Locking Protocols for Materialized Aggregate Join Views

Gang Luo Jeffrey F. Naughton
University of Wisconsin-Madison
{gangluo, naughton}@cs.wisc.edu

Curt J. Ellmann Michael W. Watzke
NCR Advance Development Lab
{curt.ellmann, michael.watzke}@ncr.com

Abstract

The maintenance of materialized aggregate join views is a well-studied problem. However, to date the published literature has largely ignored the issue of concurrency control. Clearly immediate materialized view maintenance with transactional consistency, if enforced by generic concurrency control mechanisms, can result in low levels of concurrency and high rates of deadlock. While this problem is superficially amenable to well-known techniques such as fine-granularity locking and special lock modes for updates that are associative and commutative, we show that these previous techniques do not fully solve the problem. We extend previous high concurrency locking techniques to apply to materialized view maintenance, and show how this extension can be implemented even in the presence of indices on the materialized view.

1. Introduction

Although materialized view maintenance has been well-studied in the research literature [GM99], with rare exceptions, to date that published literature has ignored concurrency control. In fact, if we use generic concurrency control mechanisms, immediate materialized aggregate join view maintenance becomes extremely problematic — the addition of a materialized aggregate join view can introduce many lock conflicts and/or deadlocks that did not arise in the absence of this materialized view.

As an example of this effect, consider a scenario in which there are two base relations: the *lineitem* relation, and the *partsupp* relation, with the schemas *lineitem* (*orderkey*, *partkey*) (and possibly some other attributes), and *partsupp* (*partkey*, *suppkey*). Suppose that in transaction T_1 some customer buys items p_{11} and p_{12} in order o_1, which will cause the tuples (o_1, p_{11}) and (o_1, p_{12}) to be inserted into the *lineitem* relation. Also suppose that concurrently in transaction T_2 another customer buys items p_{21} and p_{22} in order o_2. This will cause the tuples (o_2, p_{21}) and (o_2, p_{22}) to be inserted into the *lineitem* relation. Suppose that parts p_{11} and p_{21} come from supplier s_1, while parts p_{12} and p_{22} come from supplier s_2. Then there are no lock conflicts nor is there any potential for deadlock between T_1 and T_2, since the tuples inserted by them are distinct.

Suppose now that we create a materialized aggregate join view *suppcount* to provide quick access to the number of parts ordered from each supplier, defined as follows:

```
create aggregate join view suppcount as
select p.suppkey, count(*)
from lineitem l, partsupp p
where l.partkey=p.partkey
group by p.suppkey;
```

Now both transactions T_1 and T_2 must update the materialized view *suppcount*. Since both T_1 and T_2 update the same pair of tuples in *suppcount* (the tuples for suppliers s_1 and s_2), there are now potential lock conflicts. To make things worse, suppose that T_1 and T_2 request their exclusive locks on *suppcount* in the following order:
(1) T_1 requests a lock for the tuple whose *suppkey*=s_1.
(2) T_2 requests a lock for the tuple whose *suppkey*=s_2.
(3) T_1 requests a lock for the tuple whose *suppkey*=s_2.
(4) T_2 requests a lock for the tuple whose *suppkey*=s_1.
Then a deadlock will occur.

The danger of this sort of deadlock is not necessarily remote. Suppose there are R suppliers, m concurrent transactions, and that each transaction represents a customer buying items randomly from r different suppliers. Then according to [GR93, page 428-429], if $mr<<R$, the probability that any particular transaction deadlocks is approximately $(m-1)(r-1)^4/(4R^2)$. (If we do not have $mr<<R$, then the probability of deadlock is essentially one. Thus, no matter whether $mr<<R$ or not, we can use a unified formula $min(1, (m-1)(r-1)^4/(4R^2))$ to roughly estimate the probability that any particular transaction deadlocks.) For reasonable values of R, m, and r, this probability of deadlock is unacceptably high. For example, if $R=3{,}000$, $m=8$, and $r=32$, the deadlock probability is approximately 18%. Merely doubling m to 16 raises this probability to 38%. In such a scenario large numbers of concurrent transactions will result in very high deadlock rates.

In view of this, one alternative is to simply avoid updating the materialized view within the transactions. Instead, we batch these updates to the materialized view

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

and apply them later in separate transactions. This "works"; unfortunately, it requires that the system gives up on serializability and/or recency (it is possible to provide a theory of serializability in the presence of deferred updates if readers of the materialized view are allowed to read old versions of the view [KLM+97].) Giving up on serializability and/or recency for materialized views may ultimately turn out to be the best approach for any number of reasons; but before giving up altogether, it is worth investigating techniques that guarantee immediate update propagation with serializability semantics yet still give reasonable performance. Providing such guarantees is desirable in certain cases. (Such guarantees are required in the TPC-R benchmark [PF00], presumably as a reflection of some real world application demands.) In this paper we explore techniques that can guarantee serializability without incurring high rates of deadlock and lock contention.

Our focus is materialized aggregate join views. In an extended relational algebra, a general instance of such a view can be expressed as $AJV=\gamma(\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)))$, where γ is the aggregate operator. SQL allows the aggregate operators *COUNT*, *SUM*, *AVG*, *MIN*, and *MAX*. However, because *MIN* and *MAX* cannot be maintained incrementally (the problem is deletes [GKS01]), we restrict our attention to the three aggregate operators that make sense for materialized aggregates: *COUNT*, *SUM*, and *AVG*. Note that by letting $n=1$ in the definition of *AJV*, we also include aggregate views over single relations.

A useful observation is that for *COUNT*, *SUM*, and *AVG*, the updates to the materialized aggregate join views are associative and commutative, so it really does not matter in which order they are processed. In our running example, the state of *suppcount* after applying the updates of T_1 and T_2 is independent of the order in which they are applied. (Some care must be exercised to ensure that transactions that, unlike T_1 and T_2, are reading *suppcount* also see a consistent view of *suppcount*.) This line of reasoning leads one to consider locking mechanisms that increase concurrency for commutative and associative operations.

Many special locking modes that support increased concurrency through the special treatment of "hot spot" aggregates in base relations [GK85, O86, Reu82] or by exploiting update semantics [BR92, RAA94] have been proposed. An early and particularly relevant example of locks that exploit update semantics was proposed by Korth [Kor83]. The basic idea is to identify classes of update transactions so that within each class, the updates are associative and commutative. For example, if a set of transactions update a record by adding various amounts to the same field in the record, they can be run in any order and the final state of the record will be the same, so they can be run concurrently. To ensure serializability, other transactions that read or write the record must conflict with these addition transactions. This insight is captured in Korth's P locking protocol, in which addition transactions get P locks on the records they update through addition, while all other data accesses (including those by transactions not doing additive updates) are protected by standard S and X locks. P locks do not conflict with each other while they do conflict with S and X locks.

Borrowing this insight, we propose a V locking protocol ("V" for "View.") In it, transactions that cause updates to materialized aggregate join views with associative and commutative aggregates (including *COUNT*, *SUM*, and *AVG*) get standard S and X locks on base relations but get V locks on the materialized view. V locks conflict with S and X locks but not with each other. At this level of discussion, V locks appear virtually identical to the (20+ year old!) P locks.

