

# Locking Protocols for Two-Tier Indexing of Partitioned Data

David Choy, C. Mohan

IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120, USA  
{dmc, mohan}@almaden.ibm.com  
www.almaden.ibm.com/u/mohan/

## Abstract

In a parallel or a distributed database management system, a relation is often horizontally partitioned across multiple nodes. To index a partitioned relation, usually either a *global* index is maintained for the entire relation, or alternatively, individual *local* indexes are maintained one for each partition of the relation. The former is costly to maintain because of remote updates and is also costly to use for complex queries, whereas the latter wastes computing resources for highly selective database searches such as those done in the case of transaction processing and is therefore not a scalable solution.

In this paper, a two-tier index is proposed, which consists of local indexes and a *coarse* global index. This index method is suitable for transaction processing as well as for query processing. It not only is more efficient to maintain and use than the conventional methods, but it also exploits parallelism and is more versatile, scalable, and easier to migrate to for a non-partitioning DBMS. To maintain the consistency between the local indexes and the coarse global index, efficient locking protocols are presented, which are designed to allow a high level of concurrent operation.

## 1. Introduction

In a parallel or a distributed database management system, a set-oriented database object, such as a relation, may be *horizontally partitioned* over multiple database nodes using a partitioning function, such as a hash on a Partition Key of the object. The number of nodes that the object is partitioned across is called the *Degree of Partitioning* for the object, and each of these nodes is called a *Store Node* of the object. The set of records of the object stored at each of its store nodes is called a *Partition* of the object.

A database object is often indexed to enable associative access to its records. To index a partitioned object, one method is to maintain a separate *Local Index* at each of its store nodes. Each Local Index is maintained for the respective partition like a conventional index for a non-partitioned object. With this method, an index search request based on a key other than the partition key has to be sent to all the store nodes and be processed for every individual Local Index.<sup>1</sup> If qualified records are found in most partitions, this is an effective solution. If, however, qualified record(s) are found only in a small fraction of the partitions (e.g., for *Transaction Processing* or for checking key uniqueness), the processing at most of the store nodes will be wasteful, and hence the transaction throughput will suffer. Therefore, this *Local Index Only* method (see Figure 1) does not scale with the degree of partitioning.

An alternate method is to maintain a *Global Index* (see Figure 2) that indexes records in all the partitions of the object (e.g., as in Tandem's NonStop SQL [Tan87]). In this case, globally unique record IDs must be used in the index. A globally unique record ID may be a concatenation of a Partition ID (PID), which identifies a store node, and a local Record ID (RID). Or it may be the Partition Key value of a record if the Partition Key is unique (e.g., as in NonStop SQL). In the latter case, the PID for an indexed record can be derived from the Partition Key value by applying the partitioning function. The Global Index can be stored at any node, or can itself be partitioned.

---

<sup>1</sup> In some cases, broadcasting of the request is needed even if the search includes the partition key, e.g., a key-range search when hash partitioning is used.

However, a Full Global Index is costly to maintain due to remote updates (which incur communication as well as locking costs), and is also costly to use for *complex queries* because:

- Remote share (S) lock(s) must be acquired to assure serializability (i.e., repeatable read or degree 3 isolation [GrLo76]). Otherwise, the retrieved record(s) have to be re-certified for all index-search conditions.
- *Index ANDing/ORing* [MoHa90] (for multiple-index access) is expensive unless the participating Global Indexes are stored at the same node and the record ID lists obtained from the indexes are short. However, storing Global Indexes at the same node reduces parallelism and creates hot spot.
- If a large number of records are qualified, long lists of record IDs have to be sent to the respective store nodes for record retrieval.

We shall call this method the *Global Index Only* method. In NonStop SQL, the primary key is required to be unique and clustering of records is forced to be based on the primary key. Further, since the records are stored in the primary index, updates to the primary key are disallowed and multiple tables' records aren't allowed to be intermixed and stored in a clustered fashion. Locking is also costly and less concurrent since it is required to be based on key values, which can vary in length from index to index (see [MoLe92, Moha95] for details). If the global index were to refer to physical locations of records, as in DB2, then, the reorganization of even a single partition will require extensive, random updates to a global index since record IDs would change.

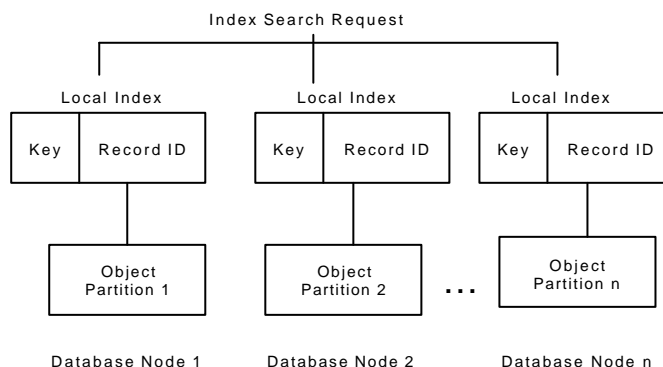
In this paper, we propose a *Two-Tier Index* for indexing partitioned data (see Figure 3), and present efficient locking protocols to ensure the consistency of the index while allowing a high level of concurrent operation. Discussions pertaining to the exploitation of two-tier indexes, such as when to create such an index, cost analysis, and query optimization, are beyond the scope of this paper.

The rest of this paper is organized as follows. Section 2 contains a description of the Two-Tier Indexing Method and the advantages of this approach. In Section 3, locking problems for maintaining a two-tier index are discussed. An efficient locking protocol that supports a high level of concurrency is then presented in Section 4. This is followed by examples in Section 5. Finally, a summary is given in Section 6.

