V Locking Protocol for Materialized Aggregate Join Views on B-tree Indices

Gang Luo
IBM T.J. Watson Research Center
luog@us.ibm.com

Abstract. Immediate materialized view maintenance with transactional consistency is highly desirable to support real-time decision making. Nevertheless, due to high deadlock rates, such maintenance can cause significant performance degradation in the database system. To increase concurrency during such maintenance, we previously proposed the V locking protocol for materialized aggregate join views and showed how to implement it on hash indices. In this paper, we address the thorny problem of implementing the V locking protocol on B-tree indices. We also formally prove that our techniques are both necessary and sufficient to ensure correctness (serializability).

1. Introduction

Materialized views are widely used in database systems and Web-based information systems to improve query performance [4]. As real-time decision making is increasingly being needed by enterprises [14], the requirement of immediate materialized view maintenance with transactional consistency is becoming more and more necessary and important for providing consistent and up-to-date query results. Reflecting real world application demands, this requirement has become mandatory in the TPC-R benchmark [13]. Graefe and Zwilling [6] also argued that materialized views are like indexes. Since indexes are always maintained immediately with transactional consistency, materialized views should fulfill the same requirement. A few detailed examples for this requirement on materialized views are provided in [6].

In a materialized aggregate join view *AJV*, multiple tuples are aggregated into one group if they have identical group by attribute values. If generic concurrency control mechanisms are used, immediate maintenance of *AJV* with transactional consistency can cause significant performance degradation in the database system. Since different tuples in a base relation of *AJV* can affect the same aggregated tuple in *AJV*, the addition of *AJV* can introduce many lock conflicts and/or deadlocks that do not arise in the absence of *AJV*. The smaller *AJV* is, the more lock conflicts and/or deadlocks will occur. In practice, this deadlock rate can easily become 50% or higher [9]. A detailed deadlock example is provided in [9].

To address this lock conflict/deadlock issue, we previously proposed the V+W locking protocol [9] for materialized aggregate join views. The key insight is that the *COUNT* and *SUM* aggregate operators are associative and commutative [7]. Hence, during maintenance of the materialized aggregate join view, we can use V locks rather than traditional X locks. V locks do not conflict with each other and can increase concurrency, while short-term W locks are used to prevent "split group duplicates" — multiple tuples in the aggregate join view for the same group, as shown in Section 2.2 below. [9] described how to implement the V+W locking protocol on both hash indices and B-tree indices.

It turns out that the W lock solution for the split group duplicate problem can be replaced by a latch (i.e., semaphore) pool solution, which is more efficient because acquiring a latch is much cheaper than acquiring a lock [5]. This leads to the V locking protocol for materialized aggregate join views presented in [10]. There, V locks are augmented with a "value-based" latch pool. Traditionally, latches are used to protect the physical integrity of certain data structures (e.g., the data structures in a page [5]). In our case of materialized view maintenance, no physical data structure would be corrupted if the latch pool were not used. The "value-based" latch pool obtains its name because it is used to protect the logical integrity of aggregate operations rather than the physical integrity of the database. [10] showed how to implement the V locking protocol on hash indices and formally proved the correctness of the implementation method. [10] also performed a simulation study in a commercial RDBMS, demonstrating that the performance of the V locking protocol can be two orders of magnitude higher than that of the traditional X locking protocol.

This paper improves upon our previous work by addressing the particularly thorny problem of implementing the V locking protocol on B-tree indices. Typically, implementing high concurrency locking modes poses special challenges when B-trees are considered, and the V locking protocol is no exception. We make three contributions. First, we present the method of implementing the V locking protocol on B-tree indices. Second, we show that our techniques are all necessary. Third, we formally prove the correctness (serializability) of our implementation method.

