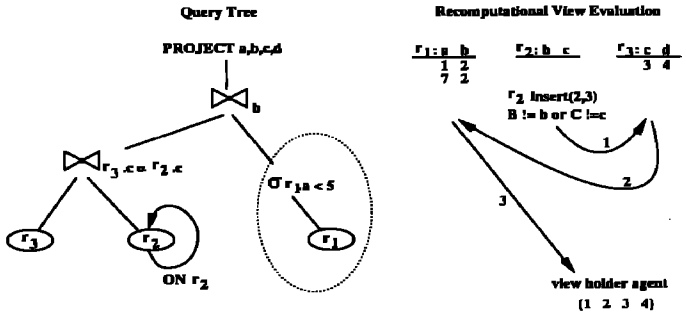


**Figure 2: Query Tree for Query `OneMonitor`**

Since, in our example, there is only one data server to monitor, we can create a materialization program for a coordinating DBMS-agent that performs monitoring and then launches a view evaluation DBMS-agent to recompute the view whenever a change to table $r_2$ occurs. To create this materialization program, a query expressed in SQL must first be scanned, parsed and then validated. The scanner identifies language tokens, while a parser and validater check the syntax of the query to determine if it is grammatically correct. Then an internal representation of the query known as a *query tree* is created. 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 [3]. It is this code that can be placed within a DBMS-agent for execution at any data server.

For our example view `OneMonitor` the query tree will appear as in Figure 2 (left-side). The execution of a query tree consists of first executing the internal node operations whenever its operands are available and then replacing the internal nodes by the table that resulted from executing the operand. The query tree of Figure 2 reflects the query trip plan used by the one 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 [3]) 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. Most current distributed query processing algorithms consider the goal of reducing the amount of data transfered during execution to be the optimization criteria when choosing a distributed query execution strategy [3]. 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.

With the code generated using the help of the distributed query processing library, the view holder agent could build and launch the coordinating DBMS-agent necessary for evaluating the *ON* condition that must also contain the one view evaluation DBMS-agent necessary for recomputing the materialized view of the `OneMonitor` query. The complete materialization program is shown in Figure 3.

```
Coordinating DBMS-agent ()
BeginBody
  Query DS2.r2
    SELECT DS2.r2.b, DS2.r2.c
    FROM DS2.r2

    Let B = b and C = c
    Let local variable version number v = 0

    Start view recomputation:
      Supply query trip plan (query tree) with query
        DS.r2 results.
      Launch view evaluation DBMS-agent with query
        trip plan, and version number v = 0.

  Begin Monitoring:
    Every (Monitor_time)
      SELECT DS2.r2.b, DS2.r2.c
      FROM DS2.r2

      If (B ≠ b or C ≠ c)
        Let version number v = v + 1
        Start view recomputation:
          Supply query trip plan (query tree) with query
            DS.r2 results.
          Launch view evaluation DBMS-agent with query.
            trip plan and current version number v.
        Let B = b and C = c
      End If
    End Every
EndBody
```

**Figure 3: Recomputational Materialization Program for the *OneMonitor* View**

Since each recomputed version of the materialized view is associated with the launching of a view evaluation DBMS-agent, we want these 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, since the version number is a locally maintained variable of the coordinating DBMS-agent.

The materialization program presented for `OneMonitor` reflects the worst case scenario where a brute force method is used to perform condition evaluation by comparing the values of $b$ and $c$ tuple by tuple. However, a special database differential utility might be helpful in reducing the cost of comparison. For example, if deletions are not permitted, a differential can be detected by just remembering the total number of tuples in a table or count (*) and comparing this value with the current count(*) result. In this case, just one arithmetic operation is required in order to determine if the view evaluation DBMS-agent should be launched. Also, it should be noted that there is no need to retrieve non-updatable attributes.

## 3.2 Coordinating *ON* Condition Evaluation

In general, the condition evaluation part of a 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 residing at $DS2$ to launch a condition evaluating DBMS-

agent that would reside at $DS1$. The coordinating DBMS-agent provides the condition evaluating DBMS-agent with the version number $v$. 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 $v$. 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 $v$.