## 2. A Two-Tier Index

We propose a two-tier indexing method for a partitioned database object. Under this scheme, a Local Index is maintained for each partition of the object. (Either a RID or a Primary Key may be used as a record pointer in a Local Index.) In addition, a *Coarse* Global Index is *optionally* maintained which, except for a Unique Index, indexes the partitions but not the individual records. For a Unique Index, the Global Index may also contain a record pointer for each indexed record to allow a global search to bypass a Local Index access. (However, this extra pointer is not required. Storing internal record pointers remotely may not be desirable for a distributed database in the interest of node integrity and autonomy.) Like a Full Global Index, the Coarse Global Index may be stored at any node and may be partitioned if necessary. This hierarchy of distributed indexes is maintained as a *single, consistent index structure* for the indexed object.

Even if multiple records in a partition have the same index key value, there will be only one entry for that key value and



**Figure 1 Indexing a Partitioned Object using Local Indexes**

that partition in the Coarse Global Index. Therefore, only when the first instance of a key value is inserted into a Local Index, a corresponding entry is inserted into the Global Index. Similarly, only when the last (i.e., the only) instance of a key value is deleted from a Local Index, its corresponding entry is deleted from the Global Index. Since updating a Global Index is much more expensive than updating a Local Index due to messages and two-phase-commit cost, this approach could significantly reduce the index maintenance cost compared to the Full Global Index approach. Other benefits of the two-tier indexing method are discussed below.

Two-tier indexes are typically used in the following manner to retrieve qualified records from a partitioned object. There are many opportunities for query optimization, which are beyond the scope of this paper.

1. If there is a selection predicate on the Partition Key that can be evaluated into PID(s), do so.
2. If there is an applicable and *selective* Coarse Global Index, obtain the qualified PID(s) from it.
  - Sort PIDs and remove duplicates; Merge them with the PIDs obtained from Step 1 if there are any. (AND or OR depending on the predicate)
  - For a Unique Index, local RID(s) are also obtained if available.
  - Additional Coarse Global Indexes may also be used [MoHa90].
  - If the PID list becomes long (i.e., the Global Index is not *selective* with respect to the selection predicate), the DBMS may abort the Global Index access (and release S locks if any are held) and proceed to broadcast or multicast the query instead (Step 3).
3. Otherwise, or if the query is not partition-selective, let the qualified PIDs be (logically) the set of all partitions.
4. Send the query to the identified store node(s) for parallel evaluation.

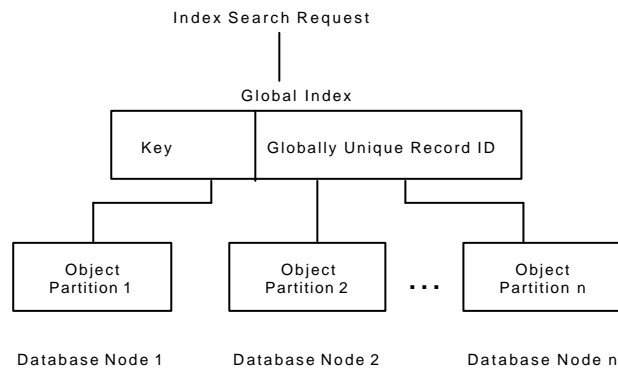
The choice of which Local Indexes to use may be different from the Global Indexes that were used in Step 2.

- Apply any suitable single-node evaluation techniques (e.g., index ANDing/ORing, list sorting, asynchronous I/O [ChHa91, TeGu84]).

Each node may use a different query evaluation plan depending on the partition statistics, the record clustering scheme, the applicable local access methods, and the availability and the relative costs of computing resources.

- For a non-repeatable-read transaction (in which case the scanned Global Index entries are not locked), a record that is retrieved using a RID obtained from a Unique Global Index should be re-certified to assure correct retrieval. (See [MoHa90, Mo90a] for descriptions of techniques to reduce the need for this re-evaluation.)
5. Merge the query results if the query is evaluated at multiple nodes.

The Coarse Global Index is mainly used to reduce the search scope so that a query may be routed to selected node(s) for processing. It may also be used to obtain global information on the index key such as the existence of a given key value (e.g., to assure key uniqueness or to verify a referential integrity constraint) or the cardinality of distinct key values.



**Figure 2 Indexing a Partitioned Object Using a Global Index**

Therefore, the Coarse Global Index helps to improve data access efficiency, reduce resource consumption, and enhance the scalability of a partitioned database. Thus the two-tier indexing method, like the *Global Index Only* method, can support a much higher transaction throughput than the *Local Index Only* method can, and, unlike the *Global Index Only* method, can scale with the degree of partitioning. This is particularly useful for transaction processing where queries are highly selective.

On the other hand, since every record and all its Local Index entries (as well as other local access methods) are always *co-located* at the same node, the performance of complex-query evaluation and access method maintenance is significantly enhanced because communication is reduced and the opportunity for parallelism is maximized. This is particularly useful for a secondary index.

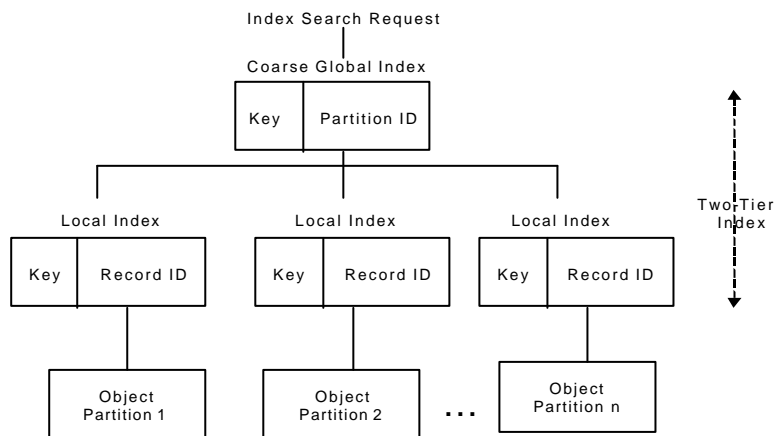
Furthermore, as a database design option, the Database Administrator (DBA) may choose to create only the Local Indexes on a key without the corresponding Coarse Global Index. This eliminates the cost of maintaining a Global Index altogether. This is useful

- When the index key is not partition-selective (i.e., each key value associates with many partitions),
- When there is already an adequate method to route queries predicated on this key (e.g., using the partition function to support equality search on the Partition Key), or
- When there is no need to obtain global information on this key either by application or by the DBMS.