In related work, Graefe and Zwilling [6] independently proposed a multi-version concurrency control protocol for materialized aggregate join views. It uses hierarchical escrow locking, snapshot transactions, key-range locking, and system transactions. The key insight of that multi-version concurrency control protocol is similar to that of our V locking protocol. Nevertheless, that multi-version concurrency control protocol cannot avoid split group duplicates in materialized aggregate join views. Instead, special operations have to be performed during materialized aggregate join view query time to address the split group duplicate problem. [6] did not formally show that its proposed techniques are all necessary. [6] also gave no rigorous proof of the correctness of its proposed concurrency control protocol.

Our focus in this paper 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 [3]), we restrict our attention to the three incrementally updateable aggregate operators: *COUNT*, *SUM*, and *AVG*. In practice, *AVG* is computed using *COUNT* and *SUM*, as *AVG=SUM/COUNT* (*COUNT* and *SUM* are distributive while *AVG* is algebraic [2]). In the rest of the paper, we only discuss *COUNT* and *SUM*, whereas our locking techniques for *COUNT* and *SUM* also apply to *AVG*. Moreover, by letting $n=1$ in the definition of *AJV*, we include aggregate views over single relations.

2. V Locks, Latches, and B-Trees

In this section, we present our method of implementing the V locking protocol on B-tree indices. On B-tree indices, we use value locks to refer to key-range locks. To be consistent with the approach advocated by Mohan [11], 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).

2.1 The V Locking Protocol

In the V locking protocol for materialized aggregate join views [10], 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. 1.

Table 1. Compatibilities among the elementary locks.

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

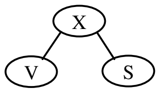

Fig. 1. The lock conversion lattice of the elementary locks.

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.

2.2 Split Groups and B-Trees

We consider 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′*. *T* integrates a new join result tuple (1, 2, 3) into *AJV* (by insertion into some base relation *R*). *T′* integrates another new join result tuple (1, 2, 4) into *AJV* (by insertion into *R*). Using standard concurrency control without V locks, to integrate a join result tuple t_1 into *AJV*, a transaction will execute something like the following operations:

(1) Obtain an X value lock for $t_1.a$ on I_B. 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 I_B.
(3) For each row id in the row id list, fetch the corresponding tuple t_2. Check whether $t_2.a=t_1.a$ 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 t_1 into 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 t_1. Also, insert the row id of this new tuple into 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) *T* obtains 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) *T´* obtains a V value lock for $a=1$ on 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) *T* inserts a new tuple t_1=(1, 2, 3) into *AJV*, and inserts the row id of t_1 into the row id list in the entry for $a=1$ of I_B.
(4) *T´* inserts a new tuple t_3=(1, 2, 4) into *AJV*, and inserts the row id of t_3 into the row id list in the entry for $a=1$ of 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.

2.3 The Latch Pool

To enable the use of V locks while avoiding split group duplicates, we use the same latch pool for aggregate join views as that described in [10]. 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 pre-allocate 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. First, during the period that a transaction holds a latch in the latch pool, this transaction does not request another latch in the latch pool. Second, 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. Third, during the period that a transaction holds a latch in the latch pool, this transaction does not request any lock. The first two properties guarantee that there are no deadlocks between latches. The third property guarantees that there are no deadlocks between latches and locks. These properties are necessary, because 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 [5] and *N* is substantially larger than the number of concurrently running transactions in the RDBMS, the probability of having false latch conflicts should be small.

2.4 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 [11] was perhaps the first published description of the issues that arise and their solution. Unfortunately, we cannot directly use the techniques in Mohan [11] to implement V locks as value (key-range) locks. Otherwise as shown in [9], serializability can be violated.

2.4.1 Operations of Interest

To implement V value locks on B-tree indices correctly, we need to combine those techniques in Mohan *et al*. [11, 5] with the technique of logical deletion of keys [12, 8]. 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 first kind V value lock on key value v_1.**
(5) **Put a second kind V value lock on key value v_1.**

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 second kind V value lock on the B-tree index. Immediately before we start the tuple integration, we request a latch on the group by attribute value of *t*. After integrating *t* into *AJV*, we release the latch on the group by attribute value of *t*.
(2) To remove a join result tuple from *AJV* (e.g., due to deletion from some base relation of *AJV*), we put a first kind V value lock on the B-tree index.

