Concurrency Control Theory for Deferred Materialized Views

Akira Kawaguchi[1] **Daniel Lieuwen**[2] **Inderpal Singh Mumick**[3]
Dallan Quass[4] **Kenneth A. Ross**[1]

[1] Columbia University, akira@cs.columbia.edu, kar@cs.columbia.edu.[‡]
[2] Bell Laboratories, lieuwen@research.bell-labs.com.
[3] AT&T Laboratories, mumick@research.att.com.
[4] Stanford University, quass@cs.stanford.edu.

Abstract. We consider concurrency control problems that arise in the presence of materialized views. Consider a database system supporting materialized views to speed up queries. For a range of important applications (*e.g.* banking, billing, network management), transactions that access materialized views would like to get some consistency guarantees—if a transaction reads a base relation after an update, and then reads a materialized view derived from the base relation, it expects to see the effect of the base update on the materialized view. If a transaction reads two views, it expects that the two views reflect a single consistent database state.

Such guarantees are not easy to obtain, as materialized views become inconsistent upon updates to base relations. *Immediate* maintenance reestablishes consistency within the transaction that updates the base relation, but this consistency comes at the cost of delaying update transactions. *Deferred* maintenance has been proposed to avoid penalizing update transactions by shifting maintenance into a different transaction (for example, into the transaction that reads the view). However, doing so causes a materialized view to become temporarily inconsistent with its definition. Consequently, transactions that read multiple materialized views, or that read a materialized view and also read and/or write base relations may execute in a non-serializable manner even when they are running under a strict two phase locking (2PL) protocol.

We formalize the concurrency control problem in systems supporting materialized views. We develop a serializability theory based upon conflicts and serialization graphs in the presence of materialized views. Concurrency control algorithms based on this theory are being developed in the SWORD/Ode database system.

[‡] The work of Akira Kawaguchi and Kenneth A. Ross was performed while visiting AT&T Bell Laboratories, and was also partially supported by a grant from the AT&T Foundation, by a David and Lucile Packard Foundation Fellowship in Science and Engineering, by a Sloan Foundation Fellowship, and by an NSF Young Investigator Award.

1 Introduction

A *view* is a derived relation defined in terms of base (stored) relations. A view thus defines a function from a set of base tables to a derived table; this function is typically recomputed every time the view is referenced. A view can be materialized by storing the tuples of the view in the database. Index structures can be built on the materialized view. Consequently, database accesses to the materialized view can be much faster than recomputing the view.

Importance of Materialized Views: Materialized views can be used to speed up the processing of complex queries. It is quicker to access a materialized view than to recompute the corresponding query from scratch. In this sense a materialized view is like a cache—a commonly used version of the data that can be accessed quickly. Materialized views are finding applications in domains such as data warehousing, mobile systems, data visualization, banking, billing, and network management [9, 15] where quick response to complex queries is critical.

Importance of Deferred View Maintenance: Like a cache, a materialized view gets out-of-date whenever the underlying base relations are modified. The process of making a materialized view consistent with the base relations from which it is derived is called *view maintenance*. A view can be maintained in an *immediate* or a *deferred* manner. Immediate view maintenance algorithms maintain the view inside the transaction that modifies the base relations, before the transaction commits. Immediate maintenance can significantly slow down update transactions, especially if the number of views is large. Deferred maintenance algorithms maintain the view *outside* the transaction that modifies the base relations. Deferred maintenance thus has minimal impact on update transactions. There exist workloads (such as when there are frequent updates to large tables and relatively infrequent reads, or alternating bursts of updates and reads) under which deferred maintenance performs better than immediate maintenance [6].

Importance of Serializability: Some application domains (*e.g.*, decision support) use materialized views, but do not require serializability guarantees. Often, the materialized views are stored in a separate database from the base data, and the only desirable property may be that the views be mutually consistent.

However, other applications (*e.g.*, banking, billing, and network management) can use materialized views, but require serializability guarantees. Queries in these environments usually run on the same database that supports on-line transaction processing because they need answers that reflect the *current* state of the database. Furthermore, queries that access complex derived data and could benefit from reading materialized views may appear in the same transaction with queries that access base data, with both sets of queries being expected to see a consistent database state. For example, in a telephone billing application, the summary of charges reported to a user or to a switch must match the detailed call data. A credit limit application in cellular or long distance telephone service would require that charges due to all telephone calls be included in the summary being read to determine whether the credit limit has been reached or not. Similarly, routing information stored as a view in a network switch must match base data in the switch.

