Locking Protocols for Materialized Aggregate Join Views

Gang Luo, Jeffrey F. Naughton, Curt J. Ellmann, and Michael W. Watzke

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 high concurrency locking techniques do not fully solve the problem, but a combination of a "value-based" latch pool and these previous high concurrency locking techniques can solve the problem.

Index Terms—Concurrency, relational databases, transaction processing.

1 Introduction

Although materialized view maintenance has been well-studied in the research literature [7], 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 *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.

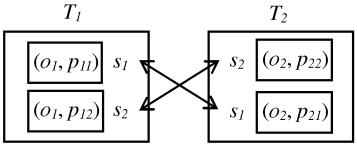
Suppose now that we create a materialized aggregate join view *suppcount* to provide quick access to the number of parts ordered from each supplier, defined as follows:

```
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 [8, pp. 428-429], if $mr \ll R$, the probability that any particular transaction deadlocks is approximately $(m-1)(r-1)^4/(4R^2)$. (If we do not have $mr \ll R$, then the probability of deadlock is essentially one.) 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 percent. Merely doubling m to 16 raises this probability to 38 percent.

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 and apply them later in separate transactions. This "works"; unfortunately, it requires that the system give up on serializability and/or recency (it is possible to provide a

- *G. Luo is with the IBM T.J. Watson Research Center, Hawthorne, NY 10532. E-mail: luog@us.ibm.com.*
- *J.F. Naughton is with the Department of Computer Sciences, University of Wisconsin-Madison, Madison, WI 53705. E-mail: naughton@cs.wisc.edu.*
- *C.J. Ellmann is with the Division of Information Technology, University of Wisconsin-Madision, Madison, WI 53705. E-mail: ellmann@wisc.edu.*
- *M.W. Watke is with NCR, Madison, WI 53719. E-mail: michael.watzke@ncr.com.*

Manuscript received 22 Apr. 2004; revised 24 Aug. 2004; accepted 17 Jan. 2005; published online 20 Apr. 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0116-0404.

theory of serializability in the presence of deferred updates if readers of the materialized view are allowed to read old versions of the view [9]). 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 [14], 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/updates—e.g., when the MIN/MAX value is deleted, we need to compute the new MIN/MAX value using all the values in the aggregate group [4]), we restrict our attention to the three incrementally updateable aggregate operators: $COUNT$, SUM, and AVG. Note that:

1. In practice, AVG is computed using $COUNT$ and SUM as $AVG = SUM/COUNT$ ($COUNT$ and SUM are distributive, while AVG is algebraic [5]). In the rest of the paper, we only discuss $COUNT$ and SUM, while our locking techniques for $COUNT$ and SUM also apply to AVG.
2. By letting $n = 1$ in the definition of AJV, we include aggregate views over single relations.

A useful observation is that, for $COUNT$ and SUM, 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. This line of reasoning leads one to consider locking mechanisms that increase concurrency for associative and commutative operations.

Many special locking modes that support increased concurrency through the special treatment of "hot spot" aggregates in base relations [3], [13], [16] or by exploiting update semantics [2], [15] have been proposed. An early and particularly relevant example of locks that exploit update semantics was proposed by Korth [10]. 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 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 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 special locks (e.g., P locks) in [10].

Unfortunately, purely using V locks cannot fully solve the materialized aggregate join view update problem. Rather, we could end up with what we call "split group duplicates"—multiple tuples in the aggregate join view for the same group, as shown in Section 2 below. To solve the split group duplicate problem, we augment V locks with a "value-based" latch pool. (We will explain what "value-based" means in the next paragraph.) With this pool of latches, 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 latches are only held for a short period and V locks do not conflict with each other. Hence, the combination of V locks and the latch pool solves the materialized aggregate join view update problem. Note: In a preliminary version of this work [12], we used W locks to solve the split group duplicate problem. The latch pool solution is better than the W lock solution, as acquiring a latch is much cheaper than acquiring a lock [8].

Traditionally, latches are used to protect the physical integrity of certain data structures (e.g., the data structures in a page [8]). In our case, no physical data structure would be corrupted if the latch pool were not used. The latch pool is used to protect the logical integrity of aggregate operations rather than the physical integrity of the database. This is why, in the previous paragraph, we use the term "value-based" latch pool.

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. Based on similar reasoning, we show that the V locking protocol also applies to materialized nonaggregate join views and can yield higher concurrency than the traditional X locking protocol in certain cases.

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 a latch pool. In Section 3, we explore the way V locks can be used to support both direct propagate updates and materialized nonaggregate join view maintenance. We also extend V locks to define a "no-lock" locking protocol. In Section 4, we prove the correctness of the V locking protocol. In Section 5, we

investigate the performance of the V locking protocol through a simulation study 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. This is because, 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. For example, consider a materialized aggregate join view AJV. The associative and commutative update operations on AJV are of the following two forms:

1. Suppose 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;

2. 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 t from the aggregate join view:

   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;

Hence, 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. This is different from the case in [10], where updates are 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. 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 2.1, we may run into the split group duplicate problem and the semantics of the aggregate join view may be violated. (The split group duplicate problem is mainly due to the self-compatibility of V locks. Previous approaches for handling "hot spot" aggregates [2], [3], [13], [15], [16] all use some kind of self-compatible lock modes. Hence, due to a similar reason, they cannot be applied to materialized aggregate join views.)