Note that even if a Coarse Global Index is not maintained, the Local Indexes can still be useful for query evaluation within their respective partitions (to support queries routed to this node, or to support *local queries* for a distributed database).

Therefore, the *performance advantages* of the two-tier indexing method over the *Global Index Only* solution or the *Local Index Only* solution include the following:

- More efficient to maintain than the *Global Index Only* solution when multiple instances of a key value exist at each node  
(Fewer messages; two-phase-commit savings)
- More efficient to use  
(Consume computing resources for the qualified partitions only; route *function* (query) instead of *data* (record-pointer list); co-location of data and access methods)
- Maximum opportunity for parallelism; reduced likelihood of hot spot and lock contention  
(Parallelized: Local-Index access, Index ANDing/ORing, list sorting, index creation)
- Local Indexes being available to facilitate joining (e.g., using the nested-loop join method) when the partitioned relations are *co-located* with respect to the join operation (i.e., when both relations are partitioned on the join column)  
(Using a Global Index to do a nested-loop join would be very costly)
- More opportunities for query optimization  
(Using Coarse Global Indexes to route query; all local access methods being available for query evaluation; partition-specific statistics and execution plan)



**Figure 3 Indexing a Partitioned Object Using a Two-Tier Index**

These advantages are quite obvious and examples would be *trivial*.

In addition, the two-tier indexing method has the following *architectural advantages*:

- More scalable than the *Local Index Only* solution
- Compatible with any data partitioning scheme and clustering scheme [CoAl88, DeGh90]
- No separate logic needed to handle query broadcast, which is necessary to support search on an un-indexed field, and no special treatment needed for indexing the Partition Key
- Accommodate node heterogeneity and autonomy for a distributed database
- Easier for a conventional, single-node (non-partitioning) DBMS to migrate to

### 3. Maintenance of a Two-Tier Index

A correct maintenance of a two-tier index structure, in order to allow a high level of concurrency, must deal with many unobvious problems. First, locking must be performed for accessing each individual index, as in the conventional case, to allow index-only accesses to the Global Index (e.g., for performing key-existence checks or counting distinct key values) as well as to a Local Index (to support local queries). In addition, to maintain the consistency of the index hierarchy as a whole, a locking protocol must be specified to *correctly coordinate* the insertion and deletion of index entries between a Global Index and its Local Indexes. Specifically, the protocol must guarantee the following:

- If the Global Index indicates that a particular key value exists in a particular partition, then the Local Index for that partition should contain that key value.
- If the Global Index indicates that a particular key value is not present in a particular partition, then the Local Index of that partition should not contain that key value.

A high-level specification of the insertion and deletion of an index key may be described as follows:

#### **KEY INSERT:**

If this is a Unique Index and the key value being inserted is already present in the Local Index, then

Confirm that the already existing key is committed, or, that it is this transaction's uncommitted insert, by locking it in S mode. If, after the lock is acquired, the key is still there then return with a unique-key-violation error.

Insert the key in the Local Index.

If the key has a value that is not already present in any of the keys currently in the Local Index, then

If this is not a Unique Index, then insert an entry with that key value for this partition into the Global Index.

If this is a Unique Index and the key value being inserted is already present in the Global Index, then

Confirm that the already existing key is committed, or, that it is this transaction's uncommitted insert, by locking it in S mode. If after the lock is acquired the key is still there, then the Local Index insert is undone and a unique-key-violation error is returned.

If this is a Unique Index and the key value being inserted is not already present in the Global Index, then insert an entry with that key value for this partition into the Global Index.

#### **KEY DELETE:**

Delete the key from the Local Index.

If the deleted key is the last (only) instance of its key value in the Local Index, then

Delete the entry with that key value for this partition from the Global Index.

#### **Locking Problems**

While making the above checks for whether the Local Index insert is the insert of the very first instance of a key value or the delete is the delete of the only currently-present instance of a key value, we have to be sure that there is no uncommitted insert or delete which could subsequently falsify the above inferences. Otherwise, inconsistency can arise

between the Local Index and the Global Index. To illustrate this, let us consider the following example with transactions T1 and T2:

T1 Inserts <"San Jose", 100> into Local Index of partition P1.

Since this is the first instance of key value "San Jose" in the Local Index, T1 inserts the key <"San Jose", P1> into the Global Index.

T2 Inserts <"San Jose", 110> into Local Index of P1.

Since "San Jose" already exists, T2 does *not* update the Global Index.

T2 Commits.

T1 Rolls back, thereby deleting its "San Jose" entries from the Local Index and the Global Index.

This leaves the entry inserted by T2 in the Local Index without a corresponding entry in the Global Index.

The problem arose because T2 relied on the *uncommitted* key inserted by T1 to conclude that it (T2) did not have to update the Global Index. It should be noted that making T1, at the time of undoing the insert of <"San Jose", 100>, notice that there is still one more instance of *San Jose* left in that Local Index and hence not perform undo on the Global Index,<sup>2</sup> introduces a complex set of unobvious recovery and performance problems. With this approach, T1 would not do the delete from the Global Index. This could still cause inconsistency if T1's earlier insert into the Global Index does not survive a crash at the Global Index's node and its log records might have been lost from virtual storage.

A possible solution to this problem would be to make T2 ensure that what it sees in the Local Index is committed data by locking it. If, after the lock is granted, the original key value is still there, then it can conclude that the Global Index does not need to be updated. This approach forces the Inserter to acquire an extra lock, and, in the case of key value locking [Mo90b, Moha95], also prevents other transactions from inserting other keys with the same key value concurrently.