Unlike Mohan *et al*. [11, 5], 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 *t* into *AJV* are as follows:

```
If the aggregate group of t exists in AJV
    Update the aggregate group in AJV;
Else Insert a new aggregate group into AJV for 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 second kind V value lock on *t.a* before we can integrate *t* into *AJV*. Similarly, suppose we delete a tuple from some base relation of *AJV*. We compute the corresponding join result tuples. For each such join result tuple *t*, we execute the following steps to remove *t* from *AJV*:

```
Find the aggregate group of 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 first kind V value lock on *t.a* before we can remove *t* from *AJV*.

The ARIES/KVL method described in Mohan [11] for implementing value locks on a B-tree index requires the insertion/deletion operation to be done immediately after a transaction obtains appropriate locks. Also, in ARIES/KVL, the value lock implementation method is closely tied to the B-tree implementation method, because ARIES/KVL strives to take advantage of both IX locks and instant locks to increase concurrency. In the V 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 obtains 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.

2.4.2 Operation 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 I_B. If such an entry exists, we put an S lock for v_1 on I_B. If no such entry exists, we find the smallest value v_2 in I_B such that $v_2>v_1$. Then we put an S lock for v_2 on I_B.

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

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

Op4. Put a first kind V value lock on key value v_1: We put a V lock for value v_1 on I_B.

Op5. Put a second kind V value lock on key value v_1: We first put a V lock for value v_1 on I_B. Then we check whether some entry for v_1 exists in I_B. If no entry for v_1 exists, we do the following:

(a) We find the smallest value v_2 in I_B such that $v_2>v_1$. Then we put a short-term V lock for v_2 on I_B. If the V lock for v_2 on I_B is acquired as an X lock, we upgrade the V lock for v_1 on I_B to an X lock. This situation may occur when transaction *T* already holds an S or X lock for v_2 on I_B.

(b) We request a latch on v_2. We insert into I_B an entry for v_1 with an empty row id list. (Note that at a later point *T* will insert a row id into this row id list after *T* inserts the corresponding tuple into the aggregate join view.) Then we release the latch on v_2.

(c) We release the short-term V lock for value v_2 on I_B.

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
first kind V value lock		V	
second kind V value lock	v_1 exists	V	
	v_1 does not exist and the V lock on v_2 is acquired as a V lock	V	V
	v_1 does not exist and the V lock on v_2 is acquired as an X lock	X	X

During the period that a transaction *T* holds a first kind V (or second kind V, or X) value lock for value v_1 on the B-tree index I_B, if *T* wants to delete the entry for value v_1, *T* needs to do a logical deletion of keys [12, 8] instead of a physical deletion. That is, instead of removing the entry for v_1 from I_B, it is left there with a *delete_flag* set to 1. If the delete is rolled back, the *delete_flag* is reset to 0. If another transaction inserts an entry for v_1 into I_B before the entry for v_1 is garbage collected, the *delete_flag* of the entry for v_1 is reset to 0. This is to avoid potential write-read conflicts discussed at the beginning of Section 2.4. The physical deletion operations are necessary, otherwise I_B may grow unbounded. To leverage the overhead of the physical deletion operations, we perform them as garbage collection by other operations (of other transactions) that happen to pass through the affected nodes in I_B [8].