2.1 An Example of Split Groups

In this section, 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 [8]. (The case that the view is stored in a heap file is almost identical.) 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 $a = 1$.

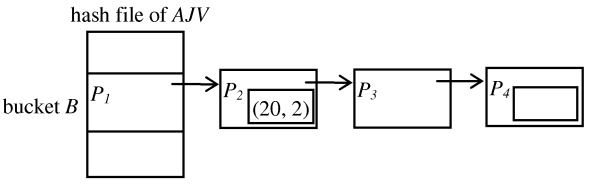


Fig. 1. Hash file of the aggregate join view AJV.

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 Fig. 1. Furthermore, let pages P_1, P_2, and P_3 be full, while there are several open slots in page P_4.

To integrate a join result tuple t_1 into the aggregate join view AJV, a transaction T performs the following steps [8]:

1. Get an X value lock for $t_1.a$ on AJV. This lock is held until 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" [8] means first getting an X latch on the next page, then releasing the X latch on the current page.)
4. If 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 latch 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.

Note that the above description is simplified compared to that in [8]. In general, as described in [8, p. 850], to request an X latch on a page, we first issue a *bufferfix* request without holding the latch. After the page is fixed in the buffer pool, we issue the latch request. This is to avoid performing I/O while holding a latch [8, p. 849].

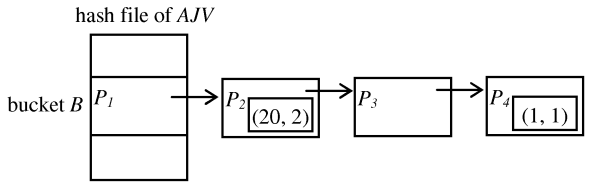


Fig. 2. Hash file of the aggregate join view AJV—after inserting tuple (1, 1).

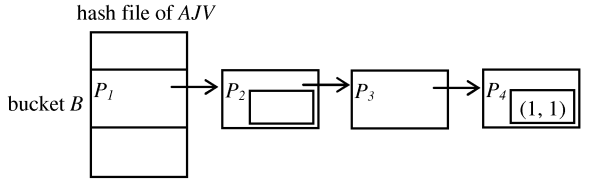


Fig. 3. Hash file of the aggregate join view AJV—after deleting tuple (20, 2).

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. T gets a V value lock for attribute $a = 1$, applies the hash function to find the corresponding hash table bucket B, then crabs all the pages in 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 $a = 1$, as illustrated in Fig. 2.
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), as illustrated in Fig. 3. Then, T'' commits, and releases the V value lock for attribute $a = 20$.
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 $a = 1$, as illustrated in Fig. 4.

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 latch 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 latch [8, p. 849]. Hence, holding an X latch on the entire bucket for the duration of the operation could cause a substantial performance hit.

2.2 Preventing Split Groups with Latches

2.2.1 The Latch Pool

To enable the use of V locks while avoiding split group duplicates, we introduce a latch pool for aggregate join views. The latches in the latch pool guarantee that for each aggregate group, at any time, at most one tuple corresponding to this group exists in the aggregate join view.

For efficiency we preallocate a latch pool that contains $N > 1$ X (exclusive) latches. We use a hash function H that maps key values into integers between 1 and N. We use requesting/releasing a latch on key value v to mean requesting/releasing the $H(v)$th latch in the latch pool.

We ensure that the following properties always hold for this latch pool:

1. During the period that a transaction holds a latch in the latch pool, this transaction does not request another latch in the latch pool.
2. To request a latch in the latch pool, a transaction must first release all the other latches in the RDBMS (including those latches that are not in the latch pool) that it currently holds.
3. During the period that a transaction holds a latch in the latch pool, this transaction does not request any lock.

Properties 1 and 2 guarantee that there are no deadlocks between latches. Property 3 guarantees that there are no deadlocks between latches and locks. These properties are necessary since, in an RDBMS, latches are not considered in deadlock detection.

We define a *false latch conflict* as one that arises due to hash conflicts (i.e., $H(v_1) = H(v_2)$ and $v_1 \neq v_2$). The value of N only influences the efficiency of the V locking protocol—the larger the number N, the smaller the probability of having false latch conflicts. It does not affect the correctness of the V locking protocol. In practice, if we use a good hash function [8] and the number N is substantially larger than the number of concurrently running transactions in the RDBMS, the probability of having false latch conflicts should be small. For example, consider the example in the Introduction with m concurrent transactions. Suppose H is a perfectly randomized hash function and that a transaction spends f percent of its execution on holding a latch in the latch pool. Note that f percent is a small fraction, as a latch is only held for a short period. Then, following a reasoning similar to that in [8, pp. 428-429], we can show that, when a

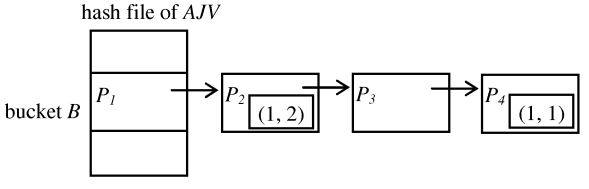


Fig. 4. Hash file of the aggregate join view AJV—after inserting tuple (1, 2).

