Database Management System (DBMS)

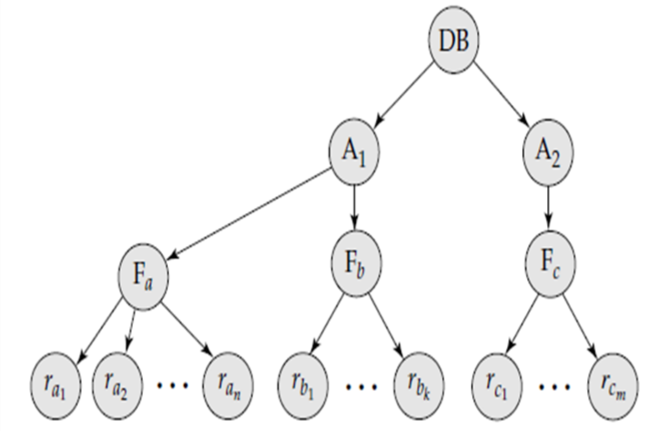
Lecture-44

Dharmendra Kumar

January 1, 2021

Multiple Granularity

Consider the following granularity hierarchy.



This tree consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type area; the database consists of exactly these areas. Each area in turn has nodes of type file as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type record. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

Concurrency Control

This protocol uses the following compatibility matrix to lock the data items.

Lock Compatibility Matrix

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in intention-shared (IS) mode, explicit locking is being done at a lower level of the tree, but with only shared-mode locks.

Concurrency Control

Similarly, if a node is locked in intention-exclusive (IX) mode, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks.

Finally, if a node is locked in shared and intention-exclusive (SIX) mode, the sub-tree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks.

Concurrency Control

The multiple-granularity locking protocol, which ensures serializability, is this:

Each transaction T_i can lock a node Q by following these rules:

1. It must observe the lock-compatibility function shown in above matrix.
2. It must lock the root of the tree first, and can lock it in any mode.
3. It can lock a node Q in S or IS mode only if it currently has the parent of Q locked in either IX or IS mode.
4. It can lock a node Q in X, SIX, or IX mode only if it currently has the parent of Q locked in either IX or SIX mode.
5. It can lock a node only if it has not previously unlocked any node (that is, T_i is two phase).
6. It can unlock a node Q only if it currently has none of the children of Q locked.

Clearly, the multiple-granularity protocol requires that locks be acquired in top-down (root-to-leaf) order, whereas locks must be released in bottom-up (leaf-to-root) order.

Example:

Consider the tree shown in the above figure and these transactions:

- Suppose that transaction T_{18} reads record r_{a_2} in file F_a. Then, T_{18} needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock r_{a_2} in S mode.
- Suppose that transaction T_{19} modifies record r_{a_9} in file F_a. Then, T_{19} needs to lock the database, area A_1, and file F_a in IX mode, and finally to lock r_{a_2} in X mode.

Concurrency Control

- Suppose that transaction T_{20} reads all the records in file F_a. Then, T_{20} needs to lock the database and area A_1 (in that order) in IS mode, and finally to lock F_a in S mode.
- Suppose that transaction T_{21} reads the entire database. It can do so after locking the database in S mode.

Clearly, transactions T_{18}, T_{20}, and T_{21} can access the database concurrently. Transaction T_{19} can execute concurrently with T_{18}, but not with either T_{20} or T_{21}.

Concurrency Control

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

- Short transactions that access only a few data items
- Long transactions that produce reports from an entire file or set of files

Note: Deadlock is possible in this protocol.

Multiversion Schemes

In multiversion concurrency control schemes, each write(Q) operation creates a new version of Q. When a transaction issues a read(Q) operation, the concurrencycontrol manager selects one of the versions of Q to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures serializability.

Multiversion Timestamp Ordering

With each data item Q, a sequence of versions $< Q_1, Q_2, ..., Q_m >$ is associated. Each version Q_k contains three data fields:

- **Content** is the value of version Q_k.
- **W-timestamp(Q_k)** is the timestamp of the transaction that created version Q_k.
- **R-timestamp(Q_k)** is the largest timestamp of any transaction that successfully read version Q_k.

A transaction T_i creates a new version Q_k of data item Q by issuing a write(Q) operation. The content field of the version holds the value written by T_i. The system initializes the W-timestamp and R-timestamp to TS(T_i). It updates the R-timestamp value of Q_k whenever a transaction T_j reads the content of Q_k, and R-timestamp(Q_k) $<$ TS(T_j).

Concurrency Control

The multiversion timestamp-ordering scheme operates as follows. Suppose that transaction T_i issues a read(Q) or write(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

1. If transaction T_i issues a read(Q), then the value returned is the content of version Q_k.
2. If transaction T_i issues write(Q), and if TS(T_i)¡R-timestamp(Q_k), then the system rolls back transaction T_i. On the other hand, if TS(T_i) $=$ W-timestamp(Q_k), the system overwrites the contents of Q_k; otherwise it creates a new version of Q.

Concurrency Control

Versions that are no longer needed are removed according to the following rule. Suppose that there are two versions, Q_k and Q_j , of a data item, and that both versions have a W-timestamp less than the timestamp of the oldest transaction in the system. Then, the older of the two versions Q_k and Q_j will not be used again, and can be deleted.

Note:

1. The multiversion timestamp-ordering scheme ensures serializability.
2. The multiversion timestamp-ordering scheme does not ensure recoverability and cascadelessness.