Problem Addressed: The goal of our work is to provide serializability guarantees to transactions that access both base relations and materialized views when the views are maintained in a deferred manner. The problem is that by allowing transactions that update base relations to commit without updating the materialized views that are derived from those base relations, we allow the database to enter an inconsistent state temporarily. We need to ensure that no transaction is allowed to read any inconsistencies. This means that if a transaction reads both a base relation and a materialized view, it must see the effects of the same set of transactions in both. It is not permissible to allow a transaction to see the effects of a transaction in a base relation, and miss them when reading a materialized view. Similarly, if a transaction reads two materialized views, we must ensure that it sees the effects of the same set of transactions in both the views.

Summary of Contributions and Paper Outline: We show through motivating examples that naive application of (deferred) view maintenance and strict two-phase locking protocols is insufficient to guarantee serializability in a system supporting materialized views (Section 1.1). Section 2 describes the notation and background. We define what is meant for a schedule to be serializable in the presence of materialized views and extend serializability theory to cover materialized views in Section 3. Related work is discussed in Section 4, and we conclude in Section 5.

1.1 Motivating Examples

We present two examples that show that naive application of deferred view maintenance and strict 2PL is insufficient to guarantee serializability when transactions read materialized views. Inconsistencies arise even when the transaction reading the view does not modify base relations. Even more opportunities for inconsistency arise when the transaction reading the view also modifies base relations. We formalize our notion of serializability in the presence of materialized views in Section 3. For now, it is sufficient to think of serializability in the presence of materialized views as the following restriction: A transaction T_1 either always or never sees the effects of changes made by a transaction T_2, regardless of whether T_1 reads a base relation updated by T_2, or it reads a materialized view derived from a base relation updated by T_2.

Example 1. **Maintain at transaction start, or before the first view read:** In this example we show that two-phase locking, in conjunction with a policy of maintaining a view at the start of a transaction that reads the view or just before reading the view for the first time, is insufficient to guarantee serializability. Consider a materialized view V derived from the one-to-one join of two base relations R and S. (We shall use this view V throughout the rest of this paper.) Let V_X denote a tuple in view V derived from the tuple R_X in R and the tuple S_X in S. (Similarly V_Y is derived from R_Y and S_Y.) Transactions T_1 and T_2 are running concurrently. Figure 1a shows their execution history.

Time	T_1	T_2
1	$rl[V_X]$	
(Assume V is up to date; hence no maintenance is needed)		
2	$r[V_X]$	
3		$wl[R_Y]$
4		$w[R_Y]$
5		$u[R_Y]$
6		commit
7	$rl[R_Y]$	
8	$r[R_Y]$	
9	$rl[V_Y]$	
10	$r[V_Y]$	
(R_Y is inconsistent with V_Y!)		
11	$u[V_X], u[R_Y], u[V_Y]$	
12	commit	

(a) Maintain at transaction start

Time	T_1	T_2
1	$rl[V_X]$	
(Assume V is up to date; hence no maintenance is needed)		
2	$r[V_X]$	
3		$wl[R_X]$
4		$w[R_X]$
5		$u[R_X]$
6		commit
(Detect that V_X is out of date)		
7	$rl[R_X]$	
8	$wl[V_X]$	
(Upgrade lock on V_X to write-lock)		
9	$m[V_X]$	
(Maintain V_X from R_X ($m[V_X]$ is a macro for $r[R_X]$, $r[S_X]$, $w[V_X]$))		
10	$r[V_X]$	
(V_X has a different value than before!)		
11	$u[V_X], u[R_X]$	
12	commit	

(b) Maintain before each read

Fig. 1. Concurrency control anomalies for deferred materialized views

In the figure,

- $rl[R_X]$ means "Get a read-lock on tuple X in relation R,"
- $r[R_X]$ means "Read tuple X in relation R,"
- $wl[R_X]$ means "Get a write-lock on tuple X in relation R,"
- $w[R_X]$ means "Write tuple X in relation R,"
- $u[R_X]$ means "Unlock X in relation R."

When T_1 starts it detects that V is already completely up to date, so maintenance is not needed. T_1 reads V_X. It then tries to get a read-lock on R_Y, but in the meantime T_2 has obtained a write-lock on R_Y, so T_1 waits for T_2 to release that lock. After T_2 commits T_1 reads R_Y and V_Y. T_1 sees T_2's changes to R_Y, but it doesn't see T_2's changes to the corresponding tuple V_Y. Therefore, transaction T_1 sees an inconsistent database state, violating serializability.