TABLE 1
Compatibilities among the Elementary Locks

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

transaction requests a latch in the latch pool, the probability that it runs into false latch conflict $\approx (m-1) \times f\%/N$.

While holding a latch in the latch pool, we allow I/Os to be performed. This violates the rule according to which latches are usually used [8, p. 849]. We think this is acceptable because, in our case, each latch in the latch pool is of a fine granularity: Each latch protects only one (in the absence of hash conflicts) or multiple aggregate groups (in the presence of hash conflicts) in the aggregate join view rather than one or multiple pages.

2.2.2 The V Locking Protocol

In the V locking protocol for materialized aggregate join views, we have three kinds of elementary locks: S, X, and V. The compatibilities among these locks are listed in Table 1, while the lock conversion lattice is shown in Fig. 5.

In the V locking protocol for materialized aggregate join views, S locks are used for reads, V 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, we define the corresponding coarser granularity intention locks [6] as follows: We define an IV lock corresponding to a V lock. The IV lock is similar to the traditional IX lock except that it is compatible with the V lock. For a fine granularity X (S) lock, we use the traditional IX (IS) locks. One can think that IX = IS+IV and X = S+V, as X locks are used for transactions that do both reads and writes, while S/V locks are used for transactions that do reads/writes. We introduce the SIV lock (S + IV) that is similar to the traditional SIX lock, i.e., the SIV lock is only compatible with the IS lock. Note that SIX = S + IX = S + (IS + IV) = (S + IS) + IV=S + IV = SIV, so we do not introduce the SIX lock, as it is the same as the SIV lock. Similarly, we introduce the VIS lock (V + IS) that is only compatible with the IV lock. Note that VIX = V + IX = V + (IS + IV) = (V + IV) + IS = V + IS = VIS, so we do not introduce the VIX lock, as it is the same as the VIS lock. All these intention locks are used in the same way as that in [6].

The compatibilities among the coarse granularity locks are listed in Table 2, while the lock conversion lattice is shown in Fig. 6. Since the use of intention locks is well understood, we do not discuss intention locks further in the rest of this paper.

TABLE 2
Compatibilities among the Coarse Granularity Locks

	S	X	V	IS	IX	IV	SIV	VIS
S	yes	no	no	yes	no	no	no	no
X	no	no	no	no	no	no	no	no
V	no	no	yes	no	no	yes	no	no
IS	yes	no	no	yes	yes	yes	yes	no
IX	no	no	no	yes	yes	yes	no	no
IV	no	no	yes	yes	yes	yes	no	yes
SIV	no	no	no	yes	no	no	no	no
VIS	no	no	no	no	no	yes	no	no

2.2.3 Using Latches in the Latch Pool

Transactions use the latches in the latch pool in the following way:

1. To integrate a new join result tuple t into an aggregate join view AJV (e.g., due to insertion into some base relation of AJV), we first put a V lock on AJV that will be held until the transaction commits/aborts. Immediately before we start the tuple integration, we request a latch on the group by attribute value of tuple t. After integrating tuple t into the aggregate join view AJV, we release the latch.
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 that will be held until the transaction commits/aborts.

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

The split group duplicate problem cannot occur because of our use of latches. The reason is as follows: 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.

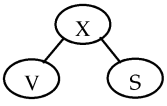


Fig. 5. The lock conversion lattice of the elementary locks.

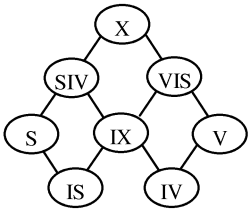


Fig. 6. The lock conversion lattice of the coarse granularity locks.

3. No tuple corresponding to that aggregate group currently exists in the aggregate join view AJV.

Using the latch in the latch pool, 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 latch on the group by attribute value of t, 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. Hence, 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). We refer the reader to Section 4 for the correctness proof of the V locking protocol.

3 Other Uses and Extensions of V Locks

In this section, we briefly discuss three other interesting aspects of using V locks for materialized view maintenance. In Section 3.1, we discuss the possibility of supporting direct propagate updates. In Section 3.2, we show how 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. In Section 3.3, we describe how to apply the V locking protocol to nonaggregate join views.

3.1 Direct Propagate Updates

In the preceding sections of this paper, 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 (Teradata) 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 (nonaggregate) join views, so we consider join views in the following discussion. 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:

delete from A where $A.a = 1$;

To maintain the join view, transaction T only needs to execute the following:

delete from JV 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, 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 nonserializable 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 corresponding tuples in the other base relations mentioned in the view definition. Note that:

1. These are exactly the locks the transactions would acquire if they were using indirect materialized view updates instead of direct propagate updates.
2. For indirect materialized view updates, the X and S locks on the base relations may cause deadlocks among transactions that update different base relations. However, S-X deadlocks on the base relations are usually not as severe as X-X deadlocks on the aggregate join view, as base relations often contain many more tuples than the aggregate join view. Moreover, the V locking protocol at least removes the X-X deadlocks on the aggregate join view.

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 and/or on base relation B. Note that these locks could be tuple-level, or table-level, or anything in between, depending on the specifics of the implementation. A formal complete correctness proof of this approach can be done easily by making minor changes to the proof in Section 4.