Unfortunately, there is a subtle difference between the problem solved by P locks and the materialized aggregate join view update problem. For P locks, the assumption is that updates are of two types: updates that modify existing tuples, which are handled by P locks; and updates that create new tuples or delete existing tuples, which are handled by X locks. At this level the same solution applies to updates of materialized aggregate join views. However, a transaction cannot know at the outset whether it will cause an update of an existing materialized view tuple, the insertion of a new tuple, or the deletion of an existing tuple. (Recall that the transaction inserts a tuple into a base relation and generates a new join result tuple, which only indirectly updates a materialized view tuple — the transaction does not know from the outset whether or not this new join result tuple will be aggregated into an existing materialized view tuple.) If we use X locks for the materialized view updates, we are back to our original problem of high lock conflict and deadlock rates. If we naively use our V locks for these updates, as we will show in Section 2, the semantics of the aggregate join view may be violated. In particular, it is possible that we could end up with what we call "split group duplicates" — multiple tuples in the aggregate join view for the same group. (Due to a similar reason, previous approaches for handling "hot spot" aggregates [GK85, O86, Reu82, BR92, RAA94] cannot be applied to materialized aggregate join views.)

To solve the split group duplicate problem, we augment V locks with a construct we call W locks. W locks are short-term locks. (The W lock sounds a lot like a latch, but it is not a latch; the split group duplicate problem arises even in the presence of latches. Furthermore, unlike latches, W locks must be considered in deadlock detection.) With W locks the semantics of materialized aggregate join views can be guaranteed — at any time, for any aggregate group, either zero or one tuple corresponding to this group exists in a materialized aggregate join view. Also, the probability of lock conflicts and deadlocks is greatly reduced, because W locks are short-term locks, and V locks do not conflict with each other or with W locks.

It is straightforward to implement V locks and W locks if the materialized view is stored without any indices or with hash indices. However, things become much more complex in the common case that there are B-tree indices over the materialized view. In this case, since the V lock is a form of a predicate lock, our first thought was to borrow from techniques that have been proposed for predicate locks. In particular, key-range locking (a limited form of predicate locking) on B-tree indices has been well-studied [Moh90a, Lom93]. However, we cannot simply use the techniques in [Moh90a, Lom93] to implement V and W key-range locks on B-tree indices. The reason is that V locks allow more concurrency than the exclusive locks considered in [Moh90a, Lom93], so during the period that a transaction *T* holds a V lock on an object, another transaction *T´* may delete this object by acquiring another V lock. To deal with this problem, we introduce a modified key-range locking strategy to implement V and W key-range locks on B-tree indices.

Other interesting properties of the V locking protocol exist because transactions getting V locks on materialized aggregate join views must get S and X locks on the base relations mentioned in their definition. The most interesting such property is that V locks can be used to support "direct propagate" updates to materialized views. Also, by considering the implications of the granularity of V locks and the interaction between base relation locks and accesses to the materialized view, we show that one can define a variant of the V locking protocol, the "no-lock" locking protocol, in which transactions do not set any long-term locks on the materialized view.

The rest of the paper is organized as follows. In Section 2, we explore the split group duplicate problem that arises with a naive use of V locks, and show how this problem can be avoided through the addition of W locks. In Section 3, we explore some thorny issues that arise when B-tree indices over the materialized views are considered. In Section 4, we explore the way V locks can be used to support direct propagate updates and extended to define a "no-lock" locking protocol. In Section 5, we investigate the performance of the V locking protocol through an evaluation in a commercial RDBMS. We conclude in Section 6.

2. The Split Group Duplicate Problem

As mentioned in the introduction, we cannot simply use V locks on aggregate join views, even though the addition operation for the *COUNT*, *SUM*, and *AVG* aggregate operators in the view definitions is both commutative and associative. Recall that the problem is that for the V lock to work correctly, updates must be classified *a priori* into those that update a field in an existing tuple and those that create a new tuple or delete an existing tuple, which cannot be done in the view update scenario. In this section, we illustrate the split group duplicate problem that arises if we ignore this subtle difference between materialized view maintenance and the "traditional" associative/commutative update problems studied by Korth [Kor83] and others. First we illustrate the problem and its solution in the presence of hash indices or in the absence of indices on the materialized view. In Section 3, we consider the problem in the presence of B-tree indices (where its solution is considerably more complex.)

2.1 An Example of Split Groups

In this subsection, we explore an example of the split group duplicate problem in the case that the aggregate join view *AJV* is stored in a hash file implemented as described by Gray and Reuter [GR93]. (The case that the view is stored in a heap file is almost identical; just view the heap file as a hash file with one bucket.) Furthermore, suppose that we are using key-value locking. Suppose the schema of the aggregate join view *AJV* is (*a*, *sum(b)*), where attribute *a* is both the value locking attribute for the view and the hash key for the hash file. Suppose originally the aggregate join view *AJV* contains the tuple (20, 2) and several other tuples, but that there is no tuple whose attribute $a=1$.

Consider the following three transactions *T*, *T´*, and *T˝*. Transaction *T* inserts a new tuple into a base relation *R* and this generates the join result tuple (1, 1), which needs to be integrated into *AJV*. Transaction *T´* inserts another new tuple into the same base relation *R* and generates the join result tuple (1, 2). Transaction *T˝* deletes a third tuple from base relation *R*, which requires the tuple (20, 2) to be deleted from *AJV*. After executing these three transactions, the tuple (20, 2) should be deleted from *AJV* while the tuple (1, 3) should appear in *AJV*.

Now suppose that 20 and 1 have the same hash value so that the tuples (20, 2) and (1, 3) are stored in the same bucket *B* of the hash file. Also, suppose that initially there are four pages in bucket *B*: one bucket page P_1 and three overflow pages P_2, P_3, and P_4, as illustrated in Figure 1. Furthermore, let pages P_1, P_2, and P_3 be full while there are several open slots in page P_4.

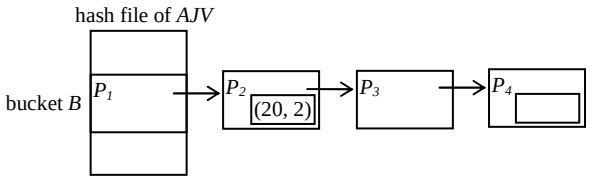

Figure 1. Hash file of the aggregate join view *AJV*.

To integrate a join result tuple t_1 into the aggregate join view *AJV*, a transaction *T* performs the following steps [GR93]:

1. Get an X value lock for $t_1.a$ on *AJV*. This value lock is held until transaction *T* commits/aborts.
2. Apply the hash function to $t_1.a$ to find the corresponding hash table bucket *B*.
3. Crab all the pages in bucket *B* to see whether a tuple t_2 whose attribute $a=t_1.a$ already exists. ("Crabbing" [GR93] means first getting an X semaphore on the

next page, then releasing the X semaphore on the current page.)

4. If tuple t_2 exists in some page P in bucket B, stop the crabbing and integrate the join result tuple t_1 into tuple t_2. The X semaphore on page P is released only after the integration is finished.
5. If tuple t_2 does not exist, crab the pages in bucket B again to find a page P that has enough free space. Insert a new tuple into page P for the join result tuple t_1. The X semaphore on page P is released only after the insertion is finished.