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. This allows the materialized view to be built such that the *ON* condition is *satisfied for version v*. For example, consider the *ON* condition $cond_1$ *AND* $cond_2$, Figure 4 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 $DS1$ a new version of the materialized view will be started and sent to the coordinating DBMS-agent at $DS2$. However, only once the condition is satisfied at $DS2$ can an entire version of the materialized view be created by a view evaluation DBMS-agent with a satisfied *ON* condition.

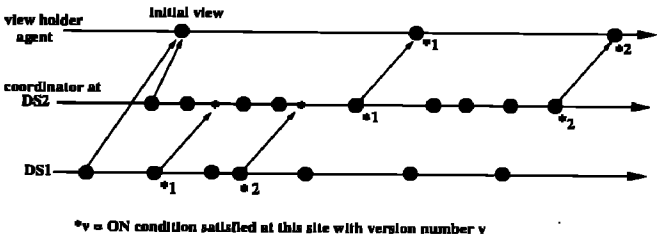


**Figure 4: Evaluating the Conjunction of Two *ON* Conditions**

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 recomputation of a materialized view. Once the *ON* condition is satisfied, the view evaluation DBMS-agent may still have other remote data servers to visit according to its query trip plan.

## 3.3 Partial Materialization and View Currency

We have seen how information from the user's customized **create view** statement can specify when view maintenance should occur. The information contained in the *ON* condition can also be used to avoid the full recomputation of a materialized view whenever the condition is satisfied. For example, suppose the *ON* condition is used to specify which data is of the most importance to the user. In our query `OneMonitor`, the condition *UPDATE ON* $DS2.r_2$ reflects that changes to the table $r_2$ hold more interest for the user and their application session than other updates. In order to process and deliver these changes faster, a *partially materialized* version of the view could be created by combining the changes detected by the condition evaluating DBMS-agent at $DS2$ with the data already stored at the view holder agent *without* performing an entire view recomputation. Since the condition evaluation is separate from the recomputation, this would require a message containing the new updates to be delivered from the condition evaluator DBMS-agent to the view holder agent. The tuples of the version that is to be directly updated would be changed and only these specific variations would be communicated, thus saving even more wireless bandwidth than what is required by a full recomputation of the view.

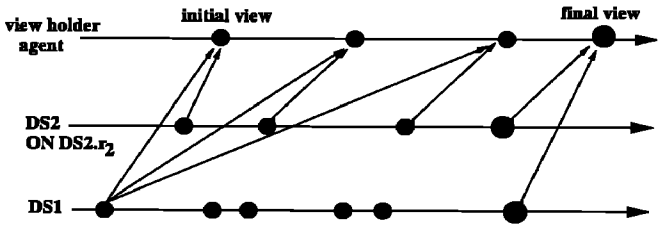


**Figure 5: Partial View Materialization**

Figure 5 shows an example of partial materialization. This allows the user and application sessions to receive specific information they will be using with less data transfer over the wireless communication links, only a partial view recomputation, a faster response time, and fewer mobile agents. At any time, the recomputation part of the materialization program can be activated, so that the view is fully recomputed with values from all the data servers involved in the view's specification. This could be done periodically in order to create view states that correspond to the state of the data sources and this is shown in the final convergent view created in Figure 5.

# 4. CUSTOMIZING VIEW CONSISTENCY

Since the mobile client may enter periods of weak connectivity or disconnection in addition to other limitations such as battery power, the view holder agent is responsible for acquiring and storing the result of a query in the form of a materialized view. However, the view holder agent must also provide some guarantees regarding view consistency and, therefore, mobile client cache consistency. As the communication capabilities vary in both the fixed and wireless networks, we want to *customize* the level of view consistency seen by an application session, so that view maintenance operations match the preferences of the user and their applications. Application-specific access and consistency allow mobile applications to trade consistency guarantees for communication costs improvements.

We have seen how the *ON* condition effects view (data) currency. Now, our goal is to understand which levels of materialized view consistency are applicable with the View Holders approach when recomputational view maintenance is used.

In the context of incremental view materialization, the analysis of the Strobe algorithms with two data sources revealed that incremental mechanisms can offer four levels of materialized view consistency. These levels, in order of their difficulty to guarantee, are: *convergence, weak consistency, strong consistency*, and *completeness* (serializability). In addition, the level of consistency reachable for a particular view maintenance algorithm is dependent on the update scenarios (i.e., single update (SUT), source-local (SLT), or global (GT)) of the various data sources involved in the view's specification.