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 tuples of the base relations they update, S locks on the tuples of the base relations with which they join, and V locks on the tuples of the materialized view). Then, the transactions can use either update approach (direct or indirect) and still be guaranteed of serializability.

3.2 Granularity and the No-Lock Locking Protocol

Unless otherwise specified, throughout the discussion in this paper we have been purposely vague about the granularity of locking. This is because the locks that we discuss and propose in this paper can be implemented at any granularity; the appropriate granularity is a question of efficiency, not of correctness. However, 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. 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 different context (locking protocols for hierarchical database systems) in [17].

In the no-lock locking protocol, like the V locking protocol, updaters of the materialized view must get X locks on the tuples in the base relations they update and S locks on the tuples in the other base relations mentioned in the view. To interact appropriately with updaters, readers of the materialized view are required to get table-level S locks on all the base relations mentioned in the view. If the materialized view is being updated, there must be a table-level X (IX or SIX) 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 the latches in the latch pool 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 as, in the no-lock locking protocol, readers and updaters of the materialized view cannot run concurrently. However, we present the no-lock locking protocol here as an interesting application of how the semantics of materialized view updates can be exploited to reduce locking on the materialized view while still guaranteeing serializability.

3.3 Applying the V Locking Protocol to Nonaggregate Join Views

Besides aggregate join views, the V locking protocol also applies to (nonaggregate) join views of the form

$$JV = \pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)).$$

In fact, for join views, only V locks are necessary. The latch pool is no longer needed. This is due to the following reasons:

1. As discussed in Section 3.1, updates to materialized views 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.
2. The split group duplicate problem does not exist on join views.

We refer the reader to Section 4 for a formal complete correctness proof of this approach.

In a practical situation, if a join view contains a large number of duplicate tuples (e.g., due to projection), then the V locking protocol can yield higher performance than the traditional X locking protocol. This is because a join view with a large number of duplicate tuples behaves much like an aggregate join view with a few tuples, as duplicate tuples are hard to differentiate [11]. This effect is clearer from the correctness proof in Section 4.2.

4 Correctness of the V Locking Protocol

In this section, we prove the correctness of the V locking protocol. The intuition for this proof is that, if two transactions updating the base relations of a join view JV have no lock conflict with each other on the base relations of JV, they must generate different join result tuples. Additionally, the addition operation for the SUM and $COUNT$ aggregate operators is both associative and commutative.

We begin by reviewing our assumptions. 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. During aggregate join view maintenance, we put appropriate locks on all the base relations of the aggregate join view (i.e., X locks on the tuples in the base relations updated and S locks on the tuples in the other base relations mentioned in the view definition). We use strict two-phase locking. We assume that the locking mechanism used by the database system on the base relations ensures serializability in the absence of aggregate join views. Unless otherwise specified, all the locks are long-term locks that are held until transaction commits. Transactions updating the aggregate join view obtain V locks and latches in the latch pool as

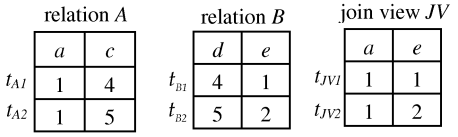


Fig. 7. Original status of base relation A, base relation B, and join view JV.

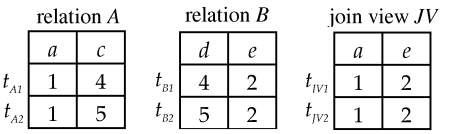


Fig. 8. Status of base relation A, base relation B, and join view JV—after updating tuple t_{B1}.

described in the V locking protocol. We make the same assumptions for nonaggregate join views.

We first prove serializability in Section 4.1 for the simple case where projection does not appear in the join view definition, while we consider projection in Section 4.2. In Section 4.3, we prove serializability for the case with aggregate join views

$$AJV = \gamma(\pi(\sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n))),$$

where γ is either $COUNT$ or SUM.

4.1 Proof for Join Views without Projection

To show that the V locking protocol guarantees serializability, we only need to prove that, for a join view $JV = \sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n)$, the following assertions hold (the strict two-phase locking protocol guarantees these four assertions for the base relations) [1], [8]:

1. Assertion 1. Transaction T's writes to join view JV are neither read nor written by other transactions until transaction T completes.
2. Assertion 2. Transaction T does not overwrite dirty data of other transactions in join view JV.
3. Assertion 3. Transaction T does not read dirty data from other transactions in join view JV.
4. Assertion 4. For any data in join view JV that is read by transaction T, other transactions do not write it before transaction T completes.

That is, we need to prove that no read-write, write-read, or write-write conflicts exist. In our proof, we assume that there are no duplicate tuples in the base relations. (At the cost of some additional complexity, the proof can be extended to handle the case where base relations contain duplicate tuples.)