Suppose now that we use V value locks instead of X value locks in this example and that the three transactions T, T', and T'' are executed in the following sequence:

1. First transaction T gets a V value lock for attribute $a=1$, applies the hash function to attribute $a=1$ to find the corresponding hash table bucket B, then crabs all the pages in bucket B to see whether a tuple t_2 whose attribute $a=1$ already exists in the hash file. After crabbing, it finds that no such tuple t_2 exists.
2. Next transaction T' gets a V value lock for attribute $a=1$, applies the hash function to attribute $a=1$ to find the corresponding hash table bucket B, and crabs all the pages in bucket B to see whether a tuple t_2 whose attribute $a=1$ already exists in the hash file. After crabbing, it finds that no such tuple t_2 exists.
3. Next, transaction T crabs the pages in bucket B again, finding that only page P_4 has enough free space. It then inserts a new tuple (1, 1) into page P_4 for the join result tuple (1, 1), commits, and releases the V value lock for attribute $a=1$.

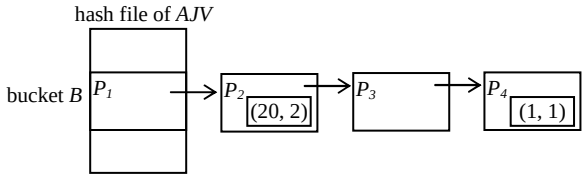

Figure 2. Hash file of the aggregate join view *AJV* – after inserting tuple (1, 1).

4. Then transaction T'' gets a V value lock for attribute $a=20$, finds that tuple (20, 2) is contained in page P_2, and deletes it (creating an open slot in page P_2). Then T'' commits, and releases the V value lock for attribute $a=20$.

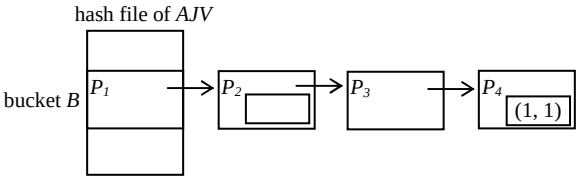

Figure 3. Hash file of the aggregate join view *AJV* – after deleting tuple (20, 2).

5. Finally, transaction T' crabs the pages in bucket B again, and finds that page P_2 has an open slot. It inserts a new tuple (1, 2) into page P_2 for the join result tuple (1, 2), commits, and releases the V value lock for attribute $a=1$.

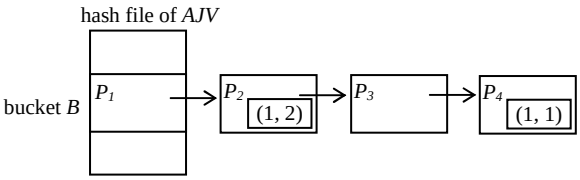

Figure 4. Hash file of the aggregate join view *AJV* – after inserting tuple (1, 2).

Now the aggregate join view *AJV* contains two tuples (1, 1) and (1, 2), whereas it should have only the single tuple (1, 3). This is why we call it the "split group duplicate" problem — the group for "1" has been split into two tuples.

One might think that during crabbing, holding an X semaphore on the entire bucket B could solve the split group duplicate problem. However, there may be multiple pages in the bucket B and some of them may not be in the buffer pool. Normally under all circumstances one tries to avoid performing I/O while holding a semaphore [GR93 page 849]. Thus, holding an X semaphore on the entire bucket for the duration of the operation could cause a substantial performance hit.

2.2 Preventing Split Groups with W Locks

To enable the use of high concurrency V locks while avoiding split group duplicates, we introduce a short-term lock mode, which we call the W lock mode, for aggregate join views. The W lock mode guarantees that for each aggregate group, at any time, at most one tuple corresponding to this group exists in the aggregate join view. With the addition of W locks we now have four kinds of elementary locks: S, X, V, and W.

The compatibilities among these locks are listed in Table 1, while the lock conversion lattice is shown in Figure 5. The W lock mode is only compatible with the V lock mode. A W lock can be either upgraded to an X lock or downgraded to a V lock. (In this respect the W lock is similar to the update mode lock [GR93], which can be either downgraded to an S lock or upgraded to an X lock.)

Table 1. Compatibilities among different locks.

	V	S	X	W
V	yes	no	no	yes
S	no	yes	no	no
X	no	no	no	no
W	yes	no	no	no

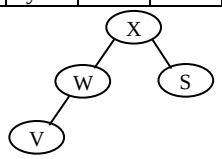

Figure 5. The lock conversion lattice.

In the V+W locking protocol for materialized aggregate join views, S locks are used for reads, V and W locks are used for associative and commutative aggregate update writes, while X locks are used for transactions that do both

reads and writes. These locks can be of any granularity, and, like traditional S and X locks, can be physical locks (e.g., tuple, page, or table locks) or value locks. For fine-granularity locks, it is straightforward to define the corresponding coarser granularity intention locks as introduced in Gray et al. [GLP+76] (see [LNE+03] for details). For example, we can define a coarse granularity IV lock corresponding to a fine-granularity V lock.

Transactions use W locks in the following way:

(1) To integrate a new join result tuple into an aggregate join view *AJV* (e.g., due to insertion into some base relation of *AJV*), we first put a short-term W lock on *AJV*. There are two special cases:
   (a) If the same transaction has already put a V lock on *AJV*, this V lock is upgraded to the W lock.
   (b) If the same transaction has already put an X lock on *AJV*, this W lock is unnecessary.

   After integrating the new join result tuple into the aggregate join view *AJV*, we downgrade the short-term W lock to a long-term V lock that will be held until the transaction commits/aborts.

(2) To remove a join result tuple from the aggregate join view *AJV* (e.g., due to deletion from some base relation of *AJV*), we only need to put a V lock on *AJV*.

In this way, during aggregate join view maintenance, high concurrency is guaranteed by the fact that V locks are compatible with themselves. Note that when using V locks and W locks, multiple transactions may concurrently update the same tuple in the aggregate join view. Thus, logical undo is required on the aggregate join view *AJV* if the transaction updating *AJV* aborts.

The split group duplicate problem cannot occur if the system uses W locks. For a full proof, see [LNE+03]; the intuition behind the proof is that by enumerating all possible cases, we see that the split group duplicate problem will only occur under the following conditions: (1) two transactions integrate two new join result tuples into the aggregate join view *AJV* simultaneously, (2) these two join result tuples belong to the same aggregate group, and (3) no tuple corresponding to that aggregate group currently exists in the aggregate join view *AJV*.

Using the short-term W lock, one transaction, say *T*, must do the update to the aggregate join view *AJV* first (by inserting a new tuple *t* with the corresponding group by attribute value into *AJV*). During the period that transaction *T* holds the short-term W lock, no other transaction can integrate another join result tuple that has the same group by attribute value as tuple *t* into the aggregate join view *AJV*. Then when a subsequent transaction *T´* updates the view, it will see the existing tuple *t*. Thus, transaction *T´* will aggregate its join result tuple that has the same group by attribute value as tuple *t* into tuple *t* (rather than inserting a new tuple into *AJV*).