In Op4 (put a first kind V value lock on key value v_1), usually an entry for value v_1 exists in the B-tree index. However, the situation that no entry for v_1 exists in the B-tree index is still possible. 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 *AJV*. If we insert a new tuple *t* into *R* and generate several new join result tuples, we need to acquire the appropriate second kind V value locks on I_B before we can integrate these new join result tuples into *AJV*. If we delete a tuple *t* from *R*, to maintain *AJV*, normally we need to first compute the corresponding join result tuples that are to be removed from *AJV*. These join result tuples must have been integrated into *AJV* before. Hence, when we acquire the first kind V value locks for their *d* attribute values, these *d* attribute values must exist in 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 [10] for details). There is only one tuple *t* in *R* whose attribute d=v, but no matching tuple in the other base relations of *AJV* that can be joined with *t*. Hence, there is no tuple in *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 *AJV*, there is no need for *T* to compute the corresponding join result tuples that are to be removed from *AJV*. *T* can execute the following "direct propagate" update operation:

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

Then when *T* requests a first kind V value lock for d=v on the B-tree index I_B, *T* will find that no entry for value *v* exists in I_B.

In Op4 (put a first kind V value lock on key value v_1), even if no entry for value v_1 exists in the B-tree index I_B, we still only need to put a V lock for v_1 on I_B. There is no need to put any lock for value v_2 on I_B. That is, no next-key locking is necessary in this case. This is because the first kind V value lock can only be used to remove a join result tuple from the aggregate join view *AJV*. In the case that no entry for v_1 currently exists in I_B, usually no join result tuple for v_1 can be removed from *AJV* (unless another transaction inserts an entry for v_1 into I_B), because no join result tuple currently exists for v_1. Then the first kind V value lock on key value v_1 is used to protect a null operation. Therefore, no next-key locking is necessary. Note: it is possible that after transaction *T* obtains the first kind V value lock for v_1 on I_B, another transaction inserts an entry for v_1 into I_B. Hence, we cannot omit the V lock for v_1 on I_B. This effect is clearer from the correctness proof in Section 4.

3. Necessity of Our Techniques

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 — we use the following aggregate join view *AJV* to illustrate the rationale for the techniques introduced in Section 2.4. The schema of *AJV* is (*a*, *sum*(*b*)). Suppose a B-tree index I_B is built on attribute *a* of *AJV*. We show that if any of the techniques from the previous section are omitted (and not replaced by other equivalent techniques), then we cannot guarantee serializability.

Technique 1. As mentioned above in Op5 (put a second kind V value lock on key value v_1), we need to request a latch on value v_2. The following example illustrates why. Suppose originally the aggregate join view *AJV* contains two tuples that correspond to *a=1* and *a=4*. Consider three transactions *T*, *T′*, and *T″* on *AJV*. *T* integrates a new join result tuple (3, 5) into *AJV*. *T′* integrates a new join result tuple (2, 6) into *AJV*. *T″* reads those tuples whose attribute *a* is between 1 and 3. Suppose no latch on v_2 is requested. Also, suppose *T*, *T′*, and *T″* are executed as follows:

(1) *T* puts a V lock for *a=3* and another V lock for *a=4* on *AJV*.

	1			4
T			V	V

(2) *T′* finds the entries for *a=1* and *a=4* in the B-tree index. *T′* puts a V lock for *a=2* and another V lock for *a=4* on *AJV*.

	1			4
T			V	V
T′		V		V

(3) *T* inserts the tuple (3, 5) and an entry for *a=3* into *AJV* and the B-tree index I_B, respectively.

	1		3	4
T			V	V
T′		V		V

(4) *T* commits and releases the V lock for *a=3* and the V lock for *a=4*.

	1		3	4
T′		V		V

(5) Before *T′* inserts the entry for *a=2* into I_B, *T″* finds the entries for *a=1*, *a=3*, and *a=4* in the B-tree index. *T″* puts an S lock for *a=1* and another S lock for *a=3* on *AJV*.

	1		3	4
T′		V		V
T″	S		S	

In this way, *T″* can start execution even before *T′* finishes execution. This is incorrect due to the write-read conflict between *T′* and *T″* (on the tuple whose attribute *a=2*).