The proof for the absence of read-write or write-read conflicts is trivial, as V and X locks are not compatible with S locks. In the following, we prove the absence of write-write conflicts. Consider the join result tuple $t_1 \bowtie \ldots \bowtie t_i \bowtie \ldots \bowtie t_n$ in the join view JV, where tuple $t_i \in R_i (1 \leq i \leq n)$. To update this join result tuple in the join view JV, transaction T has to update some tuple in some base relation. Suppose transaction T updates tuple t_i in base relation R_i for some $1 \leq i \leq n$. Then, transaction T needs to use an X lock to protect tuple $t_i \in R_i$. Also, for join view maintenance, transaction T needs to use S locks to protect all the other tuples $t_j \in R_j$ $(1 \leq j \leq n, j \neq i)$. Then, according to the two-phase locking protocol, before transaction T finishes execution, no other transaction can update any tuple $t_k \in R_k$ $(1 \leq k \leq n)$. That is, no other transaction can update the same join result tuple $t_1 \bowtie \ldots \bowtie t_i \bowtie \ldots \bowtie t_n$ in the join view JV until transaction T finishes execution. For a similar reason, transaction T does not overwrite dirty data of other transactions in the join view JV. □

4.2 Proof for Join Views with Duplicate-Preserving Projection

Now, we prove the correctness of the V locking protocol for the general case where

$$JV = \pi(\sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n)).$$

We assume that join view JV allows duplicate tuples. If no duplicate tuples are allowed in JV, we assume that each tuple in JV has a *dupcnt* attribute recording the number of copies of that tuple [11]; otherwise, JV cannot be incrementally maintained efficiently. For example, suppose we do not maintain the *dupcnt* attribute in JV. We delete a tuple from a base relation R_i $(1 \leq i \leq n)$ of JV and this tuple (when joined with other base relations) produces tuple t in JV. Then, we cannot decide whether we should delete tuple t from JV or not, as there may be other tuples in base relation R_i that (when joined with other base relations) also produce tuple t in JV. If we maintain the *dupcnt* attribute in the join view JV, then JV becomes an aggregate join view. The proof for the aggregate join view case is shown in Section 4.3 below. Hence, in the following, we only consider join views that allow duplicate tuples.

For a join view JV with projection, multiple tuples in JV may have the same value due to projection. In this case, the V locking protocol allows multiple transactions to update the same tuple in the join view JV concurrently. Hence, the proof in Section 4.1 no longer works.

We use an example to illustrate the point. Suppose the schema of base relation A is (a, c), the schema of base relation B is (d, e). The join view JV is defined as follows:

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

Suppose base relation A, base relation B, and the join view JV originally look as shown in Fig. 7.

Consider the following two transactions. Transaction T_1 updates tuple t_{B1} in base relation B from (4, 1) to (4, 2). To maintain the join view JV, we compute the old and new join result tuples (1, 4, 4, 1) and (1, 4, 4, 2). Then, we update tuple t_{JV1} in the join view JV from (1, 1) to (1, 2), as illustrated in Fig. 8.

Now, a second transaction T_2 updates tuple t_{B2} in base relation B from (5, 2) to (5, 3). To maintain the join view JV, we compute the old and new join result tuples (1, 5, 5, 2) and (1, 5, 5, 3). Then, we need to update one tuple in the join

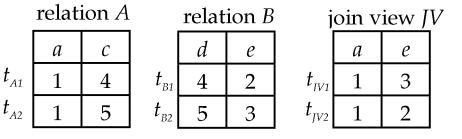


Fig. 9. Status of base relation A, base relation B, and join view JV—after updating tuple t_{B2}.

view JV from (1, 2) to (1, 3). Since all the tuples in the join view JV have value (1, 2) at present, it makes no difference which tuple we select to update. Suppose we select tuple t_{JV1} in the join view JV for update, as illustrated in Fig. 9.

Note that transactions T_1 and T_2 update the same tuple t_{JV1} in the join view JV. At this point, if we abort transaction T_1, we cannot change tuple t_{JV1} in the join view JV back to the value (1, 1), as the current value of tuple t_{JV1} is (1, 3) rather than (1, 2). However, we can pick up any other tuple (such as t_{JV2}) in the join view JV that has value (1, 2) and change its value back to (1, 1). That is, our V locking protocol requires logical undo (instead of physical undo) on the join view if the transaction holding the V lock aborts. During logical undo, no additional lock is needed. This is because V locks can "conceptually" be regarded as value locks. The value of the other tuple (such as t_{JV2}), (1, 2), has been locked before.

In the following, we give an "indirect" proof of the correctness of the V locking protocol using the serializability result in Section 4.1. Our intuition is that, although multiple tuples in the join view JV may have the same value due to projection, they originally come from different join result tuples before projection. Hence, we can show serializability by "going back" to the original join result tuples.

Consider an arbitrary database DB containing multiple base relations and join views. Suppose that there is another database DB' that is a "copy" of DB. The only difference between DB and DB' is that, for each join view with projection $JV = \pi(\sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n))$ in DB, we replace it by a join view without projection $JV' = \sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n)$ in DB'. Hence, $JV = \pi(JV')$. Each tuple t in the join view JV corresponds to one tuple t' in JV' (by projection).

Consider multiple transactions $T_1, T_2, \ldots$, and T_g. To prove serializability, we need to show that, in DB, any allowed concurrent execution of these transactions is equivalent to some serial execution of these transactions. Suppose that multiple transactions $T'_1, T'_2, \ldots$, and T'_g exist in DB'. Each transaction T'_j $(1 \leq j \leq g)$ is a "copy" of transaction T_j with the following differences:

1. Suppose, in DB, transaction T_j reads tuples Δ of JV. In DB', we let transaction T'_j read the tuples Δ' in JV' that correspond to Δ in JV.