As mentioned in the introduction, the W lock is similar in some respect to the latches that are used by DBMS to enforce serial updates to concurrently accessed data structures. However, there are some important differences. Unlike latches, W locks must be considered in deadlock detection, because although deadlocks are much less likely with W locks than with long-term X locks, they are still possible. Also, latches are orthogonal to the locking protocol in that they cannot be upgraded or downgraded to any locks (latches are either held or released.) Finally, and perhaps most importantly, the standard use of latches (short-term exclusion on updated data structures) will not prevent the split group duplicate problem efficiently.

2.3 Correctness of the V+W Locking Protocol

Due to space constraints, in this section we sketch the main ideas behind the correctness of this protocol, and refer the reader to [LNE+03] for the full proof. We begin by reviewing our assumptions.

We are considering materialized aggregate join views that can be expressed as $AJV=\gamma(\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)))$, where γ is one of *COUNT*, *SUM*, or *AVG*. Unless otherwise specified, we assume that an aggregate join view *AJV* is maintained in the following way: first compute the join result tuple(s) resulting from the update(s) to the base relation(s) of *AJV*, then integrate these join result tuple(s) into *AJV*. We use strict two-phase locking (except for W locks). We assume that the locking mechanism used by the database system on the base relations ensures serializability in the absence of materialized aggregate join views. Unless otherwise specified, all the locks are long-term locks that are held until transaction commits. Transactions updating the materialized aggregate join view obtain V and W locks as described earlier in this section.

The main ideas used in the proof are:

(1) In the absence of updates that cause the insertion or deletion of tuples in the aggregate join view, the proof of serializability is the same as that in the original P locking paper [Kor83] (since in this case, the combination of V and W locks are indistinguishable from P locks.)

(2) In the presence of updates that cause the insertion or deletion of tuples in the aggregate join view, the short-term W locks guarantee that the "race" conditions that can cause the split group duplicate problem cannot occur; these updates interact correctly with transactions that read but do not update the join view (because to these read transactions, both V and W locks are indistinguishable from X locks.)

Hence the V+W locking protocol ensures that the semantics of the aggregate join view are preserved and that only conflict serializable schedules can occur.

3. V and W Locks and B-Trees

In this section, we consider the particularly thorny problem of implementing V locks (with the required W locks) in the presence of B-tree indices. This section is included for completeness; typically, implementing high concurrency locking modes poses special challenges when

B-trees are considered, and the V+W locks are no exception. However, we wish to warn the reader that this section is rather intricate and perhaps even tedious; for the reader not interested in these details, the rest of the paper can be safely read and understood while omitting this section.

On B-tree indices, we use value locks to refer to key-range locks. To be consistent with the approach advocated by Mohan [Moh90a], we use next-key locking to implement key-range locking. We use "key" to refer to the indexed attribute of the B-tree index. We assume that the entry of the B-tree index is of the following format: (key value, row id list).

3.1 Split Groups and B-Trees

We begin by considering how split group duplicates can arise when a B-tree index is declared over the aggregate join view *AJV*. Suppose the schema of *AJV* is (*a*, *b*, *sum(c)*), and we build a B-tree index I_B on attribute *a*. Also, assume there is no tuple (1, 2, *X*) in *AJV*, for any *X*. Consider the following two transactions *T* and *T′*. Transaction *T* integrates a new join result tuple (1, 2, 3) into the aggregate join view *AJV* (by insertion into some base relation *R*). Transaction *T′* integrates another new join result tuple (1, 2, 4) into the aggregate join view *AJV* (by insertion into the same base relation *R*). Using standard concurrency control without V locks, to integrate a join result tuple t_1 into the aggregate join view *AJV*, a transaction will execute something like the following operations:

(1) Get an X value lock for $t_1.a$ on the B-tree index I_B of *AJV*. This value lock is held until the transaction commits/aborts.
(2) Make a copy of the row id list in the entry for $t_1.a$ of the B-tree index I_B.
(3) For each row id in the row id list, fetch the corresponding tuple t_2. Check whether or not $t_2.a=t_1.a$ and $t_2.b=t_1.b$.
(4) If some tuple t_2 satisfies the condition $t_2.a=t_1.a$ and $t_2.b=t_1.b$, integrate tuple t_1 into tuple t_2 and stop.
(5) If no tuple t_2 satisfies the condition $t_2.a=t_1.a$ and $t_2.b=t_1.b$, insert a new tuple into *AJV* for tuple t_1. Also, insert the row id of this new tuple into the B-tree index I_B.

Suppose now we use V value locks instead of X value locks and the two transactions *T* and *T′* above are executed in the following sequence:

(1) Transaction *T* gets a V value lock for *a=1* on the B-tree index I_B, searches the row id list in the entry for *a=1*, and finds that no tuple t_2 whose attributes $t_2.a=1$ and $t_2.b=2$ exists in *AJV*.
(2) Transaction *T′* gets a V value lock for *a=1* on the B-tree index I_B, searches the row id list in the entry for *a=1*, and finds that no tuple t_2 whose attributes $t_2.a=1$ and $t_2.b=2$ exists in *AJV*.
(3) Transaction *T* inserts a new tuple t_1=(1, 2, 3) into *AJV*, and inserts the row id of tuple t_1 into the row id list in the entry for *a=1* of the B-tree index I_B.
(4) Transaction *T′* inserts a new tuple t_3=(1, 2, 4) into *AJV*, and inserts the row id of tuple t_3 into the row id list in the entry for *a=1* of the B-tree index I_B.

Now the aggregate join view *AJV* contains two tuples (1, 2, 3) and (1, 2, 4) instead of a single tuple (1, 2, 7); hence, we have the split group duplicate problem.

3.2 Implementing V Locking with B-trees

Implementing a high concurrency locking scheme in the presence of indices is difficult, especially if we consider issues of recoverability. Key-value locking as proposed by Mohan [Moh90a] was perhaps the first published description of the issues that arise and their solution. Unfortunately, we cannot directly use the techniques in [Moh90a] to implement V and W as value (key-range) locks.

To illustrate why, we use the following example. Suppose the schema of the aggregate join view *AJV* is (*a*, *sum(b)*), and a B-tree index is built on attribute *a* of the aggregate join view *AJV*. Suppose originally the aggregate join view *AJV* contains four tuples that correspond to *a=2*, *a=3*, *a=4*, and *a=5*. Consider the following three transactions *T*, *T′*, and *T″* that result in updates to the aggregate join view *AJV*. Transaction *T* deletes the tuple whose attribute *a=3* (by deletion from some base relation *R* of *AJV*). Transaction *T′* deletes the tuple whose attribute *a=4* (by deletion from the same base relation *R* of *AJV*). Transaction *T″* reads those tuples whose attribute *a* is between 2 and 5. Suppose we ignore the special properties of V locks and use the techniques in [Moh90a] to implement V and W value locks on the B-tree index. Then the three transactions *T*, *T′*, and *T″* could be executed in the following sequence:

(1) Transaction *T* puts a V lock for *a=3* and another V lock for *a=4* on the aggregate join view *AJV*.

	2	3	4	5
T		V	V	

(2) Transaction *T′* puts a V lock for *a=4* and another V lock for *a=5* on the aggregate join view *AJV*.

	2	3	4	5
T		V	V	
T′			V	V

(3) Transaction *T′* deletes the entry for *a=4* from the B-tree index. Transaction *T′* commits and releases the two V locks for *a=4* and *a=5*.