Technique 2. As mentioned above in Op5 (put a second kind V value lock on key value v_1), if the V lock for value v_2 on the B-tree index I_B is acquired as an X lock, we need to upgrade the V lock for value v_1 on I_B to an X lock. The following example illustrates why. Suppose originally the aggregate join view *AJV* contains only one tuple that corresponds to *a=4*. Consider two transactions *T* and *T′* on *AJV*. *T* first reads those tuples whose attribute *a* is between 1 and 4, then integrates a new join result tuple (3, 6) into *AJV*. *T′* integrates a new join result tuple (2, 5) into *AJV*. Suppose the V lock for v_1 on I_B is not upgraded to an X lock. Also, suppose *T* and *T′* are executed as follows:

(1) *T* finds the entry for *a=4* in the B-tree index I_B. *T* puts an S lock for *a=4* on *AJV*. *T* reads the tuple in *AJV* whose attribute *a=4*.

			4
T			S

(2) *T* puts a V lock for *a=3* and another V lock for *a=4* on *AJV*. Note the V lock for *a=4* is acquired as an X lock because *T* has already put an S lock for *a=4* on *AJV*.

			4
T		V	X

(3) *T* inserts the tuple (3, 6) and an entry for *a=3* into *AJV* and I_B, respectively. Then *T* releases the V lock for *a=4* on *AJV*. Note *T* still holds an X lock for *a=4* on *AJV*.

		3	4
T		V	X

(4) Before *T* finishes execution, *T′* finds the entries for *a=3* and *a=4* in I_B. *T′* puts a V lock for *a*=2 and another V lock for *a=3* on *AJV*.

		3	4
T		V	X
T′	V	V	

In this way, *T′* can start execution even before *T* finishes execution. This is incorrect due to the read-write conflict between *T* and *T′* (on the tuple whose attribute *a=2*).

Technique 3. As mentioned above in Op5 (put a second kind V value lock on key value v_1), if no entry for value v_1 exists in the B-tree index I_B, we need to insert an entry for v_1 into I_B. The following example illustrates why. Suppose originally the aggregate join view *AJV* contains two tuples that correspond to *a=1* and *a=5*. Consider three transactions *T*, *T′*, and *T″* on *AJV*. *T* integrates two new join result tuples (4, 5) and (2, 6) into *AJV*. *T′* integrates a new join result tuple (3, 7) into *AJV*. *T″* reads those tuples whose attribute *a* is between 1 and 3. Suppose we do not insert an entry for v_1 into I_B. Also, suppose *T*, *T′*, and *T″* are executed as follows:

(1) *T* finds the entries for *a=1* and *a=5* in the B-tree index I_B. For the new join result tuple (4, 5), *T* puts a V lock for *a=4* and another V lock for *a=5* on *AJV*.

	1				5
T				V	V

(2) *T* finds the entries for *a=1* and *a=5* in I_B. For the new join result tuple (2, 6), *T* puts a V lock for *a=2* and another V lock for *a=5* on *AJV*.

	1				5
T		V		V	V

(3) *T* inserts the tuple (4, 6) and an entry for *a=4* into *AJV* and I_B, respectively.

	1			4	5
T		V		V	V

(4) *T′* finds the entries for *a=1*, *a=4*, and *a=5* in I_B. *T′* puts a V lock for *a=3* and another V lock for *a=4* on *AJV*.

	1			4	5
T		V		V	V
T′			V	V	

(5) *T′* inserts the tuple (3, 7) and an entry for *a=3* into *AJV* and I_B, respectively.

	1		3	4	5
T		V		V	V
T′			V	V	

(6) *T′* commits and releases the two V locks for *a=3* and *a=4*.

	1		3	4	5
T		V		V	V

(7) Before *T* inserts the entry for *a=2* into I_B, *T″* finds the entries for *a=1*, *a=3*, *a=4*, and *a=5* in I_B. *T″* puts an S lock for *a=1* and another S lock for *a=3* on *AJV*.