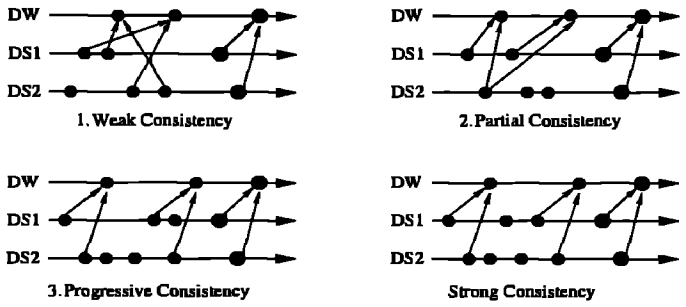


**Figure 6: New Levels of View Consistency for Recomputational View Maintenance**

In understanding how view consistency is affected by mobile view currency customization when recomputational view maintenance is used, we discovered two additional view consistency levels that we need to take into consideration. The first, called *partial consistency* is analogous to the view consistency that can be achieved when partial materialization is used as described in Section 3.3. The second, called *progressive consistency* represents the view consistency that is achieved when the view is fully recomputated every time the *ON* condition is satisfied. Figure 6 shows these new levels and how they relate to the snapshot consistency levels defined in [14]. The new levels are between weak consistency and strong consistency. For a View Holder accessing two data sources this is a complete list of possible consistency levels. In the rest of this section, we formally define these consistency levels.

For a particular data source, we can define the *source state sequence* [13], $S_{seq} = ss_0, ss_1, ...ss_{final}$, to be the base data states after each update commits. $VS_{seq} = vs_0, vs_1, ...vs_{final}$ is the state sequence of *view states* created after each view recomputation is performed. Individual view states of $VS_{seq}$ are stored as versions within the view holder agent whenever a mobile client's application session requests a particular state. A *source state vector*, $ssv[]$, is the state of all source data at a particular moment in time and represents the contents of the base tables. If there are $u$ sources where each source has a unique id $x$, then at a given moment in time a source state vector $ssv$ contains $u$ elements such that the $x^{th}$ component of the vector or $ssv[x]$ contains the visible state of source x.

View state $vs_j$, is *snapshot consistent* with a source state vector, $ssv_i$, if $V(vs_j) = V(ssv_i)$ [14]. We can now define partial and progressive consistency as forms of view consistency slowly advancing from weak to strong consistency. Note that what constitutes a transaction depends upon the source update scenarios.

DEFINITION 1. **Weak Consistency:** *Convergence holds and each view's state $vs_i$ reflects some valid source state for each data source. In other words, for the source state vector $ssv_j$ such that $V(vs_i) = V(ssv_j)$, each data source $x$ executed a serial schedule of transactions $X = t_1, t_2, ..., t_n$ that created $x$'s entry into the source state vector.*

DEFINITION 2. **New Partial Consistency:** *For a view's state $vs_i$, let each data source's entry in the source state vector be denoted by $vs_i = ssv_i = [ds_{i_1}, ds_{i_2}, ..., ds_{i_u}]$ for each data source $ds_{i_x}$ $1 \leq x \leq u$.*

*Partial consistency means that weak consistency holds and there exists a data source $ds_{i_j}$ such that if $vs_i < vs_l$ then $ds_{i_j} < dsl_j$ and $ds_{i_z} = ds_{l_z}$ for all $z \neq j$.*

DEFINITION 3. **New Progressive Consistency:** *For a view's state $vs_i$, let each data source's entry in the source state vector be denoted by $vs_i = ssv_i = [ds_{i_1}, ds_{i_2}, ..., ds_{i_u}]$ for each data source $ds_{i_x}$ $1 \leq x \leq u$.*

*Progressive consistency means that weak consistency holds and there exists a data source $ds_{i_j}$ such that if $vs_i < vs_l$ then $ds_{i_j} < dsl_j$ and $ds_{i_z} \leq ds_{l_z}$ for all $z \neq j$.*

DEFINITION 4. **Strong Consistency:** *Weak Consistency holds and the order in time of the view's states matches the order of the corresponding source states. Let the source state vector $ssv_k = [ds_{1_k}, ds_{2_k}, ..., ds_{u_k}]$ contain the state of all the data sources after a given set of transactions $T_k$ have been executed. Then there exists a mapping $m$ from view to source states after the execution of $T_k$ transactions such that:*