	2	3		5
T		V	V	

(4) Transaction *T* deletes the entry for *a=3* from the B-tree index.

	2			5
T		V	V	

(5) Before transaction *T* finishes execution, transaction *T″* finds the entries for *a=2* and *a=5* in the B-tree index. Transaction *T″* puts an S lock for *a=2* and another S lock for *a=5* on the aggregate join view *AJV*.

	2			5
T		V	V	
T″	S			S

In this way, transaction T'' can start execution even before transaction T finishes execution. This is not correct, because there is a write-read conflict between transaction T and transaction T'' (on the tuple whose attribute $a=3$). The main reason that this undesirable situation (transactions with write-read conflict can execute concurrently) occurs is due to the fact that V locks are compatible with themselves. Thus, during the period that a transaction holds a V lock on an object, another transaction may delete this object by acquiring another V lock.

To implement V and W value locks on B-tree indices correctly, we need to combine those techniques in [Moh90a, GR93] with the technique of logical deletion of keys [Moh90b, KMH97]. In Section 3.2.1, we describe the protocol for each of the basic B-tree operations in the presence of V locks. In Section 3.2.2, we explore the need for the techniques used in Section 3.2.1. We briefly justify the correctness of the implementation method in Section 3.2.3.

3.2.1 Basic Operations for B-tree Indices

In our protocol, there are five operations of interest:

(1) **Fetch**: Fetch the row ids for a given key value v_1.
(2) **Fetch next**: Given the current key value v_1, find the next key value $v_2>v_1$ existing in the B-tree index, and fetch the row id(s) associated with key value v_2.
(3) **Put an X value lock on key value v_1**.
(4) **Put a V value lock on key value v_1**.
(5) **Put a W value lock on key value v_1**.

Unlike [Moh90a, GR93], we do not consider the operations of insert and delete. We show why this is by an example. Suppose a B-tree index is built on attribute *a* of an aggregate join view *AJV*. Assume we insert a tuple into some base relation of *AJV* and generate a new join result tuple *t*. The steps to integrate the join result tuple *t* into the aggregate join view *AJV* are as follows:

If the aggregate group of tuple *t* exists in *AJV*
    Update the aggregate group in *AJV*;
Else
    Insert a new aggregate group into *AJV* for tuple *t*;

Once again, we do not know whether we need to update an existing aggregate group in *AJV* or insert a new aggregate group into *AJV* until we read *AJV*. However, we do know that we need to acquire a W value lock on *t.a* before we can integrate tuple *t* into the aggregate join view *AJV*. Similarly, suppose we delete a tuple from some base relation of the aggregate join view *AJV*. We compute the corresponding join result tuples. For each such join result tuple *t*, we execute the following steps to remove tuple *t* from the aggregate join view *AJV*:

Find the aggregate group of tuple *t* in *AJV*;
Update the aggregate group in *AJV*;
If all join result tuples have been removed from the aggregate group
    Delete the aggregate group from *AJV*;

In this case, we do not know whether we need to update an aggregate group in *AJV* or delete an aggregate group from *AJV* in advance. However, we do know that we need to acquire a V value lock on *t.a* before we can remove tuple *t* from the aggregate join view *AJV*.

The ARIES/KVL method described in [Moh90a] for implementing value locks on a B-tree index requires the insertion/deletion operation to be done immediately after a transaction gets appropriate locks. Also, in ARIES/KVL, the value lock implementation method is closely tied to the B-tree implementation method. This is because ARIES/KVL strives to take advantage of both IX locks and instant locks to increase concurrency. In the V+W locking mechanism, high concurrency has already been guaranteed by the fact that V locks are compatible with themselves.

We can exploit this advantage so that our method for implementing value locks for aggregate join views on B-tree indices is more general and flexible than the ARIES/KVL method. Specifically, in our method, after a transaction gets appropriate locks, we allow it to execute other operations before it executes the insertion/deletion/update/read operation. Also, our value lock implementation method is only loosely tied to the B-tree implementation method.

Our method for implementing value locks for aggregate join views on B-tree indices is as follows. Consider a transaction *T*.

Op1. Fetch: We first check whether some entry for value v_1 exists in the B-tree index. If such an entry exists, we put an S lock for value v_1 on the B-tree index. If no such entry exists, we find the smallest value v_2 in the B-tree index such that $v_2>v_1$. Then we put an S lock for value v_2 on the B-tree index.

Op2. Fetch next: We find the smallest value v_2 in the B-tree index such that $v_2>v_1$. Then we put an S lock for value v_2 on the B-tree index.

Op3. Put an X value lock on key value v_1: We first put an X lock for value v_1 on the B-tree index. Then we check whether some entry for value v_1 exists in the B-tree index. If no such entry exists, we find the smallest value v_2 in the B-tree index such that $v_2>v_1$. Then we put an X lock for value v_2 on the B-tree index.

Op4. Put a V value lock on key value v_1: We first check whether some entry for value v_1 exists in the B-tree index. If such an entry exists, we put a V lock for value v_1 on the B-tree index. If no entry for value v_1 exists, we find the smallest value v_2 in the B-tree index such that $v_2>v_1$. Then we put an X (not V) lock for value v_2 on the B-tree index.

Op5. Put a W value lock on key value v_1: We first put a W lock for value v_1 on the B-tree index. Then we check whether some entry for value v_1 exists in the B-tree index. If no entry for value v_1 exists, we do the following:

(a) Find the smallest value v_2 in the B-tree index such that $v_2>v_1$. Then we put a short-term W lock for value v_2 on the B-tree index. If the W lock for

value v_2 on the B-tree index is acquired as an X lock, we upgrade the W lock for value v_1 on the B-tree index to an X lock. This situation may occur when transaction *T* already holds an S or X lock for value v_2 on the B-tree index.

(b) We insert into the B-tree index an entry for value v_1 with an empty row id list. Note: that at a later point transaction *T* will insert a row id into this row id list after transaction *T* inserts the corresponding tuple into the aggregate join view.

(c) We release the short-term W lock for value v_2 on the B-tree index.

Table 2 summarizes the locks acquired during different operations.

Table 2. Summary of locking.

		current key v_1	next key v_2
fetch	v_1 exists	S	
	v_1 does not exist		S
fetch next			S
X value lock	v_1 exists	X	
	v_1 does not exist	X	X
V value lock	v_1 exists	V	
	v_1 does not exist		X
W value lock	v_1 exists	W	
	v_1 does not exist and the W lock on v_2 is acquired as a W lock	W	W
	v_1 does not exist and the W lock on v_2 is acquired as an X lock	X	X

During the period that a transaction *T* holds a V (or W, or X) value lock for value v_1 on the B-tree index, if transaction *T* wants to delete the entry for value v_1, transaction *T* needs to do a logical deletion of keys [Moh90b, KMH97] instead of a physical deletion. That is, instead of removing the entry for value v_1 from the B-tree index, it is left there with a *delete_flag* set to 1. If the delete were to be rolled back, then the *delete_flag* is reset to 0. If another transaction inserts an entry for value v_1 into the B-tree index before the entry for value v_1 is garbage collected, the *delete_flag* of the entry for value v_1 is reset to 0.