2. Suppose, in DB, transaction T_j updates JV by Δ. According to the join view maintenance algorithm, transaction T_j needs to first compute the corresponding join result tuples Δ' that produce Δ, then integrate Δ' into JV. In DB', we let transaction T'_j update JV' by Δ'. That is, we always keep $JV = \pi(JV')$.

Hence, except for the projection on the join views:

1. For every j $(1 \leq j \leq g)$, transactions T'_j and T_j read and write the "same" tuples.
2. At any time, DB' is always a "copy" of DB.

For any allowed concurrent execution CE of transactions $T_1, T_2, \ldots$, and T_g in DB, we consider the corresponding (and also allowed) concurrent execution CE' of transactions $T'_1, T'_2, \ldots$, and T'_g in DB'. By the reasoning in Section 4.1, we know that, in DB', such concurrent execution CE' of transactions $T'_1, T'_2, \ldots$, and T'_g is equivalent to some serial execution of the same transactions. Suppose one such serial execution is transactions $T'_{k_1}, T'_{k_2}, \ldots$, and T'_{k_g}, where $\{k_1, k_2, \ldots, k_g\}$ is a permutation of $\{1, 2, \ldots, g\}$. Then, it is easy to see that, in DB, the concurrent execution CE of transactions $T_1, T_2, \ldots$, and T_g is equivalent to the serial execution of transactions $T_{k_1}, T_{k_2}, \ldots$, and T_{k_g}. □

4.3 Proof for Aggregate Join Views

We can also prove the correctness (serializability) of the V locking protocol for aggregate join views. Such a proof is similar to the proof in Section 4.2, so we only point out the differences between these two proofs and omit the details:

1. For any aggregate join view $AJV = \gamma(\pi(\sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n)))$ in DB, we replace it by a join view $JV' = \sigma(R_1 \bowtie \ldots \bowtie R_i \bowtie \ldots \bowtie R_n)$ in DB'. Each tuple in the aggregate join view AJV corresponds to one or multiple tuples in JV' (by projection and aggregation). At any time, we always keep $AJV = \gamma(\pi(JV'))$, utilizing the fact that the addition operation for the SUM and $COUNT$ aggregate operators is both associative and commutative.
2. In the presence of updates that cause the insertion or deletion of tuples in the aggregate join view, the latches in the latch pool guarantee that the "race" conditions that can cause the split group duplicate problem cannot occur. For each aggregate group, at any time at most one tuple corresponding to this group exists in the aggregate join view AJV. □

5 Performance of the V Locking Protocol

In this section, we investigate the performance of the V locking protocol through a simulation study in IBM's DB2 Version 7.2. We focus on the throughput of a targeted class of transactions (i.e., transactions that update a base relation of an aggregate join view). 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 node, so there were four data server nodes on the 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. Table 3 shows the sizes of the *lineitem* and *partsupp* relations. 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 *suppkeys*,

TABLE 3
Test Data Set

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

each corresponding to the same number of tuples in the *partsupp* relation.

We used the following kind of transaction for 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.

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

1. We used the default setting of DB2, where the buffer pool size is 250 pages on each data server node. (We also tested larger buffer pool sizes. The results were similar and, thus, are omitted.)
2. We ran x Ts. Each of these x Ts has a different *orderkey* value. x is an arbitrarily large number. Its specific value does not matter, as we only focused on throughput.
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 a concurrency 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. (We also performed a number of aggregate groups test that varies R. The results of this test did not provide more insight, so we omit them here.)
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*. In an actual implementation of the V locking protocol, we would encounter the following issues:
   a. Conflicts of short-term X page latches and conflicts of the latches in the latch pool 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*.

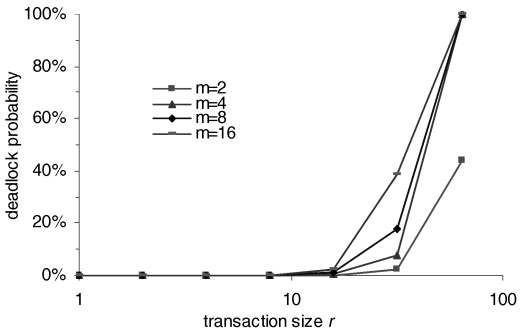


Fig. 10. Predicted deadlock probability of the X lock method (concurrency test).

As a result, our performance numbers are not exact performance predictions, which will depend upon the actual implementation details of the V locking protocol. Rather, our experiments are intended to illustrate trends of when the V lock method tends to do better than the X lock method.

5.2 Concurrency Test Results

We discuss the deadlock probability and throughput testing results from the concurrency test in Sections 5.2.1 and 5.2.2, respectively.

5.2.1 Deadlock Probability

As mentioned in the introduction, for the X lock method, we can use the formula $(m-1)(r-1)^4/(4R^2)$ to roughly estimate the probability that any particular transaction deadlocks. We show the deadlock probability of the X lock method computed by the formula in Fig. 10. (Note: All figures in Sections 5.2.1 and 5.2.2 use logarithmic scale for the x-axis.)

For the X lock method, the deadlock probability increases linearly with both m and the fourth power of r. When both m and r are small, this deadlock probability is small. However, when either m or r becomes large, this deadlock probability approaches 1 quickly. For example, consider the case with $m = 16$. When $r = 16$, this deadlock probability is only 2 percent. However, when $r = 32$, this deadlock probability becomes 38 percent. The larger r, the smaller m is needed to make this deadlock probability become close to 1.