Note that the schedule violates serializability even if the operations $rl[V_X]$ and $r[V_X]$ at time 1 and 2 are removed. However, by including these two operations we show that a policy to maintain a view just before the first time the view is read does not ensure serializability.

The previous example shows that simply maintaining the view once at the beginning of a transaction or just before first reading the view is insufficient to guarantee serializability, even when the schedule satisfies a strict two-phase locking protocol. The following example shows that maintaining the view before each read operation is also insufficient to guarantee serializability.

Example 2. **Maintain just before each read of the view:**
Figure 1b shows an execution history where the view is maintained before each read. When T_1 starts it detects that V is already completely up to date, so maintenance is not needed. T_1 reads V_X as before. Meanwhile, T_2 updates R_X and commits. When T_1 tries to re-read V_X, it notices that the view is now out of date, so it maintains V_X. It does this by obtaining a read lock on R_X, a write lock on V_X, and applying a *maintain* operation, denoted m. The maintain operation is a macro for reading all base relation tuples that contribute to the view tuple, then writing the view tuple. Tuple V_X now has a different value than the first time it was read, violating serializability. (Another alternative might be to not maintain a tuple that is out of date if it had been read previously in the transaction, but then the read of V_X would be inconsistent with a read of R_X.)

As illustrated by the previous examples, naive application of deferred view maintenance and strict 2PL is insufficient to guarantee serializability when transactions read materialized views. This paper extends serializability theory to handle transactions that read materialized views, especially when the views are maintained in a deferred manner. A view can be maintained within the reader's transaction or as a separate transaction. Maintaining the view in a separate transaction has performance advantages, but can lead to additional consistency problems.

2 Notation and Background

A database consists of a set of base relations and views. A read-write locking model is used for both base and view data. A view is defined by an SQL `CREATE VIEW` statement. All base relations and view names that appear in the `FROM` clause of the SQL statement (including subqueries) defining a view V are said to *derive* view V.

Definition 1. (Dependency Graph) The dependency graph $G(V)$ of a view is a directed graph, with a node for every base relation and view used in defining view V, either directly or through other views. There is an edge from a node N_1 to a node N_2 if N_1 derives view N_2.

Algorithms that compute changes to a view in response to changes to the base relations are called *incremental view maintenance* algorithms. A classification and survey of several view maintenance algorithms appears in [9]. View maintenance algorithms assume that the changes to the base relations are stored in one or more *delta* relations. In this paper, we will make the assumption that for every base relation R used in a view V, we have available a delta relation $\Delta R(V)$, containing all modifications (insertions, deletions, and updates) made to relation R since the last time view V was maintained. The particular algorithms used to obtain $\Delta R(V)$ and perform incremental maintenance are orthogonal to our discussion of concurrency control. We will abstract the maintenance algorithm for view V as the following maintain function:

1. Check if view V needs to be refreshed.
2. If yes, then:
   (a) Read all delta relations for view V.
   (b) Read all or part of the base relations in the dependency graph of V.
   (c) Write the changed tuples to the materialization of view V.

The second step of maintain updates the materialized view to bring it up-to-date, and will be called the *refresh*. Depending upon the SQL statement defining view V, it is possible that Step 2b is not required (as for single-relation selection and other self-maintainable views [8]). However, in general, Step 2b is needed.

A system has a choice about when to apply the incremental view maintenance algorithm—*immediate maintenance* within the transaction that modifies a base relation, or *deferred maintenance* outside the modifying transaction.

For immediate view maintenance, serializability between transactions that modify base relations and read materialized views can be guaranteed by using two-phase locking, with the following minor extension. If a transaction updates a base relation and then reads a materialized view derived from the base relation, the changes made by the transaction must be propagated into the view before the view is read so that the transaction sees the effects of its own writes. If a transaction that reads a view V does not modify base relations that derive V, then propagating changes to views at transaction commit is sufficient to obtain serializability.

Deferred maintenance may be done just before reading the view (lazy deferred), at regular intervals by a daemon process (periodic deferred), on demand by a user (also called periodic deferred), or it may be triggered by some other condition. An example of such a triggering condition is when the materialized view exceeds a tolerance range for inconsistency [18]. Deferred view maintenance breaks the serializability guarantee of two phase locking. Additional steps must be taken to guarantee serializability in the presence of deferred view maintenance.

There has been substantial work on concurrency control for databases with multiple versions of data or multiple copies (replicas) of data. This is relevant for materialized views because one can think of a materialized view as another "version" of the underlying data. The relationship between the present work and the theories of multi-version concurrency control and replica concurrency control are discussed in Section 4.