A similar problem also occurs if a Global Index operation relies on an uncommitted delete of the last instance of a key value in a Local Index by another transaction.

In the next section, we present a locking protocol which

- Minimizes the number of locks acquired to enhance efficiency,
- Uses minimum (i.e., least restrictive) lock modes and lock durations to improve concurrency,
- Is compatible with, and further enhances, existing index management schemes to support high performance operation, and
- Does not require a new method to handle recovery, i.e., each Global Index or Local Index can be recovered separately in a conventional way (e.g., as in System R [GrMc81] or ARIES [MoHa92, Moha99]).

We assume that the readers of this paper are familiar with the System R lock modes, the compatibility relationships amongst them and the durations of locking [GrLo76, MoLe92, Mo90b]. We also assume that each Global or Local Index is implemented using a B<sup>+</sup>-tree.

Besides maintaining the consistency between a Global Index and its Local Indexes, this protocol also exploits the following techniques to assure correct operation on each individual (Global or Local) index, while supporting high concurrency.

- To assure key uniqueness for a Unique Index

*Problem:* T1 deletes a key and T2 inserts a key with the same key value. If T2 commits and T1 rolls back, then there will be two instances of that key.

---

<sup>2</sup> This means that the undo of the Global Index is based on the current state of the Local Index after an undo is performed on the Local Index. This is called logical undo and it is discussed in [MoHa92] in the context of how undos for free space inventory pages are done at the time of undo of data page updates.

*Solution:* Before deleting a key, T1 leaves a *tombstone* by holding a lock on the next key for COMMIT duration. Before inserting a key, T2 acquires an incompatible lock on the next key for INSTANT duration to be sure of the absence of a tombstone.

- To support Repeatable-Read (RR)

*Problem:* A key read by a RR transaction T1 may be deleted by T2 before T1 commits, or a key not found by T1 may be subsequently inserted by T2, thereby making key reading non-repeatable to T1. Similarly, T1 may miss a key deleted by T2, and later T2 rolls back making the read non-repeatable to T1.

*Solution:* T1 acquires a lock for COMMIT duration on every key it reads. If a key is not found, a lock would be held on the next key. To delete a key, T2 acquires an incompatible lock on the key for INSTANT duration to make sure it has not been read by a RR transaction, and acquires an incompatible lock on the next key for COMMIT duration to prevent a RR transaction from missing the deleted key (i.e., reading the *hole*). To insert a key, T2 acquires an incompatible lock on the next key for INSTANT duration to make sure the inserted key has not been missed by a RR transaction, and acquires an incompatible lock on the inserted key for COMMIT duration to prevent other transactions from reading it.

- To avoid deadlocks involving latches

*Problem:* When any lock is requested while holding one or more page latches, if the lock is not immediately acquirable, then a deadlock involving the latch(es) could occur.

*Solution:* To avoid a deadlock involving latches, when a lock is not immediately available, all held latches must be released before waiting for the lock. If a lock is acquired without the appropriate latch(es) being held at the time of lock acquisition, then the previously inferred information must be revalidated by reacquiring the latch(es) [MoLe92, Mo90b, Moha95].

#### 4. An Efficient Locking Protocol

To minimize the cost of locking, we are able to limit locking to only those involving

- The key being inserted or deleted,
- The key following the one being inserted or deleted, and
- In the case of non-unique local indexes, occasionally, the key preceding the one being inserted or deleted.

In the following, an **Entry Locking** protocol is presented, which exploits the Commit\_LSN idea of [Mo90a]. It also enhances the ARIES/IM protocol [MoLe92, Moha95] by using additional lock modes to exploit more of the semantics of the operations and the state of the index, thereby allowing even higher concurrency between Reader, Inserter and Deleter transactions. For our purpose, an *Index Entry* is a  $\langle \text{Key Value}, \text{Pointer} \rangle$  pair. In a Local Index, the Pointer may be a RID or a Primary Key. In a Global Index, the Pointer may be a PID or a Partition Key, or if the index is unique, it may be a  $\langle \text{PID}, \text{RID} \rangle$  pair or a unique Partition Key. A similar protocol for **Key-Value Locking** is given in the Appendix. It is an extension of the ARIES/KVL protocol [Mo90b, Moha95].

The locks (mode and duration) that are to be acquired for the different indexes and different index operations are summarized in Table 1 and 2. For each index operation, the order in which these locks are acquired is: the next entry, the previous entry if necessary, and the current entry. An Insert or Delete operation that needs to be performed on the Global Index can be initiated only after all the locking required for performing the corresponding Local Index operation has been completed. This is because only after the latter is finished can it be correctly determined whether a Global Index operation is in fact needed.

	NEXT ENTRY	PREVIOUS ENTRY	CURRENT ENTRY
<b>READ</b>	No Lock	No Lock	{ <b>Manual, Commit</b> } <b>S</b> <b>Manual</b> IF Cursor-Stability <b>Commit</b> IF Repeatable-Read
<b>INSERT</b>	<b>Instant IX</b>	<b>Instant IS</b> IF (global exists) AND (next ≠ current) AND (prev=current) AND (LSN of prev's page ≥ Commit_LSN)	<b>Commit {X, SIX, IX}</b> <b>X</b> IF (first instance) OR (prev held in X) OR (next held in X) <b>SIX</b> IF (next held in S or SIX) <b>IX</b> Otherwise
<b>DELETE</b>	<b>Commit SIX</b>	<b>Instant S</b> IF (global exists) AND (next ≠ current) AND (prev=current) AND (LSN of prev's page ≥ Commit_LSN)	<b>Instant X</b>

**Table 1 Entry locking for Non-Unique Local Indexes**

	NEXT ENTRY	CURRENT ENTRY
<b>READ</b>	No Lock	{ <b>Manual, Commit</b> } <b>S</b> <b>Manual</b> IF Cursor-Stability <b>Commit</b> IF Repeatable-Read
<b>INSERT</b>	<b>Instant IX</b>	<b>Commit {SIX, IX}</b> <b>SIX</b> IF (next held in S, SIX or X) <b>IX</b> Otherwise
<b>DELETE</b>	<b>Commit SIX</b>	<b>Instant X</b>

**Table 2 Entry Locking Protocol for Unique Local Indexes and for Unique and Non-Unique Global Indexes**

We shall discuss the protocol's design for a Local Index only. The protocol for a Global Index is the same as that for a Unique Local Index (Table 2).