The physical deletion operations are performed as garbage collection by other operations (of other transactions) that happen to pass through the affected nodes in the B-tree index [KMH97]. A node reorganization removes all such entries from a leaf of the B-tree index that have been marked deleted and currently have no locks on them. This can be implemented in the following way. We introduce a special short-term Z lock mode that is not compatible with any lock mode (including itself). A transaction *T* can get a Z lock on an object if no transaction (including transaction *T* itself) is currently holding any lock on this object. Also, during the period that transaction *T* holds a Z lock on an object, no transaction (including transaction *T* itself) can be granted another lock (including Z lock) on this object.

Note the Z lock mode is different from the X lock mode. For example, if transaction *T* itself is currently holding an S lock on an object, transaction *T* can still get an X lock on this object. That is, transaction *T* can get an X lock on an object if no other transaction is currently holding any lock on this object. For each entry with value *v* whose *delete_flag=1*, we request a conditional Z lock (conditional locks are discussed in [Moh90a]) for value *v*. If the conditional Z lock request is granted, we delete this entry from the leaf of the B-tree index, then we release the Z lock. If the conditional Z lock request is denied, we do not do anything with this entry. Then the physical deletion of this entry is left to other future operations.

We use the Z lock (instead of X lock) to prevent the following undesirable situation: a transaction that is currently using an entry (e.g, holding an S lock on the entry), where the entry is marked logically deleted, tries to physically delete this entry. Z locks can be implemented easily using the techniques in [GR93, Chapter 8] (by making small changes to the lock manager). Note the above method is different from the method described in [Moh90b] while both methods work. We choose the Z lock method to simplify our key-range locking protocol for aggregate join views on B-tree indices. As mentioned in [Moh90b], the log record for garbage collection is a redo-only log record.

In Op4 (put a V value lock on key value v_1), the situation that no entry for value v_1 exists in the B-tree index does not often occur. To illustrate this, consider an aggregate join view *AJV* that is defined on base relation *R* and several other base relations. Suppose a B-tree index I_B is built on attribute *d* of the aggregate join view *AJV*. If we insert a new tuple *t* into base relation *R* and generate several new join result tuples, we need to acquire appropriate W value locks on the B-tree index I_B before we can integrate these new join result tuples into the aggregate join view *AJV*. If we delete a tuple *t* from base relation *R*, to maintain the aggregate join view *AJV*, normally we need to first compute the corresponding join result tuples that are to be removed from the aggregate join view *AJV*. These join result tuples must have been integrated into the aggregate join view *AJV* before. Thus, when we acquire V value locks for their *d* attribute values, these *d* attribute values must exist in the B-tree index I_B.

However, there is an exception. Suppose attribute *d* of the aggregate join view *AJV* comes from base relation *R*. Consider the following scenario (see Section 4 below for details). There is only one tuple *t* in base relation *R* whose attribute *d=v*. However, there is no matching tuple in the other base relations of the aggregate join view *AJV* that can be joined with tuple *t*. Thus, there is no tuple in the aggregate join view *AJV* whose attribute *d=v*. Suppose transaction *T* executes the following SQL statement:

delete from *R* where *R.d=v*;

In this case, to maintain the aggregate join view *AJV*, there is no need for transaction *T* to compute the corresponding join result tuples that are to be removed from the aggregate join view *AJV*. Transaction *T* can execute the following "direct propagate" update operation:

delete from *AJV* where *AJV.d*=*v*;

Then when transaction *T* requests a V value lock for *d*=*v* on the B-tree index I_B, transaction *T* will find that no entry for value *v* exists in the B-tree index I_B. We will return to direct propagate updates in Section 4.

3.2.2 Are These Techniques Necessary?

The preceding section is admittedly dense and intricate, so it is reasonable to ask if all this effort is really necessary. Unfortunately the answer appears to be yes — if any of the techniques from the previous section are omitted (and not replaced by other equivalent techniques), then we cannot guarantee serializability. (The reason why serializability is guaranteed by our techniques is shown in the correctness proof in Section 3.2.3.) Due to space constraints, we refer the reader to [LNE+03] for detailed examples illustrating the necessity of these techniques.

3.2.3 Sketch of Correctness

Due to space constraints, we only briefly justify the correctness (serializability) of our key-range locking strategy for aggregate join views on B-tree indices. A formal complete correctness proof is available in [LNE+03]. Suppose a B-tree index I_B is built on attribute *d* of an aggregate join view *AJV*. To prove serializability, for any value *v* (no matter whether or not an entry for value *v* exists in the B-tree index, i.e., the phantom problem [GR93] is also considered), we only need to show that there is no read-write, write-read, or write-write conflict between two different transactions on those tuples of the aggregate join view *AJV* whose attribute *d* has value *v* [BHG87, GR93]. As shown in [Kor83], write-write conflicts are avoided by the commutative and associative properties of the addition operation. Furthermore, the use of W locks guarantees that for each aggregate group, at any time at most one tuple corresponding to this group exists in the aggregate join view *AJV*. To show that write-read and read-write conflicts do not exist, we enumerate all the possible cases: whether an entry for value *v* exists on the B-tree index or not, which transaction gets the value lock on value *v* first, and so on. Since we use next-key locking, in the enumeration, we only need to focus on value *v* and the smallest existing value *v*′ in the B-tree index I_B such that $v' > v$.

4. Other Uses and Extensions of V Locks

In this section we briefly discuss two other interesting aspects of using V locks for materialized view maintenance: the possibility of supporting direct propagate updates, and how observations about the appropriate granularity of V locks illustrate the possibility of a locking protocol for materialized views that supports serializability without requiring any long-term locks whatsoever on the views.

4.1 Direct Propagate Updates

In the preceding sections of this paper, with one exception at the end of Section 3.2.1, we have assumed that materialized aggregate join views are maintained by first computing the join of the newly updated (inserted, deleted) tuples with the other base relations, then aggregating these join result tuples into the aggregate join view. In this section we will refer to this approach as the "indirect approach" to updating the materialized view. However, in certain situations, it is possible to propagate updates on base relations directly to the materialized view, without computing any join. As we know of at least one commercial system that supports such direct propagate updates, in this section we investigate how they can be handled in our framework.

Direct propagate updates are perhaps most useful in the case of (non-aggregate) join views, so we consider join views in the following discussion. (Technically, we do not need to mention the distinction between join views and aggregate join views, since non-aggregate join views are really included in our general class of views – recall that we are considering views $AJV=\gamma(\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)))$. If the aggregate operator γ in this formula has the effect of putting every tuple of the enclosed project-select-join in its own group, then what we have is really a non-aggregate join view.) However, the same discussion holds for direct propagate updates to aggregate join views.

Our focus in this paper is not to explore the merits of direct propagate updates or when they apply; rather, it is to see how they can be accommodated by the V locking protocol. We begin with an example. Suppose we have two base relations, *A*(*a*, *b*, *c*) and *B*(*d*, *e*, *f*). Consider the following join view:

create join view *JV* as
select *A.a*, *A.b*, *B.e*, *B.f* from *A*, *B* where *A.c*=*B.d*;

Next consider a transaction *T* that executes the following SQL statement:

update *A* set *A.b*=2 where *A.a*=1;

To maintain the join view, transaction *T* only needs to execute the following operation (without performing a join with base relation *B*):

update *JV* set *JV.b*=2 where *JV.a*=1;

This is a "direct propagate" update, since transaction *T* does not compute a join to maintain the view. Similarly, suppose that a transaction *T*′ executes the following SQL statement:

update *B* set *B.e*=4 where *B.f*=3;

To maintain *JV*, transaction *T*′ can also do a direct propagate update with the following operation:

update *JV* set *JV.e*=4 where *JV.f*=3;

If these transactions naively use V locks on the materialized view, there is apparently a problem: since two V locks do not conflict, *T* and *T*′ can execute

concurrently. This is not correct, since there is a write-write conflict between *T* and *T´* on any tuple in *JV* with *a=1* and *f=3*. This could lead to a non-serializable schedule.