3 Serializability Theory for Materialized Views

Traditionally, views in databases have been *virtual*. Queries over virtual views are answered by accessing base data. We would like view maintenance to be *transparent* to the user of a database system. In other words, view maintenance may improve the *performance* of queries, but it may not change the *semantics* of queries over views. We require that a materialized view appear to the user as if it were a virtual view. Our extended notion of serializability, called *Mat-serializability*, incorporates this notion of transparency.

Definition 2. (Serializability for Materialized Views) A transaction history is *Mat-Serializable* iff it is equivalent to some serial execution of the same transactions with all materialized-view reads treated as virtual-view reads.

The serializability theory of materialized views can be seen as an extension of multi-version serializability theory to account for the following:

- Views are not copies of base tuples, but are derived using query functions.
- User histories contain operations over both views and base relations, as well as a maintain operation.

The reason that traditional concurrency control algorithms don't work in the presence of materialized views is that there is an implied relationship between tuples in a materialized view and tuples in the base relations on which the view is defined. This relationship is unknown to, and therefore not enforced by, traditional concurrency control algorithms, which treat base relations and views as independent lockable entities. The reason two-version concurrency control does not work is that the relationship between versions is complex, the user has access to both versions, and certification is not feasible before commit time.

In this section, we extend traditional serializability theory to include materialized views. We extend the traditional read-write conflict matrix to include new operations that apply to materialized views, and show that an execution history is Mat-Serializable iff a serialization graph with edges due to the new conflicts is acyclic.

3.1 Derivation Sets

We formally define the set of tuples in base relations that contribute to the derivation of a tuple in a view. For simplicity of presentation, we define this concept here for a single-block SQL query defining a view. Note that a predicate is said to be *local* if all the attributes referenced in the predicate are from the same relation.

Definition 3. (Localized query) Consider an arbitrary view V defined by a single-block SQL query. We construct the localized query V' from V by performing the following steps to V. First, we omit the `GROUPBY` and `HAVING` clauses, and omit any aggregated attributes from the `SELECT` clause. We then replace the `WHERE` clause by the strongest condition C that can be derived from the original `WHERE` clause, such that C consists only of local predicates and predicates relating attributes in the `SELECT` clause.

A localized query allows us to isolate those tuples in the underlying relations that, for some database extension, would affect the result of the query. This concept is formalized in the next definition.

Definition 4. (Derivation set) Let V be a view, let V' be its localized query, let t be a tuple with the arity of V, and let every attribute value of t be from the appropriate domain for V. (t may or may not actually be in the extension of the view.) Let t' be the restriction of t to the schema of V'. Instantiate the conditions of V' with the corresponding values from t', so that every remaining condition is a local selection condition on an input relation R in the `FROM` clause of V'. The *derivation set* of t in V, written as $ds(t)$ when V is understood from context, is defined as the set of all tuples in the schema of the `FROM` clause relations R that satisfy the conditions in the `WHERE` clause of V', and that have attribute values equal to those attributes of t specified in the `SELECT` clause (for those relations with selected attributes).

The derivation set represents the set of all tuples whose insertion, deletion or modification could potentially, for some database extension, affect the tuple t in the view. (For an update to a tuple, we need to check both the old and new value for membership in the derivation set.) It is important that we define the derivation set in a *database independent* way. We do not want to have to consult the database (which may be in the process of being updated) in order to determine the presence or absence of a conflict.

Example 3. The view V on the left has a localized query given on the right.

```
SELECT  Emp.EmpId, min(Deadline)
FROM    Emp, Works, Proj
WHERE   Emp.EmpId = Works.EmpId
        and Proj.ProjId = Works.ProjId
        and Proj.manager = 'Fred'
GROUPBY Emp.EmpId
HAVING  min(Deadline) > 10/23/95
```

```
SELECT Emp.EmpId
FROM   Emp, Works, Proj
WHERE  Emp.EmpId = Works.EmpId
       and Proj.manager = 'Fred'
```

For the tuple $t = (123, 10/30/95)$, $ds(t)$ consists of (a) All Emp tuples with EmpId 123, (b) all Works tuples with EmpId 123, and (c) all Proj tuples with manager "Fred". We expect that any change to tuples in $ds(t)$ could affect t's presence in the view; a change of a tuple outside $ds(t)$ to a value in $ds(t)$ would also affect t.