## 4.1 READ Operation

The locking for a read operation is the same as that under a conventional protocol: an S lock is acquired on the current entry. If a key is not found, then the next entry would be locked to ensure that the missed key is not currently in the uncommitted state and that it is not inserted subsequently by another transaction until the reading transaction terminates.

### 4.1.1 Current Entry

The duration of the lock depends on the *consistency level* required by the transaction:

For a Cursor-Stability (CS) transaction,

Lock current entry in **S** mode for **Manual** duration

(i.e., release the lock when the read cursor moves away from the key)

For a Repeatable-Read (RR) transaction,

Lock current entry in **S** mode for **Commit** duration

## 4.2 INSERT Operation

### 4.2.1 Next Entry

The next entry's lock must always be obtained. If there is no next entry, then a special End-of-Index lock that is specific to this Local Index is used for this purpose.

Lock next entry in **IX** mode for **Instant** duration

This lock handles the following:

- To ensure that no RR transaction has missed this key.
- For a Unique Index, to ensure that there is no uncommitted delete of the to-be-inserted key value.
- If there is no other instance of the to-be-inserted key, to ensure that there is no uncommitted delete of such a key so that a Global Index insert can be safely performed.
- If the next entry has the same key value as the to-be-inserted key, to ensure that it is not an uncommitted insert so that a Global Index insert is not needed.

### 4.2.2 Previous Entry

The previous entry needs to be locked only if the following conditions are true:

IF (non-unique index) AND (global index exists) AND

(next entry's key value > to-be-inserted entry's key value) AND

(previous entry's key value = to-be-inserted entry's key value) AND

((LSN of page containing previous entry) = Commit\_LSN)

THEN lock previous entry in **IS** mode for **Instant** duration

The intent behind this locking is

- To ensure that the Inserter of the first-inserted instance of the key value being inserted has committed and so a Global Index insertion is not needed, or
- To determine that the first-inserted instance of that key value is an uncommitted insert by the current transaction itself.

This lock request can be avoided if the Commit\_LSN mechanism [Mo90a] can be used to infer that the previous entry is in the committed state.

### 4.2.3 Current (To-Be-Inserted) Entry

The lock mode to be acquired on the to-be-inserted entry is determined by the following:

IF (this is a non-unique Local Index) AND

((first instance of key value is being inserted) OR

(previous entry had to be locked now and it was found to be already locked in X mode by current transaction) OR

(next entry already locked in X mode by current transaction))

THEN

lock entry to be inserted in **X** mode for **Commit** duration

ELSE

IF (next entry already locked in X, SIX or S mode by current transaction)

THEN

lock entry to be inserted in **SIX** mode for **Commit** duration

ELSE

lock entry to be inserted in **IX** mode for **Commit** duration

At least an IX lock needs to be acquired to ensure that other transactions do not read uncommitted data and that Deleters do not create a *tombstone* (see next section) on this uncommitted entry. Such a tombstone, if allowed, will disappear if the uncommitted insertion were to be rolled back.

If the current transaction had earlier performed a (repeatable) read of the next entry, the Lock Manager, via the return code on the IX lock request on the next entry, indicates that an S or SIX lock is being held by the current transaction. Then, the Reader-cum-Inserters next entry's S lock must be *added* on to the to-be-inserted entry to replicate the *missed-you* note, so that the newly inserted entry does not hide the *missed-you* note thereby allowing another transaction to insert an entry *behind* the newly inserted entry. (This is called *lock state replication via next key locking* in [Mo90b, Moha95].)

If the current transaction is the Inserters of the first-inserted instance of the current key value, then the return code from the Lock Manager, either when the Instant IX lock is requested on the next entry or when the Instant IS lock is requested on the previous entry, will indicate that the current transaction already holds an X lock on the respective entry. In this case, the current transaction will obtain an X lock on the current entry also. This is a left (and respectively, right) side propagation of the *uncommitted first instance* information. This propagation is not necessary if the Local Index is a Unique Index. In that case, the first Inserters needs to get only an IX lock and the one attempting a subsequent insert would request an S lock on the already existing entry to decide whether there is a unique-key violation. Before a unique-key violation can be reported it must be determined that either the first instance is in the committed state or that it is an insert by the current transaction.

### 4.3 DELETE Operation

#### 4.3.1 Next Entry

Lock next entry in **SIX** mode for **Commit** duration

This lock is acquired to handle the following:

- To leave behind a *tombstone* indicating an uncommitted delete to Reader transactions.

The lock on the tombstone entry must be such that no other transaction is able to (1) hide the tombstone by inserting an entry behind the tombstone entry with a higher key than the deleted key (i.e., to prevent the insert of a key between the deleted key and the tombstone key), and (2) delete the tombstone entry. (1) is taken care of by making Inserters acquire an IX lock on the next entry since IX conflicts with SIX, and (2) is taken care of by making Deleters acquire an X lock on the to-be-deleted entry. If the same transaction that performed the original delete inserts an entry behind the tombstone, then it finds out about the original tombstone via the Lock Manager return code on the next entry lock during insert and replicates the tombstone on to the newly inserted entry by acquiring an SIX or X lock on the new entry, instead of the usual IX lock. If the tombstone is being deleted by the same transaction then the tombstone gets replicated on to the entry next to the original tombstone due to the SIX lock that is acquired on the next entry during a delete.