1. *Each view's state $vs_j$ reflects a valid source state, $m(vs_j) = ssv_k$ for some $k$.*
2. *If $vs_j < vs_l$, then $m(vs_j) < m(vs_l)$.*

For all three levels: weak, partial, and progressive, each view's state must reflect a valid state at each data source. In addition, there may be a different schedule of updates at each data source and the view may reflect a different set of committed transactions at each data source. This is true even during progressive consistency when updates may occur at a data source after it has been visited during the query trip plan (see number three of Figure 6).

However, when comparing different recomputed view states we can now make some guarantees regarding the currency of the data made visible during the query trip plan. Progressive consistency provides the guarantee that each successive view state contains more data currency than its predecessor. Partial consistency reflects *ON* condition evaluation and recomputation *without* the entire query trip plan being executed once the condition is satisfied. Progression consistency reflects full recomputational view maintenance. At any time, due to the separation of condition and view evaluation within the materialization program, full recomputation can be activated or de-activated. This will change the level of view consistency from subtransactionspartial to progressive view consistency if the data sources are operating within the SUT or SLT update scenarios. Weak consistency is still applicable in the mobile agent scenario if, for example, mobile agents are possibly allowed to read older versions of data at a particular data source (e.g., reading old versions from a data warehouse running 2VNL). However, we will assume for this rest of this paper that mobile agents read the most recent attribute values available from a data source.

## 4.1 Customized Recomputational Consistency

Without new levels of view consistency, customized data access by monitoring or triggering on individual data sources would require the expense of strong or complete consistency algorithms executed across all the update transactions performed at all the data sources. Recomputational maintenance does *not* require a collection of the exact changes that happened at each data source. If we now examine the SUT and SLT update scenarios when recomputational view maintenance occurs, then we can define *recomputational consistency* and reason about the *customized recomputational*

*correctness.* Recomputational consistency explains why the view state sequence created makes guarantees to ensure that monitored updates at the data sources are chronologically reflected in the view states.

A mobile agent recomputing a view will create a new view state, $vs_i$, by visiting each data source and performing a subquery from the view's specification. The contents of a data source $DS_x$ used during view evaluation is the $x^th$ component of the source state vector $ssv_i$. We assume that every entire view recomputation is associated with a monotonically increasing sequence number, and the sequence numbers are processed by the data sources and view holder agent in order.

DEFINITION 5. **Customized Recomputational Consistency:** *Let current[] be the source state vector reflecting the current state of the data sources. A view state sequence $VS_{seq} = vs_1, vs_2, ..., vs_{final}$ reflecting mobile agent recomputational view maintenance is consistent if there exists a function derived: view state $\longrightarrow$ source state vector such that $derived(vs_i) = ssv_i$ and,*

- *Convergence holds: $V(vs_{final}) = V(derived(vs_{final}))$ This means that the final view state is derived from the final state of each data source.*
- *Weak Consistency holds: $\forall i, V(vs_i) = V(derived(vs_i))$ and each data source $x$ executed a serial schedule of update transactions $X = t_1, t_2, ..., t_n$ that created $x$'s entry in $derived(vs_i)$. This means that each source state vector is valid and reflects actual updates that occurred at the data sources.*
- *Data Source Chronology: $\forall i, derived(vs_i) \leq current[]$ Since view maintenance is asynchronous, the view's state does not correspond to the current state of the data sources, but only valid states that existed at some time.*
- *View State Chronology: if $vs_i < vs_j$ then $derived(vs_i) \leq derived(vs_j)$. Successive view states correspond to successive source state vectors.*
- *Customized On Condition Data Currency: Let $derived(vs_i) = ssv_i$, and $derived(vs_j) = ssv_j$. If $vs_i < vs_j$ then there exists a vector entry $x$ such that $ssv_i[x] < ssv_j[x]$ and $DS_x$ has an active monitoring or triggering ON condition. Changes in the source state vectors reflect updated customized data currency.*

THEOREM 1. *Partial and progressive consistency are stronger than weak consistency.*