	1		3	4	5
T		V		V	V
T″	S		S		

In this way, *T″* can start execution even before *T* finishes execution. This is incorrect due to the write-read conflict between *T* and *T″* (on the tuple whose attribute *a=2*).

4. Correctness of the Key-range Locking Protocol

In this section, we prove the correctness (serializability) of our key-range locking strategy for aggregate join views on B-tree indices. 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_1 (irrespective of whether an entry for v_1 exists in I_B, i.e., the phantom problem [5] 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 *AJV* whose d=v_1 [1, 5]. As shown in Korth [7], write-write conflicts are avoided by the associative and commutative properties of the addition operation. Furthermore, the use of the latches in the latch pool guarantees that for each aggregate group, at any time at most one tuple corresponding to this group exists in *AJV*. We enumerate all the possible cases to show that write-read and read-write conflicts do not exist. Since we use next-key locking, in the enumeration, we only need to focus on v_1 and the smallest existing value v_2 in I_B such that $v_2>v_1$.

Consider the following two transactions T and T'. T updates (some of) the tuples in the aggregate join view *AJV* whose attribute d has value v_1. T' reads the tuples in *AJV* whose attribute d has value v_1 (e.g., through a range query). Suppose v_2 is the smallest existing value in the B-tree index I_B such that $v_2>v_1$. T needs to obtain a first kind V (or second kind V, or X) value lock for $d=v_1$ on I_B. T' needs to obtain an S value lock for $d=v_1$ on I_B. There are four possible cases:

(1) Case 1: An entry E for value v_1 already exists in the B-tree index I_B. Also, transaction T' obtains the S value lock for $d=v_1$ on I_B first.
To put an S value lock for $d=v_1$ on I_B, T' needs to put an S lock for $d=v_1$ on *AJV*. During the period that T' holds the S lock for $d=v_1$ on *AJV*, the entry E for value v_1 always exists in I_B. Then during this period, transaction T cannot obtain the V (or V, or X) lock for $d=v_1$ on *AJV*. That is, T cannot obtain the first kind V (or second kind V, or X) value lock for $d=v_1$ on I_B.

(2) Case 2: An entry E for value v_1 already exists in the B-tree index I_B. Also, transaction T obtains a first kind V (or second kind V, or X) value lock for $d=v_1$ on I_B first.
To put a first kind V (or second kind V, or X) value lock for $d=v_1$ on I_B, T needs to put a V (or V, or X) lock for $d=v_1$ on *AJV*. During the period that T holds the V (or V, or X) lock for $d=v_1$ on *AJV*, the entry E for v_1 always exists in I_B. Note during this period, if some transaction deletes E from I_B, E is only logically deleted. Only after T releases the V (or V, or X) lock for $d=v_1$ on *AJV* may E be physically deleted from I_B. Hence, during the period that T holds the V (or V, or X) lock for $d=v_1$ on *AJV*, transaction T' cannot obtain the S lock for $d=v_1$ on *AJV*. That is, T' cannot obtain the S value lock for $d=v_1$ on I_B.

(3) Case 3: No entry for value v_1 exists in the B-tree index I_B. Also, transaction T' obtains the S value lock for $d=v_1$ on I_B first.
To put an S value lock for $d=v_1$ on I_B, T' needs to put an S lock for $d=v_2$ on *AJV*. During the period that T' holds the S lock for $d=v_2$ on *AJV*, no other transaction T'' can insert an entry for value v_3 into I_B such that $v_1 \le v_3 < v_2$, because this would require T'' to obtain a V (or X) lock for $d=v_2$ on *AJV*. Then during the period that T' holds the S lock for $d=v_2$ on *AJV*, transaction T cannot obtain the second kind V (or X) value lock for $d=v_1$ on I_B, because this would require T to obtain a V (or X) lock for $d=v_2$ on *AJV*. Note during the period that T' holds the S lock for $d=v_2$ on *AJV*, T can put a V lock for $d=v_1$ on *AJV* and then obtain the first kind V value lock for $d=v_1$ on I_B. However, during this period, T cannot use the first kind V value lock for $d=v_1$ on I_B to do any update. This is because no entry for v_1 exists in I_B, and T cannot use the first kind V value lock for $d=v_1$ to insert an entry for v_1 into I_B. Hence, there is no read-write conflict between T and T' on $d=v_1$. Also, if T' itself inserts an entry for value v_3 into I_B such that $v_1 \le v_3 < v_2$, T' will hold an X lock for $d=v_3$ on *AJV* (see how the second kind V and X value locks are implemented on the B-tree index in Section 2.4). Then T still cannot obtain the second kind V (or X) value lock for $d=v_1$ on I_B before T' finishes execution. (If