- For a Unique Index, to alert a potential Inserters of the same key value about the uncommitted delete of that key value.
- If the next entry's key value is the same as that of the to-be-deleted entry, to ensure that at least one instance of that key value will not be deleted by another transaction until the current transaction terminates. Thus the Global Index entry cannot be deleted by another transaction and the current transaction can roll back safely.
- If the only instance of a key value is being deleted (and thus the corresponding Global Index entry will also be deleted), to ensure that no other transaction is able to insert any instance of that key value until the current transaction terminates. Such an Inserters would insert an entry into the Global Index and would create a problem if the current transaction were to roll back and restore the Global Index entry that it deleted.

#### 4.3.2 Previous Entry

The previous entry needs to be locked only if the following conditions are true:

```
IF (non-unique index) AND (global index exists) AND
  (next entry's key value > to-be-deleted entry's key value) AND
  (previous entry's key value = to-be-deleted entry's key value) AND
  ((LSN of page containing previous entry) = Commit_LSN)
THEN lock previous entry in S mode for Instant duration
```

The intent of this lock is

- To see if there exists currently in the Index at least one instance of the key value being deleted that is in the committed state, or
- To determine that all the existing instances of that key value are uncommitted inserts by the current transaction.

If the current transaction thinks that it is not deleting the only instance of the key value and hence it does not have to delete the Global Index entry, then it has to make sure that later all the remaining instances of the key value do not get deleted because they are all uncommitted inserts and all the inserting transactions roll back those inserts, thereby erroneously leaving behind the key value in the Global Index. If all the remaining instances of the key value are uncommitted inserts of transactions other than the current one, then the current transaction has to wait (e.g., by requesting an S lock on the previous entry) until at least one of those transactions terminates and check the state of the keys again.

If the current transaction is the one that inserted the previous entry (and, possibly, all other instances of the key value also), then the current transaction can safely delete the current entry. This is because if the current transaction rolls back then it will reinsert the current entry and so there will be no need to modify the Global Index. On the other hand, if the current transaction were to subsequently delete (in forward processing) all the other instances then it will at that time delete the corresponding entry in the Global Index.

### 4.3.3 Current (To-Be-Deleted) Entry

Lock entry to be deleted in **X** mode for **Instant** duration

## 4.4 Remarks

Comparing this protocol to ARIES/IM, additional concurrency is provided by exploiting more lock modes and sometimes an additional lock (on the previous entry) is acquired. The latter occurs only in a non-unique Local Index when the entry being inserted (or, deleted) is the *rightmost* instance of an already existing key value. This extra lock may be avoided if the Commit\_LSN technique [Mo90a] is used to infer that the previous entry is a committed one.

Furthermore, index insertion and deletion normally follow the insertion, deletion, or update of a data record. If a *data-only locking* approach [MoLe92, Moha95] is followed, the Index Manager may often forego the locking on the current entry. This is because either an IX lock (in the case of a record insertion) or an X lock (in the case of a record update or deletion) would have been obtained on the corresponding data record before the Index Manager is called.

## 5. Examples

The intricacy of this protocol can be illustrated through examples. Consider the following non-unique index in a committed state, where each index entry is a <Key Value, Pointer> pair.

Local Index (L1) on partition 1 (P1) contains:

<"A",100>, <"B",105>, <"B",300>, <"G",155>, <"H",50>

Local Index (L2) on partition 2 (P2) contains:

<"B",200>, <"C",50>, <"C",110>, <"G",108>

The corresponding coarse Global Index (GI) contains:

<"A",1>, <"B",1>, <"B",2>, <"C",2>, <"G",1>, <"G",2>, <"H",1>

Each of the following scenarios acts independently on this two-tier index.

### 5.1 Scenario 1

T1 inserts <"D",305> and then <"D",310> and <"D",306> into L1. Before T1 commits, T2 tries to insert <"D",400> into L1.

T1: L1 processing:

Locks Instant IX <"G",155> (next entry)  
Locks Commit X <"D",305> (current entry; first instance of "D")  
Inserts <"D",305> into L1  
Determines the need to insert <"D",1> into GI

GI processing:

Locks Instant IX <"G",1> (next entry)  
Locks Commit IX <"D",1> (current entry)  
Inserts <"D",1> into GI

L1 processing:

Locks Instant IX <"G",155> (next entry)  
Locks Instant IS <"D",305> (previous entry)  
Locks Commit X <"D",310> (X on current since previous held in X)  
Inserts <"D",310> into L1  
Determines no need for insert into GI

L1 processing:

Locks Instant IX <"D",310> (next entry)  
Locks Commit X <"D",306> (X on current since next held in X)  
Inserts <"D",306> into L1  
Determines no need for insert into GI

T2: L1 processing:

Locks Instant IX <"G",155> (next entry)  
Waits to acquire Instant IS on <"D",310> (previous entry)  
(blocked by T1's Commit X)

Delaying T2's insert until T1 commits or rolls back is necessary. Otherwise, T2 has to decide whether or not to insert <"D",1> into GI based on T1's uncommitted insert. The X lock on <"D",310> (respectively, <"D",306>) is a right (left) side propagation of the *uncommitted first instance* information. Otherwise, <"D",310> and <"D",306> would hide the uncommitted first instance <"D",305>.

## 5.2 Scenario 2

T1 inserts <"B",305> into L1. Before T1 commits, T2 inserts <"B",400> into L1.

T1: L1 processing:

Locks Instant IX <"G",155> (next entry)  
Locks Instant IS <"B",300> (previous entry)  
Locks Commit IX <"B",305> (current entry; not first instance)  
Inserts <"B",305> into L1  
Determines no need for insert into GI

T2: L1 processing:

- Locks Instant IX <"G",155> (next entry)
- Locks Instant IS <"B",305> (previous entry)
- Locks Commit IX <"B",400> (current entry; not first instance)
- Inserts <"B",400> into L1
- Determines no need for insert into GI

T2 can proceed since there exists a committed "B" in L1. This is enabled by T2's ability to acquire the IS lock on <"B",305> since it is compatible with the IX lock held by T1.