3.2 Materialized View Histories and Virtual View Histories

Traditional serializability theory defines two basic operations on a data item: read and write. We need to distinguish the read of a view tuple from the read of a base relation tuple; hence we will denote a read operation on a view tuple as r^V, and a read operation on a base relation tuple as r. Since we assume view tuples are only maintained, not modified directly by transactions, the write operation in a user transaction, denoted w, applies only to tuples in base relations.

We also add a new maintenance operation, denoted m, that maintains a given view tuple t, similar to the macro $m[V_X]$ introduced in Figure 1b. The maintain operation $m[t]$ performs the following sequence of operations:

Check if tuple t in view V needs to be refreshed; *i.e.*, there has been a write to a tuple in $ds(t)$ since t was last maintained. If yes, then:

1. Read the tuples of the delta relations for V that are in $ds(t)$.
2. Read the tuples of the base relations in the dependency graph of V that are in $ds(t)$.
3. Write t.

Note that the check for a write is built in as a part of the m operation. *Actual* refresh is performed only when necessary; *i.e.*, when a modification to some tuple in the view tuple's derivation set has occurred since the last maintenance operation on the view tuple. We assume that, from the point of view of concurrency control, maintenance operations are done atomically. We assume that a view tuple is always maintained by reading the delta relations and base relation tuples in its derivation set. Even when a view is defined in terms of other views, we treat those views as virtual and only read base relation tuples to maintain the view.

Recall that an execution history is Mat-Serializable if it is equivalent to a serial ordering of the transactions with all materialized-view reads treated as virtual-view reads. The definition does not require that we treat a materialized-view read as a virtual-view read that occurs at the same point in time as the materialized-view read. Because views are materialized, they may correspond to database states that are slightly out of date. In fact, a materialized-view read is equivalent to a virtual-view read performed at the point in time when the view tuple was last maintained. So, it is possible to translate a materialized-view read to an earlier virtual-view read, so long as the validity condition specified in the following Definition 5 is satisfied.

Transactions in a *materialized view history* have operations r, w, m, and r^V, where r^V is interpreted as reading from the stored version of the view. Transactions in a *virtual view history* have operations r, w and r^V, where r^V is interpreted as reading from the virtual view, *i.e.*, from the corresponding base relations. (Note that we do not translate the r^V operation into a sequence of base relation read operations in the history.)

Definition 5. (Valid History) Let t be a tuple in a view. A materialized view history is *valid* if for every transaction T in the history that writes a tuple in $ds(t)$ and later reads t, there is a maintain operation on t (either in T or elsewhere) occurring between the write to $ds(t)$ and the read of t.

In order for a history to be valid, transactions must see the effects of their own writes on the view tuples they read. A valid history guarantees that when translating a materialized-view read to a virtual-view read, the virtual-view read does not occur before a previous write by the same transaction to a tuple in the derivation set of the view tuple.

Lemma 6. *A valid materialized-view history can be translated to an equivalent virtual-view history by translating all materialized-view tuple reads to virtual-view tuple reads performed at the point in time when the view tuple was last maintained, and omitting the maintain operations.*

	$r[u]$	$r^V[t]$	$w[u]$
$r[u]$	$-$	$-$	$\times$
$r^V[t]$	$-$	$-$	$\times$
$w[u]$	$\times$	$\times$	$\times$

Table 1. Conflict Matrix

Lemma 6 guarantees that any valid history involving materialized views can be translated to an equivalent history involving virtual views. Let $virtual(H)$ be the virtual-view history that is equivalent to the valid materialized-view history H according to Lemma 6. By our definition of Mat-serializability, in order for a valid materialized-view history H to be Mat-Serializable, $virtual(H)$ must be serializable. An invalid materialized-view history cannot be translated to an equivalent virtual-view history, and hence cannot be Mat-Serializable, since transactions would not see the effects of their own earlier writes.

Table 1 extends the read-write conflict matrix to virtual-view histories, including reads of virtual views. In the table, t denotes a view tuple and u denotes a base relation tuple. In the conflict between $r^V[t]$ and $w[u]$, we assume u is a tuple in $ds(t)$. Note that there is no maintain operation in the conflict matrix, since maintenance operations do not appear in virtual view histories. Conflicts between r and w operations derive from the traditional read-write conflicts: If a read or write operation precedes and conflicts with another read or write operation in the history, it must precede the other in the equivalent serial history. Conflicts between r^V and w operations are similar. Consider any pair of $r^V[t]/w[u]$ operations in a virtual-view history. If the write precedes the view read, then the write must precede the view read in the equivalent serial history since the effect of the write is visible to the view reader. Conversely, suppose that the view read precedes the write. Then the view read must precede the write in the equivalent serial history because the view reader does not see the effect of the write.