$v_1=v_3$, then T cannot obtain the first kind V value lock for $d=v_1$ on I_B before T' finishes execution. If $v_1<v_3$, T cannot use the first kind V value lock for $d=v_1$ on I_B to do any update.)

(4) Case 4: No entry for value v_1 exists in the B-tree index I_B. Also, transaction T obtains the first kind V (or second kind V, or X) value lock for $d=v_1$ on I_B first. In this case, there are three possible scenarios:

(a) T obtains the first kind V value lock for $d=v_1$ on I_B first. Hence, T puts a V lock for $d=v_1$ on *AJV*. During the period that T holds the V lock for $d=v_1$ on *AJV*, another transaction T'' can insert an entry for v_1 into I_B. $T''\neq T$, as T cannot use a first kind V value lock for $d=v_1$ to insert an entry for v_1 into I_B. Before T'' inserts an entry for v_1 into I_B, no such entry exists in I_B. Hence, T cannot use the first kind V value lock for $d=v_1$ to do update and no write-read conflict exists between T and T' on $d=v_1$. After T'' inserts an entry for v_1 into I_B, the entry for v_1 cannot be physically deleted from I_B before T releases the V lock for $d=v_1$ on *AJV*. Hence, during this period, T' cannot obtain the S value lock for $d=v_1$ on I_B, as T' cannot put an S lock for $d=v_1$ on *AJV*.

(b) T obtains the second kind V value lock for $d=v_1$ on I_B first. Hence, T puts a V lock for $d=v_1$ and another V lock for $d=v_2$ on *AJV*. Also, T inserts a new entry for v_1 into I_B. Before T inserts the new entry for v_1 into I_B, T holds the V lock and the latch for $d=v_2$ on *AJV*. During that period, no other transaction T'' can insert an entry for value v_3 into I_B such that $v_1\leq v_3<v_2$, because this would require T'' to obtain a latch (or X lock) for $d=v_2$ on *AJV*. Then during that period, T' cannot obtain the S value lock for $d=v_1$ on I_B, because this would require T' to obtain an S lock for $d=v_2$ on *AJV*. After T inserts the new entry for v_1 into I_B, T will hold a V lock for $d=v_1$ on *AJV* until T finishes execution. Then during this period, T' still cannot obtain the S value lock for $d=v_1$ on I_B, as this would require T' to obtain an S lock for $d=v_1$ on *AJV*.

(c) T obtains the X value lock for $d=v_1$ on I_B first. Hence, T puts an X lock for $d=v_1$ and another X lock for $d=v_2$ on *AJV*. During the period that T holds these two X locks, no other transaction T'' can insert an entry for value v_3 into I_B such that $v_1\leq v_3<v_2$, as this would require T'' to obtain a V (or X) lock for $d=v_2$ on *AJV*. Also, T' cannot obtain the S value lock for $d=v_1$ on I_B. This is because to do so, depending on whether T has inserted a new entry for v_1 into I_B or not, T' needs to obtain an S lock for either $d=v_1$ or $d=v_2$ on *AJV*.

In the above three scenarios, the situation that T itself inserts an entry for value v_3 into I_B such that $v_1\leq v_3<v_2$ can be discussed in a way similar to that of Case 3.