We show the deadlock probability of the X lock method measured in our tests in Fig. 11. Figs. 10 and 11 roughly match. This indicates that our formula gives a fairly good

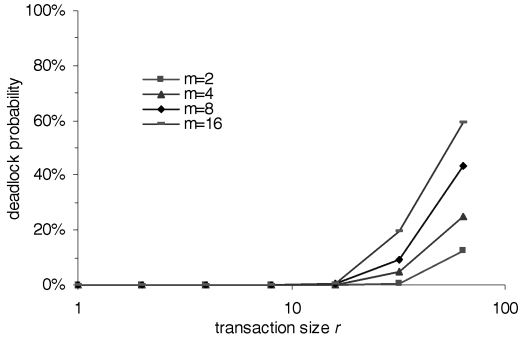


Fig. 11. Measured deadlock probability of the X lock method (concurrency test).

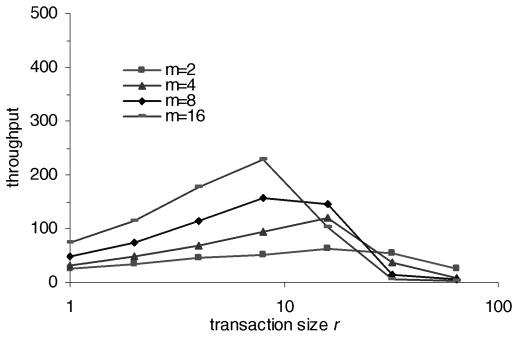


Fig. 12. Throughput of the X lock method (concurrency test).

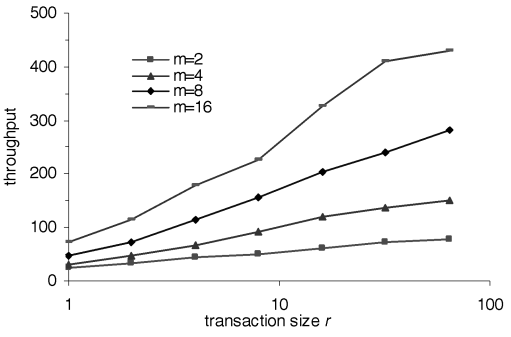


Fig. 13. Throughput of the V lock method (concurrency test).

rough estimate of the deadlock probability of the X lock method.

To see how deadlocks influence performance, we investigated the relationship between the throughput and the deadlock probability. By definition, 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.

Hence, once the system starts to deadlock, the deadlock problem tends to become worse and worse. Eventually, the throughput of the X lock method deteriorates significantly.

5.2.2 Throughput

We show the throughput of the X lock method in Fig. 12. (The throughput numbers in Figs. 12 and 13 are scaled by the same constant factor.) For a given m, when r is small, the throughput of the X lock method keeps increasing with r. This is because executing a large transaction is much more efficient than executing a large number of small transactions. When r becomes large enough (e.g., $r = 32$), the X lock method causes a large number of deadlocks. That is, the X lock method runs into a severe deadlock problem. The larger m, the smaller r is needed for the X lock method to run into the deadlock problem. Once the deadlock problem occurs, the throughput of the X lock method deteriorates significantly. Actually, it decreases as r increases. This is because the larger r, the more transactions are aborted and re-executed due to deadlock.

For a given r, before the deadlock problem occurs, the throughput of the X lock method increases with m. This is because the larger m is, the higher concurrency in the RDBMS. However, when r is large enough (e.g., $r = 32$) and the X lock method runs into the deadlock problem, due to the extreme overhead of repeated transaction abortion and re-execution, the throughput of the X lock method decreases as m increases.

We show the throughput of the V lock method in Fig. 13. The general trend of the throughput of the V lock method is similar to that of the X lock method (before the deadlock problem occurs). That is, the throughput of the V lock method increases with both m and r. However, the V lock method never deadlocks. For a given m, the throughput of the V lock method keeps increasing with r (until all system resources become fully utilized). Once the X lock method runs into the deadlock problem, the V lock method exhibits great performance advantages over the X lock method.

We show the ratio of the throughput of the V lock method to that of the X lock method in Fig. 14. (Note that Fig. 14 uses logarithmic scale for both the x-axis and the y-axis.) 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. This is because the larger m or r, the easier the transactions deadlock in the X lock method. The extreme overhead of repeated transaction abortion and re-execution exceeds the benefit of the higher concurrency (efficiency) brought by a larger m (r).

Note:

1. Our tests do not address the performance impact of N—the number of X latches in the latch pool. In general, we cannot control the number of aggregate groups in an aggregate join view, which is usually small due to aggregation. However, we can control the number N, which can be relatively large since each latch only occupies a few bytes [8]. As mentioned in Section 2.2.1, we would expect the performance

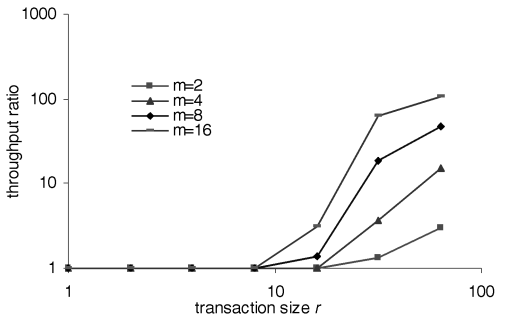


Fig. 14. Throughput improvement gained by the V lock method (concurrency test).