3.3 Mat-Serialization Graphs

In this section, we show how to construct a serialization graph for histories that include maintaining and reading materialized views. We call the extended graph a *Mat-Serialization graph.* We show that a valid history is Mat-Serializable iff the Mat-Serialization graph is acyclic. We then describe two types of anomalies that can occur if cycles are allowed in the Mat-Serialization graph.

Recall that a traditional serialization graph is a directed graph whose nodes are transactions, and where there is an edge from transaction T_i to T_j if an operation by T_i precedes and conflicts with an operation by T_j. We define a Mat-Serialization graph for a materialized-view history H similarly, to include edges between transactions in $virtual(H)$ due to the conflicts presented in Table 1. The direction of an edge due to conflicts between (base relation) reads and writes is from the transaction containing the preceding operation to the transaction containing the succeeding operation as usual. The Mat-Serialization graph represents all of the dependencies on the equivalent virtual-view history $virtual(H)$ for a materialized-view history H.

Definition 7. (Mat-Serialization Graph) Let H be a valid materialized-view history. A Mat-Serialization graph of H is a directed graph with a node for each transaction in H, and an edge from transaction T_i to T_j if an operation of T_i precedes an operation of T_j in $virtual(H)$ where the two operations conflict according to Table 1.

Theorem 8. *A valid execution history is* Mat-Serializable *iff the Mat-Serialization graph for the history is acyclic.*

Theorem 8 states that a materialized-view history H is Mat-Serializable iff the serialization graph corresponding to its equivalent virtual-view schedule $virtual(H)$ is acyclic, and each transaction sees the effects of its own writes on the view tuples it reads. A proof of the theorem appears in [13]. It is interesting to examine why we do not need to consider conflicts involving maintenance operations. Intuitively, the reason is that the value written by a maintenance operation $m[t]$ is based solely upon previous writes to tuples in $ds(t)$, and not upon operations in the transaction doing the maintenance (unless the transaction doing the maintenance previously wrote tuples in $ds(t)$). It is acceptable to relax isolation and allow transactions to read the effects of maintenance operations in transactions that may be serialized later. This relaxation is possible so long as all writes to tuples in the derivation set of the maintained tuple that occur before the maintenance operation take place in transactions that are serialized before the view reader. The resulting histories are Mat-Serializable, even though if we thought of the views as regular base relations, with a conflict between a maintain operation and a subsequent view-read operation, they would not be serializable.

Anomalies: Let us now consider the effects of cycles in the Mat-Serialization graph. We demonstrate two anomalies caused by different types of cycles in the Mat-Serialization graph.

Missing Update Transaction T_1 reads a tuple t in a view. Transaction T_2 inserts, deletes, or modifies a tuple in the derivation set of t and commits. If transaction T_1 also reads the tuple written by T_2 (or attempts to read it in case of a delete), T_2's update will appear to be missing from the view.

Example 1 is a case of the missing update anomaly. The Mat-Serialization graph for Example 1 contains a cycle between transactions T_1 and T_2. There is an edge from T_2 to T_1 because R_Y is written at time 4 by transaction T_2 and later read at time 8 by transaction T_1. In addition, there is an edge from T_1 to T_2 because V_Y, which was assumed to be maintained before time 1, is read at time 10 by transaction T_1, and R_Y, which is in the derivation set of V_Y, is written at time

4 by transaction T_2. Even though V_Y is read by T_1 after the write of R_Y by T_2, the edge is from T_1 to T_2 because V_Y was last maintained before time 1, which precedes the write of R_Y by T_2.

Unrepeatable View Read Transaction T_1 reads a tuple t in a view. Transaction T_2 inserts, deletes, or modifies a tuple in the derivation set of t and commits. If t is maintained and T_1 later re-reads t, it will read a modified value or find that t has been deleted.

Example 2 is a case of the unrepeatable view read anomaly. Like Example 1, Example 2's Mat-Serialization graph contains a cycle between transactions T_1 and T_2. There is an edge from T_1 to T_2 because V_X, which was assumed to be maintained before time 1, is read at time 2 by transaction T_1, and R_X, which is in the derivation set of V_X, is written at time 4 by transaction T_2. There is an edge from T_2 to T_1 because R_X is written at time 4 by T_2, and V_X, which has R_X in its derivation set and is maintained at time 9, is read at time 10 by T_1.