Hence, for any value v_1, there is no read-write or write-read conflict between two different transactions on those tuples of the aggregate join view *AJV* whose attribute d has value v_1. As discussed at the beginning of this section, write-write conflicts do not exist and thus our key-range locking protocol guarantees serializability.

5. The Case of Non-aggregate Join Views with B-tree Indices

Implementing the V locking protocol for (non-aggregate) join views in the presence of B-tree indices is tricky. For example, suppose we do not use the latches in the latch pool. That is, we only use S, X, and V value locks on join views. Suppose we implement S, X, and V value locks for join views on B-tree indices in the same way as described in Section 2.4. Also, suppose a B-tree index I_B is built on attribute a of a join view $JV=\pi(\sigma(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n))$. Then to insert a new join result tuple t into JV, we need to first put a V value lock for $t.a$ on I_B. If no entry for $t.a$ exists in I_B, we need to find the smallest value v_2 in I_B such that $v_2>t.a$ and put a V lock for v_2 on I_B. Unfortunately, this approach does not work. The reason is similar to what is shown for Technique 1 in Section 3. (We can replace the V lock for value v_2 on I_B by an X lock. However, the X lock for v_2 on I_B cannot be downgraded to a V lock. Hence, this X lock greatly reduces concurrency.)

To implement value locks for join views on B-tree indices with high concurrency, we can use the latches in the latch pool and treat join views in the same way as aggregate join views. For join views, we still use four kinds of value locks: S, X, first kind V, and second kind V. For example, suppose a B-tree index I_B is built on attribute a of a join view JV. As described in Section 2.4, to insert a new join result tuple t into JV, we first put a second kind V value lock for $t.a$ on I_B. To delete a join result tuple t from JV, we first put a first kind V value lock for $t.a$ on I_B. For join views, all the four different kinds of value locks (S, X, first kind V, and second kind V) can be implemented on B-tree indices in the same way as described in Section 2.4. The only exception is that we no longer need the latch on the group by attribute value of tuple t. The correctness (serializability) of the implementation can be proved in a way similar to that described in Section 4. Note here, for join views, the latches in the latch pool are used for a different purpose from that for aggregate join views.

References

1. Bernstein P.A., Hadzilacos V., Goodman N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley publishers (1987)
2. Gray J., Bosworth A., Layman A. *et al.*: Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. ICDE 1996, pp. 152-159 (1996)
3. Gehrke J., Korn F., Srivastava D.: On Computing Correlated Aggregates over Continual Data Streams. SIGMOD 2001, pp. 13-24 (2001)
4. Gupta A., Mumick I.S.: Materialized Views: Techniques, Implementations, and Applications. MIT Press (1999)
5. Gray J., Reuter A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers (1993)
6. Graefe G., Zwilling M.J.: Transaction Support for Indexed Views. SIGMOD 2004, pp. 323-334 (2004)
7. Korth H.F.: Locking Primitives in a Database System. JACM 30(1), 55-79 (1983)
8. Kornacker M., Mohan C., Hellerstein J.M.: Concurrency and Recovery in Generalized Search Trees. SIGMOD 1997, pp. 62-72 (1997)
9. Luo G., Naughton J.F., Ellmann C.J., Watzke M.W.: Locking Protocols for Materialized Aggregate Join Views. VLDB 2003, pp. 596-607 (2003)
10. Luo G., Naughton J.F., Ellmann C.J., Watzke M.W.: Locking Protocols for Materialized Aggregate Join Views. TKDE 17(6), 796-807 (2005)
11. Mohan C.: ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. VLDB 1990, pp. 392-405 (1990)
12. Mohan C.: Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. VLDB 1990, pp. 406-418 (1990)
13. Poess M., Floyd C.: New TPC Benchmarks for Decision Support and Web Commerce. SIGMOD Record 29(4), 64-71 (2000)
14. Dver A.: Real-time Enterprise. Business week Dec. 2, 2002 issue (2002)