impact of N to be relatively small compared to the performance impact of lock conflicts.

2. In general, locks are long-term and not released until transaction commit time; latches are short-term and will be released quickly [8]. Hence, in the case that all updates are concentrated on a single tuple in the aggregate join view, we would still expect the V lock method to perform better than the X lock method. However, without an actual implementation of the V locking protocol, it is difficult to measure precisely the benefit of the V lock method over the X lock method in this case.

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 latch pool 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 and materialized nonaggregate join view maintenance.

An extended version of this paper that contains the B-tree index locking protocol is available at http://www.cs.wisc.edu/~gangluo/latch_final_full.pdf.

Acknowledgment

The authors would like to thank C. Mohan, Henry F. Korth, David B. Lomet, and the anonymous reviewers for useful discussions. This work was supported by the NCR Corporation and also by US National Science Foundation grants CDA-9623632 and ITR 0086002.

References

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

[2] B.R. Badrinath and K. Ramamritham, "Semantics-Based Concurrency Control: Beyond Commutativity," *ACM Trans. Database Systems,* vol. 17, no. 1, pp. 163-199, 1992.

[3] D. Gawlick and D. Kinkade, "Varieties of Concurrency Control in IMS/VS Fast Path," *IEEE Database Eng. Bull.,* vol. 8, no. 2, pp. 3-10, 1985.

[4] J. Gehrke, F. Korn, and D. Srivastava, "On Computing Correlated Aggregates over Continual Data Streams," *Proc. SIGMOD,* pp. 13-24, 2001.

[5] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total," *Proc. Int'l Conf. Data Eng.,* pp. 152-159, 1996.

[6] J. Gray, R.A. Lorie, G.R. Putzolu, and I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," *Proc. IFIP Working Conf. Modeling in Data Base Management Systems,* pp. 365-394, 1976.

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

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

[9] A. Kawaguchi, D.F. Lieuwen, I.S. Mumick, D. Quass, and K.A. Ross, "Concurrency Control Theory for Deferred Materialized Views," *Proc. Int'l Conf. Database Theory,* pp. 306-320, 1997.

[10] H.F. Korth, "Locking Primitives in a Database System," *J. ACM,* vol. 30, no. 1, pp. 55-79, 1983.

[11] W. Labio, J. Yang, Y. Cui, H. Garcia-Molina, and J. Widom, "Performance Issues in Incremental Warehouse Maintenance," *Proc. Very Large Data Bases Conf.,* pp. 461-472, 2000.

[12] G. Luo, J.F. Naughton, C.J. Ellmann, and M.W. Watzke, "Locking Protocols for Materialized Aggregate Join Views," *Proc. Very Large Data Bases Conf.,* pp. 596-607, 2003.

[13] P.E. O'Neil, "The Escrow Transactional Method," *ACM Trans. Database Systems,* vol. 11, no. 4, pp. 405-430, 1986.

[14] M. Poess and C. Floyd, "New TPC Benchmarks for Decision Support and Web Commerce," *SIGMOD Record,* vol. 29, no. 4, pp. 64-71, 2000.

[15] R.F. Resende, D. Agrawal, and A.E. Abbadi, "Semantic Locking in Object-Oriented Database Systems," *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications,* pp. 388-402, 1994.

[16] A. Reuter, "Concurrency on High-Traffic Data Elements," *Proc. ACM Symp. Principles of Database Systems,* pp. 83-92, 1982.

[17] A. Silberschatz and Z.M. Kedem, "Consistency in Hierarchical Database Systems," *J. ACM,* vol. 27, no. 1, pp. 72-80, 1980.

Gang Luo received the BSc degree from Shanghai Jiaotong University, People's Republic of China, in 1994, and the PhD degree from the University of Wisconsin-Madison in 2004. He is currently a research staff member at the IBM T.J. Watson Research Center. He has broad interests in various parts of relational database systems. Recently, he has been working on using HCI techniques to improve both the user-friendliness and the performance of relational database systems.

Jeffrey F. Naughton received the bachelor's degree in mathematics from the University of Wisconsin-Madison and the PhD degree in computer science from Stanford University. He served as a faculty member in the Computer Science Department at Princeton University before moving to the University of Wisconsin-Madison, where he is currently a professor of computer science. His research has focused on improving the performance and functionality of database management systems. He has published more than 100 technical papers and received the US National Science Foundation's Presidential Young Investigator award in 1991 and was named an ACM fellow in 2002.

Curt J. Ellmann received the master's degree in computer science from the University of Wisconsin-Madison. He works for the Division of Information Technology at the University of Wisconsin-Madison. His current focus is on "open source" software projects that the university is involved with. Prior to that, he was a manager for the optimizer group for the Teradata Database system, and the site manager for the Teradata software development lab in Madison, Wisconsin.

Michael W. Watzke received the bachelor's degree in electrical engineering and computer science from the University of Wisconsin-Madison. He has worked in the database software and information technology consulting fields for 18 years. He is currently working for NCR/Teradata as a database architect focusing on the areas of application integration and performance.