### 5.3 Scenario 3

T1 deletes <"B",300> and <"G",155> from L1. Before T1 commits, T2 tries to insert <"B",5> and <"G",400> into L1.

T1: L1 processing:

- Locks Commit SIX <"G",155> (next entry)
- Locks Instant S <"B",105> (prev entry; to ensure not uncommitted)
- Locks Instant X <"B",300> (current entry)
- Deletes <"B",300> from L1
- Determines no need to delete from GI

L1 processing:

- Locks Commit SIX <"H",50> (next entry)
- Locks Instant X <"G",155> (current entry)
- Deletes <"G",155> from L1
- Determines the need to delete <"G",1> from GI

GI processing:

- Locks Commit SIX <"G",2> (next entry)
- Locks Instant X <"G",1> (current entry)
- Deletes <"G",1> from GI

T2: L1 processing:

- Locks Instant IX <"B",105> (next entry)
- Locks Commit IX <"B",5> (current entry)
- Inserts <"B",5> into L1
- Determines no need for insert into GI

L1 processing:

- Waits to acquire Instant IX on <"H",50> (next entry)
- (blocked by T1's Commit SIX)

In spite of the uncommitted delete of <"B",300> by T1, T2 is able to insert another instance of *B*. This is okay because there is a committed instance of *B* (<"B",105>).

Delaying T2's insert is necessary to prevent T2 from inserting a <"G",1> into G based on the uncommitted delete of <"G",155> by T1.

#### 5.4 Scenario 4

T1 inserts <"G",220> into L1 and then deletes <"G",155> from L1. Before T1 commits, T2 inserts <"G",230> into L1.

T1: L1 processing:

- Locks Instant IX <"H",50> (next entry)
- Locks Instant IS <"G",155> (previous entry)
- Locks Commit IX <"G",220> (current entry)
- Inserts <"G",220> into L1
- Determines no need for insert into G

L1 processing:

- Locks Commit SIX <"G",220> (next entry)
- Locks Instant X <"G",155> (current entry)
- Deletes <"G",155> from L1
- Determines no need for delete from G

T2: L1 processing:

- Locks Instant IX <"H",50> (next entry)
- Locks Instant IS <"G",220> (previous entry)
- Locks Commit IX <"G",230> (current entry)
- Inserts <"G",230> into L1
- Determines no need for insert into G

Even though <"G",220> is an uncommitted insert by T1, T2 is able to insert <"G",230> immediately since the IS lock requested by T2 on <"G",220> is compatible with the SIX lock held by T1. This is okay because there will be a committed instance of "G" in L1 in any case. If T1 rolls back, then <"G",155> will remain. If T1 commits, then <"G",220> will be committed. If T1 commits after a partial abort, which causes the delete to be undone but not the insert, both <"G",155> and <"G",220> will remain when T1 terminates.

If <"G",155> had not existed in L1 before T1 began, then an X lock would have been obtained by T1 on <"G",220>. This would block T2's insertion until T1 terminates.

#### 5.5 Scenario 5

T1 reads records with key value A via L1 (repeatable-read), and then inserts <"A",108> and <"A",101> into L1. Before T1 commits, T2 tries to insert <"A",103> into L1.

T1: L1 processing:

- Locks Commit S <"A",100>
- Reads record 100
- Locks Commit S <"B",105> (no more "A" is found in L1)

L1 processing:

- Locks Instant IX <"B",105> (next entry)

Locks Instant IS <"A",100> (previous entry)  
Locks Commit SIX <"A",108> (SIX on current since next held in S)  
Inserts <"A",108> into L1  
Determines no need for insert into G1

L1 processing:

Locks Instant IX <"A",108> (next entry)  
Locks Commit SIX <"A",101> (SIX on current since next held in SIX)  
Inserts <"A",101> into L1  
Determines no need for insert into G1

T2: L1 processing:

Waits to acquire Instant IX on <"A",108> (next entry)  
(blocked by T1's Commit SIX)

The Commit S lock held by T1 on <"B",105> prevents an insertion of *A* by another transaction. The Commit SIX lock on <"A",108> propagates this S lock to the newly inserted entry so that no other transaction can insert an entry between <"A",100> and <"A",108>. A similar propagation happens for <"A",101>.

## 6. Summary

In this paper, we proposed a two-tier index structure for indexing a partitioned database object and presented an efficient method to maintain such an index to support a high-concurrency operation. This indexing scheme has a variety of advantages over the conventional schemes, including exploitation of parallelism for complex queries and efficient search for selective queries. Therefore, it is a versatile and scalable solution suitable for both Query Processing and Transaction Processing.

The two-tier index structure can easily be extended to a multi-tier one in a large network of database nodes. For example, a (coarse) *global* index may be maintained for each cluster of nodes connected by a LAN, and a *more global* index may be maintained for a geographic area comprising of multiple clusters connected by a WAN, and so on. The conventional indexing methods would not be feasible in this environment. Each *global* index in this multi-tier structure may be associated with a particular search scope (i.e., a semantic subset) of a relation. A database query qualified by a specific search scope may use the suitable *global index(es)* from this hierarchy of indexes.

The proposed indexing scheme offers many opportunities for optimization. An analysis of various cost trade-offs and a discussion on query optimization are beyond the scope of this paper.