3.4 Conflicts at View-Level Granularity

Using derivation sets to determine serializability is of interest if we have an efficient means to lock the derivation sets. When the derivation set consists of a single tuple and can easily be determined from the view tuple (as when the view is a replica of a base relation), it will be possible to lock the derivation set. In general, locking a derivation set requires predicate locks, and requires view maintenance algorithms that work at the tuple level rather than table level. In some cases, (*e.g.* when the view is defined by a join over a key) predicate locks can be realized using indices. However, general predicate locks are not available in most systems. Furthermore, current view maintenance algorithms maintain the entire view, rather than one tuple in the view. Therefore, it is important to consider concurrency control when maintenance is done at the *view-level granularity*. In the following we show that it is possible for algorithms based on view-level granularity to guarantee Mat-serializability.

At view-level granularity, the maintenance operation maintains an entire view, not just an individual view tuple, and obtains table-level locks on all base relations in the view's dependency graph. We continue to assume that operations reading view tuples, and reading and writing base relation tuples obtain tuple-level locks. Conflicts at view-level granularity are easily derived from the tuple-level conflicts of Table 1. We extend the conflict between a read of a view tuple and a write to a tuple in the view tuple's derivation set to include writes to any base relation in the dependency graph of the view. Conflicts between operations on base relation tuples continue to be at the tuple level.

When constructing the Mat-Serialization graph for a materialized-view history H, we construct edges between a view-read operation and a base relation write as before using $virtual(H)$. A history is valid if a write to a view in each transaction T is followed by a maintain operation on the view (either in T or elsewhere) before the view is next read by T.

Corollary 9. *An execution history is Mat-Serializable if the Mat-Serialization graph with conflicts at view-level granularity is acyclic, and the history is valid.*

Corollary 9 states that an execution history is Mat-Serializable if its Mat-Serialization graph for view-level conflicts is acyclic and if each transaction sees the effects of its own writes.

4 Related Work

View Maintenance: Most work on view maintenance [2, 4, 7, 10, 11, 12, 16] has dealt with automatic derivation of the maintain function from the view definition. [9] is a survey and classification of several view maintenance algorithms. [5] deals with deriving the maintain functions specifically for deferred maintenance. Our current work is different from all of the above in that we do not derive (or advocate) any particular maintenance algorithm.

Deferred view maintenance has been implemented in ADMS [17], and has also been proposed by [18, 19, 20]. [18, 19] consider only select-project views. They do not discuss concurrency, and avoid the problems presented here since (1) they do not store the base relations and views at the same node, (2) they assume that a transaction reads only one view, and (3) select-project views can be maintained without accessing base relations. [20] considers a data warehouse where a view is materialized over relations stored in remote autonomous databases. They consider a concurrency problem that arises in computing the maintain function itself, and propose a view maintenance algorithm that can work in *absence* of concurrency control between the warehouse and remote databases. [17] is most closely related to our work, and proposes a concurrency control algorithm to permit concurrent maintenance of several related views. However, they do not allow view reader transactions to modify base relations, and it is not clear whether they allow such transactions to even read base relations directly. Only select-join views are considered; it is not clear how the technique will generalize to other views. There is no discussion of serialization theory.

Multi-Version Concurrency Control: We can draw the following analogy between multi-version serializability theory [1] and the serializability theory of materialized views described in Section 3. The process of translating materialized-view histories into virtual-view histories captures the conflicts represented by version order edges in the multi-version serializability graph, and the validity condition is analogous to the requirement that a transaction read the same version that it writes. The main differences between multi-version concurrency control and concurrency control for materialized views are: (1) a materialized view is just a single extra version of (some of) the data, (2) both base data and materialized views are visible to the user, while versions are invisible to the user, and (3) multiple versions are simply copies of single data items while views are query functions of multiple data items.

Replicas: A materialized view can be seen as a complex form of replicated data. However, replica concurrency control [1] differs from our problem and solution in that: (1) Replicas are assumed to be interchangeable for readers, while view and base relations are not interchangeable, and (2) Replica management either causes updates to be propagated within the updating transaction (similar to immediate maintenance), or causes the reader to read a majority of copies. Our goal in using materialized views is to improve performance by reading derived data and (where possible) to avoid reading the base data; reading multiple copies of the data is not well-suited to enhancing performance.

5 Conclusion

In this paper we have identified an important new problem – concurrency control in database systems that support materialized views. We motivated the problem through examples and developed a serialization theory in the presence of materialized views.