PROOF. 1. The properties of weak consistency hold for both partial and progressive consistency (i.e., each view state is based on a valid data source states). 2. Partial and progressive consistency have a property, called View State Chronology, that provides an ordering among the view states. If view state $vs_i < vs_j$ in the view state sequence then by the definition of recomputational consistency 5, $derived(vs_i) \leq derived(vs_j)$. Weak consistency does not have this property. □

THEOREM 2. *Progressive consistency is stronger than partial consistency.*

PROOF. (Sketch) Progressive has a property that partial does not have in that updated customized data currency is expected across all data sources (i.e., across all components of the source state vector). □

THEOREM 3. *Strong consistency is stronger than partial and progressive consistency.*

PROOF. Strong consistency requires that for one view state the components of the source state vector correspond to the data sources after a set $T_k$ of transactions have executed. Let the source state vector $ssv_k = [ds_{1_k}, ds_{2_k}, ..., ds_{u_k}]$ contain the state of all the data sources after a given set of transactions $T_k$ have been executed. Suppose partial or progressive consistency is equivalent to strong. Then there exists a mapping $m$ from view to source states after the execution of $T_k$ transactions such that each view's state $vs_j$ reflects a valid source state vector where $m(vs_j) = ssv_k$ for some $k$. Consider the case with two data sources $DS1$ and $DS2$. When recomputing the view to create state $vs_j$ it is possible to obtain data from $DS1$ after $T_k$ transactions have executed, but to arrive at $DS2$ after $T_{k+1}$ transactions have executed. Therefore, there does not exist a mapping $m(vs_j) = ssv_k$, and strong consistency can not hold. □

LEMMA 1. *Monitoring and triggering can guarantee at best progressive consistency for the SUT and SLT update scenarios.*

PROOF. We assume that a monitor or trigger reads the most current values of the attributes from a base table when the *ON* condition is satisfied.

As long as every entire view recomputation is associated with a monotonically increasing sequence number, and the sequence numbers are processed by the data sources and view holder agent in order, then any monitoring or triggering view currency customization provides progressive consistency.

Strong consistency means the order in time of the view's states matches the order of the corresponding source states after a set $T$ of transactions have executed. Since we can not guarantee that more updates past what is contained in $T$ have not executed at any of the data sources during a query trip for creating a new view state, strong consistency may not hold.

Completeness requires that every source state be reflected in order by the materialized view. Periodically querying the database's catalog or data may skip updates that occur at a particular data source, and therefore, completeness can not be guaranteed. □

Weak consistency occurs when the state of a materialized view reflects valid updates that occurred at each data source, but the view may reflect a different set of committed transactions at each data source [14]. If we have a global transaction G comprised of two , $gt_1$ at site $DS1$ and $gt_2$ at site $DS2$, then weak consistency is the result in the GT scenario if the state of a materialized view is derived from a source state of $DS1$ after $gt_1$ executes, and a source state of $DS2$ before $gt_2$ is executed. The view created would reflect the committed global transaction at $DS1$ but not at $DS2$, and would be weakly consistent with respect to the data sources.

LEMMA 2. *Monitoring and triggering can guarantee at best weak consistency in the GT update scenario.*

PROOF. Case 1: Global transactions do not use an atomic commit protocol.
For global transaction $G = \{gt_1, gt_2\}$ consider two *ON* condition evaluators, one at $DS_1$ and one at $DS_2$. Without

loss of generality, assume that the monitoring or triggering condition evaluator at $DS1$ initiates the recomputation of the view once $gt_1$ commits. Since neither monitoring nor triggering can be aware of the completion of a global transaction, the view version created could reflect the committed global transaction at $DS1$ but not at $DS2$. This version when cached by the mobile client would be weakly consistent with respect to the data sources.

Case 2: Global transactions use an atomic commit protocol. For global transactions $G_x = \{gt_{x1}, gt_{x2}\}$ and $G_y = \{gt_{y1}, gt_{y2}\}$, consider an $ON$ condition evaluator at $DS_1$. Let the monitoring or triggering condition evaluator at $DS1$ recompute the view once $gt_{x1}$ commits. If the global transactions use an atomic commit protocol, such as 2PC, then $gt_{x1}$ commits $\Leftrightarrow$ $gt_{x2}$ commits. Updates obtained from $DS1$ will be transferred to $DS2$ in order to complete the recomputation of the view and obtain the results created from the commitment of $gt_{x2}$. However, upon arrival at $DS2$, $gt_{y2}$ may have committed even though the results of $gt_{y1}$ are not part of the materialization. Therefore, this view version would reflect the committed global transaction $G_y$ at $DS2$ but not at $DS1$, leading to weak consistency with respect to the data sources. □

Thus, with the introduction of the two new levels of consistency the full view consistency hierarchy becomes: *convergence* ⊂ *weak consistency* ⊂ *partial consistency* ⊂ *progressive consistency* ⊂ *strong consistency* ⊂ *completeness*.