## 7. References

- [ChHa91] Cheng, J., Haderle, D., Hedges, R., Iyer, B., Messinger, T., Mohan, C., Wang, Y. *An Efficient Hybrid Join Algorithm: a DB2 Prototype*, **Proc. 7th International Conference on Data Engineering**, Kobe, April 1991. An expanded version of this paper is available as **IBM Research Report RJ7884**, IBM Almaden Research Center, December 1990.
- [CoAl88] Copeland, G., Alexander, W., Boughter, E., and Keller, T., *Data Placement in Bubba*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Chicago, May 1988.
- [DeGh90] DeWitt, D., Ghandeharizadeh, S., Schneider, D., Bricker, A., Hsiao, H.-I., and Rasmussen, R., *The Gamma Database Machine Project*, **IEEE Transactions on Knowledge and Data Engineering**, Vol. 2, No. 1, March 1990.
- [Fa85] Faloutsos, C., *Access Methods for Text*, **ACM Computing Surveys**, Vol. 17, No. 1, pp. 49-74, March 1985.
- [GrLo76] Gray, J., Lorie, R., Putzolu, F., and Traiger, I., *Granularity of Locks and Degrees of Consistency in a Shared Data Base*, **Proc. IFIP Working Conference on Modelling of Database Management Systems**, Freudenstadt, January 1976.
- [GrMc81] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., and Traiger, I., *The Recovery Manager of the System R Database Manager*, **ACM Computing Surveys**, Vol. 13, No. 2, June 1981.

- [Gu84] Guttman, A., *R-trees: A Dynamic Index Structure for Spatial Searching*, **Proc. ACM SIGMOD Conference on Management of Data**, pp. 47-57, 1984.
- [Lo93] Lomet, D. *Key Range Locking Strategies for Improved Concurrency*, **Proc. 19th International Conference on Very Large Data Bases**, Dublin, August 1993.
- [Mo90a] Mohan, C. *Commit\_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990.
- [Mo90b] Mohan, C. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. A different version of this paper is available as **IBM Research Report RJ7008**, IBM Almaden Research Center, September 1989
- [MoHa90] Mohan, C., Haderle, D., Wang, Y., Cheng, J. *Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques*, **Proc. International Conference on Extending Data Base Technology**, Venice, March 1990. An expanded version of this paper is available as **IBM Research Report RJ7341**, IBM Almaden Research Center, March 1990; Revised May 1990.
- [MoHa92] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, **ACM Transactions on Database Systems**, Vol. 17, No. 1, March 1992.
- [Moha95] Mohan, C. *Concurrency Control and Recovery Methods for B<sup>+</sup>-Tree Indexes: ARIES/KVL and ARIES/IM*, In **Performance of Concurrency Control Mechanisms in Centralized Database Systems**, V. Kumar (Ed.), Prentice Hall, 1995.
- [Moha99] Mohan, C. *Repeating History Beyond ARIES*, **Proc. 25th International Conference on Very Large Data Bases**, Edinburgh, September 1999.
- [MoLe92] Mohan, C., Levine, F. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, **Proc. ACM SIGMOD International Conference on Management of Data**, San Diego, June 1992. A longer version of this paper is available as **IBM Research Report RJ6846**, IBM Almaden Research Center, August 1989.
- [Tan87] The Tandem Database Group *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*, **Proc. 2nd International Workshop on High Performance Transaction Systems**, Asilomar, September 1987.
- [TeGu84] Teng, J., and Gumaer, R., *Managing IBM Database 2 Buffers to Maximize Performance*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.
- [WeVo01] Weikum, G., Vossen, G. **Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery**, Morgan Kaufmann, 2001.

## 8. Appendix: Key-Value Locking Protocol

A Key-Value Locking protocol for a two-tier index is given here. It is an enhancement of the ARIES/KVL protocol [Mo90b, Moha95]. Tables 3, 4 and 5 show the locks that are to be acquired. Notice that a lock on the previous key value is not needed. Also, note that for non-unique indexes next locking is done only sometimes. The protocol for a unique index (Table 5) is identical to that of entry locking (Table 2).

	NEXT KEY VALUE	CURRENT KEY VALUE
<b>READ</b>	No Lock	{Manual, Commit} S <b>Manual</b> IF Cursor-Stability <b>Commit</b> IF Repeatable-Read
<b>INSERT</b>	<b>Instant IX</b> IF (first instance)	<b>Commit {X, IX}</b> <b>X</b> IF (first instance) AND ((global index exists) OR (next held in S, SIX or X)) <b>IX</b> Otherwise
<b>DELETE</b>	<b>Commit SIX</b> IF (only instance)	{ <b>Instant, Commit</b> } X <b>Instant</b> IF (only instance) <b>Commit</b> Otherwise

Table 3 Key-Value Locking Protocol for Non-Unique Local Indexes

	NEXT KEY VALUE	CURRENT KEY VALUE
<b>READ</b>	No Lock	{ <b>Manual, Commit</b> } S <b>Manual</b> IF Cursor-Stability <b>Commit</b> IF Repeatable-Read
<b>INSERT</b>	<b>Instant IX</b> IF (first instance)	<b>Commit {SIX, IX}</b> <b>SIX</b> IF (next held in S, SIX or X) <b>IX</b> Otherwise
<b>DELETE</b>	<b>Commit SIX</b> IF (definitely or possibly only instance)	{ <b>Instant, Commit</b> } X <b>Instant</b> IF (only instance) <b>Commit</b> Otherwise

Table 4 Key-Value Locking Protocol for Non-Unique Global Indexes

	NEXT KEY VALUE	CURRENT KEY VALUE
<b>READ</b>	No Lock	{ <b>Manual, Commit</b> } S <b>Manual</b> IF Cursor-Stability <b>Commit</b> IF Repeatable-Read
<b>INSERT</b>	<b>Instant IX</b>	<b>Commit {SIX, IX}</b> <b>SIX</b> IF (next held in S, SIX or X) <b>IX</b> Otherwise
<b>DELETE</b>	<b>Commit SIX</b>	<b>Instant X</b>

Table 5 Key-Value Locking Protocol for Unique Local Indexes and Unique Global Indexes