Applications like billing, banking, retailing, and data warehousing all have needs for materialized views. Often performance and scalability make it desirable and/or necessary to maintain views in a deferred manner. Currently these applications materialize the views into base relations and write application code to maintain the views. However, if subsequent queries and transactions access both materialized views and base relations, or if they access multiple materialized views, inconsistent results can be obtained. There is currently no system support to ensure the consistency of materialized views in these applications.

This paper identifies the concurrency control problems that must be solved to support materialized views. This is the first paper on the topic of concurrency control for materialized views, and surely leaves several open questions. Are there notions of non-serializability that will be of interest in a system with materialized views? While there are workload scenarios (frequent updates to large tables and relatively infrequent reads, or alternating bursts of updates and reads) where doing deferred maintenance with table level read locks is helpful, there are also workload scenarios where using predicate locks will be of great value. What is the class of views for which predicate locks can be implemented efficiently?

Mat-Serializability doesn't give us any guarantees that the views read by transactions will be up to date. For example, a transaction that reads a single view might be serialized in the "distant past". In [13] we define an additional property, *currency*, that requires that a transaction T see at least the effects of all transactions that committed up to a certain time.

We have developed concurrency control algorithms based on the theory developed in this paper; these algorithms are currently being implemented in the SWORD/Ode database system [14].

References

1. P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.
2. J. A. Blakeley, P. Larson, and F. W. Tompa. Efficiently Updating Materialized Views. In C. Zaniolo, editor, *Proceedings of ACM SIGMOD 1986 International Conference*, pages 61–71, Washington, D.C., May 1986.

3. M. Carey and D. Schneider, editors. *Proceedings of ACM SIGMOD 1995 International Conference*, San Jose, CA, May 1995.
4. S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In G. M. Lohman, A. Sernadas, and R. Camps, editors, *Proceedings of the Seventeenth VLDB Conference*, pages 108–119, Barcelona, Spain, September 1991.
5. L. Colby, T. Griffin, L. Libkin, I. S. Mumick, and H. Trickey. Algorithms for deferred view maintenance. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of ACM SIGMOD 1996 International Conference*, Montreal, Canada, June 1996.
6. L. Colby, A. Kawaguchi, D. Lieuwen, I. S. Mumick, and K. A. Ross. Implementing materialized views. Unpublished Manuscript, 1996.
7. T. Griffin and L. Libkin. Incremental maintenance of views with duplicates. In Carey and Schneider [3], pages 328–339.
8. A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proceedings of the Fifth EDBT Conference*, Avignon, France, March 1996.
9. A. Gupta and I. S. Mumick. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Engineering Bulletin, Special Issue on Materialized Views and Data Warehousing*, 18(2):3–19, June 1995.
10. A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference*, Washington, DC, May 1993.
11. E. N. Hanson. A performance analysis of view materialization strategies. In U. Dayal and I. Traiger, editors, *Proceedings of ACM SIGMOD 1987 International Conference*, pages 440–453, San Francisco, CA, May 1987.
12. H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues in the chronicle data model. In *Proceedings of the Fourteenth PODS Symposium*, pages 113–124, San Jose, CA, May 1995.
13. A. Kawaguchi, D. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross. Concurrency control theory for deferred materialized views. Technical Memorandum, AT&T and Bell Laboratories, 1996.
14. A. Kawaguchi, D. Lieuwen, I. S. Mumick, D. Quass, and K. A. Ross. Implementing concurrency control for deferred view maintenance. Unpublished manuscript, 1996.
15. I. S. Mumick. The Rejuvenation of Materialized Views. In *Proceedings of the Sixth International Conference on Information Systems and Management of Data (CISMOD)*, Bombay, India, November 1995.
16. X. Qian and G. Wiederhold. Incremental recomputation of active relational expressions. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):337–341, 1991.
17. N. Roussopoulos and H. Kang. Principles and techniques in the design of ADMS$\underline{+}$. *IEEE Computer*, pages 19–25, December 1986.
18. A. Segev and W. Fang. Currency-based updates to distributed materialized views. In *Proceedings of the Sixth IEEE International Conference on Data Engineering*, pages 512–520, Los Angeles, CA, February 1990.
19. A. Segev and J. Park. Updating distributed materialized views. *IEEE Transactions on Knowledge and Data Engineering*, 1(2):173–184, June 1989.
20. Y. Zhuge, H. Garcia-Molina, J. Hammer, and J. Widom. View maintenance in a warehousing environment. In Carey and Schneider [3], pages 316–327.

This article was processed using the LaTeX macro package with LLNCS style