One way to prevent this would be to require all direct propagate updates to get X locks on the materialized view tuples that they update while indirect updates still use V locks. While this is correct, it is also possible to use V locks for the direct updates if we require that transactions that update base relations in materialized view definitions get X locks on the tuples in the base relations they update and S locks on the other base relations mentioned in the view definition. Note that these are exactly the locks the transactions would acquire if they were using indirect materialized view updates instead of direct propagate updates.

Informally, this approach with V locks works because updates to materialized views (even direct propagate updates) are not arbitrary; rather, they must be preceded by updates to base relations. So if two transactions using V locks would conflict in the join view on some tuple *t*, they must conflict on one or more of the base relations updated by the transactions, and locks at that level will resolve the conflict.

In our running example, *T* and *T´* would conflict on base relation *A* (since *T* must get an X lock and *T´* must get an S lock on the same tuples in *A*) and/or on base relation *B* (since *T* must get an S lock and *T´* must get an X lock on the same tuples in *B*.) Note that these locks could be tuple-level, or table-level, or anything in between, depending on the specifics of the implementation. We refer the reader to [LNE+03] for a formal complete correctness proof of this approach.

Unlike the situation for indirect updates to materialized aggregate join views, for direct propagate updates the V lock will not result in increased concurrency over X locks. Our point here is to show that we do not need special locking techniques to handle direct propagate updates: the transactions obtain locks as if they were doing updates indirectly (X locks on the base relations they update, S locks on the base relations with which they join, and V locks on the materialized view.) Then the transactions can use either update approach (direct or indirect) and still be guaranteed of serializability.

4.2 Granularity and the No-Lock Locking Protocol

Throughout the discussion in this paper we have been purposely vague about the granularity of locking. This is because the V lock can be implemented at any granularity; the appropriate granularity is a question of efficiency, not of correctness. V locks have some interesting properties with respect to granularity and concurrency, which we explore in this section.

In general, finer granularity locking results in higher concurrency. This is not true of V locks if we consider only transactions that update the materialized views. The reason is that V locks do not conflict with one another, so that a single table-level V lock on a materialized view is the same, with respect to concurrency of update transactions, as many tuple-level V locks on the materialized view.

This is not to say that a single table-level V lock per materialized view is a good idea; indeed, a single table-level V lock will block all readers of the materialized view (since it looks like an X lock to any transaction other than an updater also getting a V lock.) Finer granularity V locks will let readers of the materialized view proceed concurrently with updaters (if, for example, they read tuples that are not being updated.) In a sense, a single V lock on the view merely signals "this materialized view is being updated;" read transactions "notice" this signal when they try to place S locks on the view.

This intuition can be generalized to produce a protocol for materialized views that requires no long-term locks at all on the materialized views. In this protocol, the function provided by the V lock on the materialized view (letting readers know that the view is being updated) is implemented by X locks on the base relations. The observation that limited locking is possible when data access patterns are constrained was exploited in a very different context (locking protocols for hierarchical database systems) in [SK80].

In the no-lock locking protocol, like the V locking protocol, updaters of the materialized view must get X locks on the base relations they update and S locks on other base relations mentioned in the view. To interact appropriately with updaters, readers of the materialized view are required to get S locks on all the base relations mentioned in the view. If the materialized view is being updated, there must be an X lock on one of the base relations involved, so the reader will block on this lock. Updaters of the materialized view need not get V locks on the materialized view (since only they would be obtaining locks on the view, and they do not conflict with each other), although they do require short-term W locks to avoid the split group duplicate problem.

It seems unlikely that in a practical situation this no-lock locking protocol would yield higher performance than the V locking protocol. The no-lock locking protocol benefits updaters (who do not have to get V locks) at the expense of readers (who have to get multiple S locks.) However, we present it here as an interesting application of how the semantics of materialized view updates can be exploited to reduce locking while still guaranteeing serializability.

5. Performance of the V Locking Protocol

In this section, we describe experiments that were performed on a commercial parallel RDBMS. We focus on the throughput of a targeted class of transactions (i.e., transactions that update a base relation of an aggregate join view). This is because in a mixed workload

environment, our V locking protocol would greatly improve the throughput of the targeted class of transactions while the throughput of other classes of transactions would remain much the same. Our measurements were performed with the database client application and server running on an Intel x86 Family 6 Model 5 Stepping 3 workstation with four 400MHz processors, 1GB main memory, six 8GB disks, and running the Microsoft Windows 2000 operating system. We allocated a processor and a disk for each data server, so there were at most four data servers on each workstation.

5.1 Benchmark Description

We used the two relations *lineitem* and *partsupp* and the aggregate join view *suppcount* that are mentioned in the introduction for the tests. The schemas of the *lineitem* and *partsupp* relations are listed as follows:

lineitem (orderkey, partkey, price, discount, tax, orderdate, comment)

partsupp (partkey, suppkey, supplycost, comment)

The underscore indicates the partitioning attributes. The aggregate join view *suppcount* is partitioned on the *suppkey* attribute. For each relation, we built an index on the partitioning attribute. In our tests, different *partsupp* tuples have different *partkey* values. There are *R* different *suppkey*s, each corresponding to the same number of tuples in the *partsupp* relation.

Table 3. Test data set.

	number of tuples	total size
lineitem	8M	586MB
partsupp	0.25M	29MB

We used the following kind of transaction for the testing:

T: Insert *r* tuples that have a specific *orderkey* value into the *lineitem* relation. Each of these *r* tuples has a different and random *partkey* value and matches a *partsupp* tuple on the *partkey* attribute. Each of these *r* matched *partsupp* tuples has a different (and thus random) *suppkey* value.

We evaluated the performance of our V lock method and the traditional X lock method in the following way:

(1) We tested our largest available hardware configuration with four data server nodes. This is to prevent certain system resources (e.g., disk I/Os) from becoming a bottleneck too easily in the presence of high concurrency.

(2) We ran *x T*'s. Each of these *x T*'s has a different *orderkey* value. *x* is an arbitrarily large number. Its specific value does not matter, as we only focus on the throughput of the RDBMS.

(3) In the X lock method, if a transaction deadlocked and aborted, we automatically re-executed it until it committed.

(4) We used the tuple throughput (number of tuples inserted successfully per second) as the performance metric. It is easy to see that the transaction throughput = the tuple throughput / *r*. In the rest of Section 5, we use throughput to refer to the tuple throughput.

(5) We performed the following test. We fixed *R=3,000*. In both the V lock method and the X lock method, we tested four cases: *m=2*, *m=4*, *m=8*, and *m=16*, where *m* is the number of concurrent transactions. In each case, we let *r* vary from 1 to 64.