## 4.2 Incremental View Maintenance with *ON* Conditions

So far we have focused on recomputation methods between the view holder agent and the data sources as being more suitable. However, it should be pointed out that mobile agents can be used to implement incremental view maintenance such as the Strobe algorithms. Condition evaluation DBMS-agents utilizing triggering constraints can collect all the insert and delete operations that occur at each relevant data source. The Strobe algorithms can then be used to create a new version of a materialized view that will be maintained by the multi-versioning methods of the view holder agent once the $ON$ condition is satisfied.

As in the case of recomputational materialization, the $ON$ condition must be taken into consideration when determining the level of view consistency achieved. Thus, as discussed above, only if an $ON$ condition is not used, triggering will allow a Strobe algorithm to reach the level of completeness.

# 5. SUMMARY

*Personalization* of information gathering for mobile clients is important due to the computing, communication, and storage limitations of mobile devices. In this paper, we presented an extended form of SQL that allows a mobile client to *customize* the materialization of views at a view holder agent. A materialization program is comprised of two parts, condition evaluation and view evaluation, which monitor the data sources and update the view when the appropriate source changes have occurred. The extension or $ON$ condition expresses the criteria for materialization, describing which data changes should invoke the creation of a new version within the view holder agent.

We studied the use of mobile agents in implementing a materialization program and found that recomputational rather than incremental view maintenance is more suitable in a view holder environment. In understanding how view consistency is affected by mobile view currency customization, we introduced two additional levels of view consistency. These new levels allow a view holder agent to provide guarantees to ensure that monitored updates at data sources operating under the SUT or SLT update scenario are chronologically reflected in the versions maintained for the mobile client. We formally defined *customized recomputational consistency* and proved properties of the new view consistency hierarchy.

# 6. 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] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, 2(5): 34–49 ,1995.

[3] R. Elmasri and S. B. Navathe. Fundamentals of Database Systems. Addison Wesley, 2000.

[4] 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, 1996.

[5] D. Kotz, Robert S. Gray, S. Nog, D. Rus, S. Chawla, and G. Cybenko. AGENT TCL: Targeting the Needs of Mobile Computers. *IEEE Internet Computing*, 1(4):58–67 , 1997.

[6] S. Papastavrou, G. Samaras and E. Pitoura. Mobile Agents for World Wide Web Distributed Database Access, *IEEE TKDE*, 12(5):802–820, 2000.

[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] N. Suri, J.M. Bradshaw, M. R. Breedy, P.T. Groth, G.A. Hill, and R.Jeffers. Strong Mobility and Fine-grained Resource Control in NOMADS. In *the 2nd Int'l Symp. on Agent Systems and Applications and 4th Int' Symp. on Mobile Agents*, 1882:2–15, 2000.

[9] S. Weissman Lauzac. The View Holder Approach: Utilizing Customized Materialized Views to Create Database Services Suitable for Mobile Database Applications. PhD Thesis, U. of Pittsburgh, 2001.

[10] S. Weissman Lauzac and P. K. Chrysanthis. Programming Views for Mobile Database Clients. In *the 9th DEXA Int'l Workshop on Mobility in Databases and Distributed Systems*, pp. 408–413, 1998.

[11] S. Weissman Lauzac and P. K. Chrysanthis. Utilizing Versions of Views within a Mobile Environment. *Journal of Computing and Information*, 1999.

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

[13] Y. Zhuge, H. Garcia-Molina, and J. Wiener. Consistency Algorithms for Multi-Source Warehouse View Maintenance. *Distributed and Parallel Databases*, 4(4), 1997.

[14] Y. Zhuge. Incremental Maintenance of Consistent Data Warehouses. PhD thesis, Stanford Univ., 1999.