(6) We could not implement our V locking protocol in the database software, as we did not have access to the source code. Since the essence of the V locking protocol is that V locks do not conflict with each other, we used the following method to evaluate the performance of the V lock method. We created *m* copies of the aggregate join view *suppcount*. At any time, each of the *m* concurrent transactions dealt with a different copy of *suppcount*. Using this method, our testing results of the V lock method would show slightly different performance from that of an actual implementation of the V locking protocol. This is because in an actual implementation of the V locking protocol, we would encounter the following issues:

   (a) Short-term X page latch conflicts and W lock conflicts during concurrent updates to the aggregate join view *suppcount*.

   (b) Hardware cache invalidation in an SMP environment during concurrent updates to the aggregate join view *suppcount*.

   However, we believe that these issues are minor compared to the substantial performance improvements gained by the V lock method over the X lock method (see Section 5.2 below for details). The general trend shown in our testing results should be close to that of an actual implementation of the V locking protocol.

5.2 Test Results

As mentioned in the introduction, for the X lock method, we can use the unified formula $min(1, (m-1)(r-1)^4/(4R^2))$ to roughly estimate the probability that any particular transaction deadlocks. We validated this formula in our tests. Due to space constraints, we refer the reader to [LNE+03] for detailed testing results.

For the X lock method, to see how deadlocks influence performance, we investigated the relationship between the throughput and the deadlock probability as follows. It is easy to see that for the X lock method, when the deadlock probability becomes close to 1, almost every transaction will deadlock. Deadlock has the following negative influences on throughout:

(1) Deadlock detection/resolution is a time-consuming process. During this period, the deadlocked transactions cannot make any progress.

(2) The deadlocked transactions will be aborted and re-executed. During re-execution, these transactions may deadlock again. This wastes system resources.

Thus, once the system starts to deadlock, the deadlock problem tends to become worse and worse. Eventually,

the X lock method runs into a severe deadlock problem and its throughput becomes significantly deteriorated.

Due to space constraints, we only show the ratio of the throughput of the V lock method over that of the X lock method in Figure 6. (Note: Figure 6 uses logarithmic scale for both the x-axis and the y-axis.) More detailed testing results (including testing results for other test settings) are available in [LNE+03]. Before the X lock method runs into the deadlock problem, the throughput of the V lock method is the same as that of the X lock method. However, when the X lock method runs into the deadlock problem, the throughput of the V lock method does not drop while the throughput of the X lock method is significantly worse. In this case, the ratio of the throughput of the V lock method to that of the X lock method is greater than 1. For example, when *r=32*, for any *m*, this ratio is at least 1.3. When *r=64*, for any *m*, this ratio is at least 3. In general, when the X lock method runs into the deadlock problem, this ratio increases with both *m* and *r*.

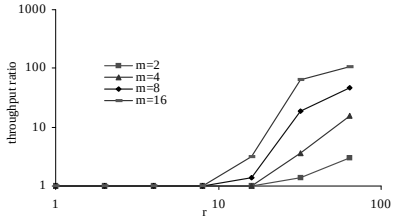

Figure 6. Throughput improvement gained by the V lock method.

6. Conclusion

The V locking protocol is designed to support concurrent, immediate updates of materialized aggregate join views without engendering the high lock conflict rates and high deadlock rates that could result if two-phase locking with S and X lock modes were used. This protocol borrows from the theory of concurrency control for associative and commutative updates, with the addition of a short-term W lock to deal with insertion anomalies that result from some special properties of materialized view updates. Perhaps surprisingly, due to the interaction between locks on base relations and locks on the materialized view, this locking protocol, designed for concurrent update of aggregates, also supports direct propagate updates to (non-aggregate) join views.

It is an open question whether or not immediate updates with serializable semantics are a good idea in the context of materialized views. Certainly there are advantages to deferred updates, including potential efficiencies from the batching of updates and shorter path lengths for transactions that update base relations mentioned in materialized views. However, these efficiencies must be balanced against the semantic uncertainty and the "stale data" problems that may result when materialized views are not "in synch" with base data. The best answer to this question will only be found through a thorough exploration of how well both approaches (deferred and immediate) can be supported; it is our hope that the techniques in this paper can contribute to the discussion in this regard.

Acknowledgements

We would like to thank C. Mohan and Henry F. Korth for useful discussions. This work was supported by the NCR Corporation and also by NSF grants CDA-9623632 and ITR 0086002.

References

[BHG87] P.A. Bernstein, V. Hadzilacos, N. Goodman. Concurrency Control and Recovery in Database Systems. Addison-Wesley publishers, 1987.

[BR92] B.R. Badrinath, K. Ramamritham. Semantics-Based Concurrency Control: Beyond Commutativity. TODS 17(1): 163-199, 1992.

[GK85] D. Gawlick, D. Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. Database Engineering Bulletin 8(2): 3-10, 1985.

[GKS01] J. Gehrke, F. Korn, and D. Srivastava. On Computing Correlated Aggregates over Continual Data Streams. SIGMOD Conf. 2001: 13-24.

[GLP+76] J. Gray, R.A. Lorie, and G.R. Putzolu et al. Granularity of Locks and Degrees of Consistency in a Shared Data Base. IFIP Working Conference on Modeling in Data Base Management Systems 1976: 365-394.

[GM99] A. Gupta, I.S. Mumick. Materialized Views: Techniques, Implementations, and Applications. MIT Press, 1999.

[GR93] J. Gray, A. Reuter. Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, 1993.

[KLM+97] A. Kawaguchi, D.F. Lieuwen, and I.S. Mumick et al. Concurrency Control Theory for Deferred Materialized Views. ICDT 1997: 306-320.

[KMH97] M. Kornacker, C. Mohan, and J.M. Hellerstein. Concurrency and Recovery in Generalized Search Trees. SIGMOD Conf. 1997: 62-72.

[Kor83] H.F. Korth. Locking Primitives in a Database System. JACM 30(1): 55-79, 1983.

[LNE+03] G. Luo, J.F. Naughton, and C.J. Ellmann et al. Locking Protocols for Materialized Aggregate Join Views. Full version, in preparation, will be available at http://www.cs.wisc.edu/~gangluo/locks_full.pdf.

[Lom93] D.B. Lomet. Key Range Locking Strategies for Improved Concurrency. VLDB 1993: 655-664.

[ML92] C. Mohan, F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. SIGMOD Conf. 1992: 371-380.

[Moh90a] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. VLDB 1990: 392-405.

[Moh90b] C. Mohan. Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. VLDB 1990: 406-418.

[O86] P.E. O'Neil. The Escrow Transactional Method. TODS 11(4): 405-430, 1986.

[PF00] M. Poess, C. Floyd. New TPC Benchmarks for Decision Support and Web Commerce. SIGMOD Record 29(4): 64-71, 2000.

[RAA94] R.F. Resende, D. Agrawal, and A.E. Abbadi. Semantic Locking in Object-Oriented Database Systems. OOPSLA 1994: 388-402.

[Reu82] A. Reuter. Concurrency on High-trafic Data Elements. PODS 1982: 83-92.

[SK80] A. Silberschatz, Z.M. Kedem. Consistency in Hierarchical Database Systems. JACM 27(1): 72-80, 1980.