Structured Locking

- Optimality of 2PL.
- Locking schedulers that exploit structure to relax 2PL.
- B-tree Locking

Papers

- B-tree locking paper from Stonebraker.
- ARIES I/M paper
- Access Method Concurrency and recovery paper by Lomet.

Optimality of 2PL

- Two phase locking protocol is characterized by transactions observing the following two rules:
  - *Well-formed:* A transaction holds an appropriate lock on the data item before accessing it.
  - *Two-Phased:* A transaction does not lock any more data items, after it has unlocked some data item.
- We saw earlier that the above two rules are sufficient to ensure that schedule are conflict serializable.
- Are the above two rules ***necessary*** for ensuring CSR?
- That is, do there exist locking protocols that ensure conflict serializability that may permit certain transactions to either not be well formed, or not two phased.

Situations When 2PL Rules Can be Relaxed

- Assume that each transaction accesses consists of exactly one database operation. In such a case, transactions do not need to acquire any locks. Schedules produced by the execution of such transactions is serial.

- Assume that each transaction acquires locks on more than half the set of data items and maintains these locks until the end of the transaction. It can then acquire and release locks arbitrarily, or not acquire any further locks at all.

- Assume that each transaction acquires a lock on a *specific* data item as the first operation and does not release the lock until the end of the transaction. It can then acquire and release locks arbitrarily, or not acquire any further locks at all.

Necessity of 2PL Rules

- In general, if
  - Transactions are not trivial.

    they may access more than one data item, and less than half the database.
  - There is no structure over the database that transaction's data access conforms to.

    no partial or total ordering can be superimposed on the data items based on the access pattern of transactions

  violating the 2PL rules may result in non-serializable executions.

- However, if the data accesses of a transaction are structured, then transactions may violate the 2PL rules without jeopardizing CSR.

Structured Locking: Path Protocol

- Assume that all the objects in the database can be linearly ordered as $x_1, x_2, \ldots, x_n$.
- To simplify discussion, let us not distinguish between read and write operations– that is, any two operations belonging to different transaction conflicts.
- Each transaction accesses data items that are consecutive in the above order.
- Thus,

  $$T_1 : o(x_2)\ o(x_3)$$

  is a legal transaction.
- However, the following transactions are not legal:

  $$T_1 : o(x_3)\ o(x_2)$$
  $$T_1 : o(x_2)\ o(x_4)$$

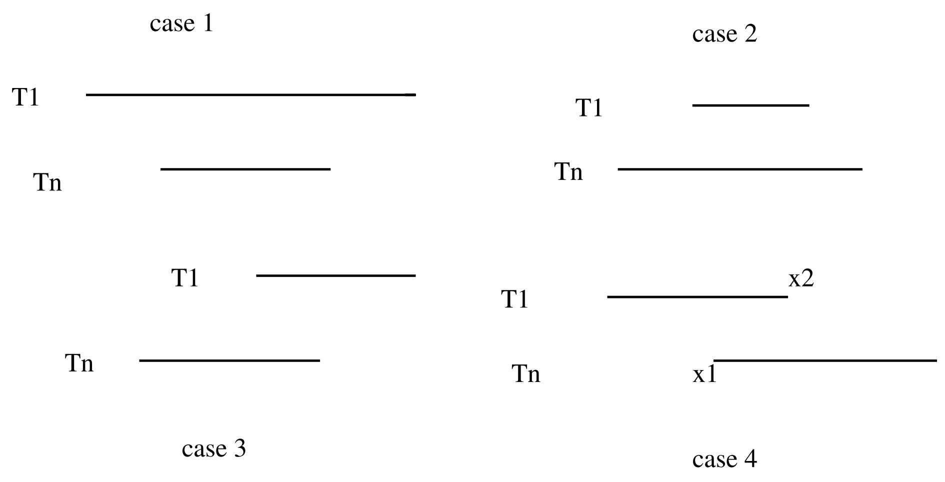
Path Protocol

- A transaction T_i, before accessing data item x_j acquires a lock on x_j.
- T_i can acquire a lock on x_j only if it has a lock on x_{j-1}, unless x_{j-1} is the first data item accessed by T_i.
- T_i does not release a lock on x_j before it has accessed x_j.
- No two transactions acquire conflicting locks on x_j at the same time.
- The path protocol does not follow the two-phase locking rule: that is, transaction T_i may acquire locks even after it has released a lock on some other data item.
- The path protocol ensures that the schedules are CSR.

Proof of Path Protocol

1. For any pair of transactions T_i and T_j, if T_i accesses a data item x before T_j, then T_i accesses any data item that T_i and T_j access in common before T_j. (proof of this follows from path protocol directly)
2. Thus, if there is an edge from T_i to T_j in SG, then T_i accesses all the data items that T_i and T_j access in common before T_j does.
3. We can further show that if there is a path $T_1 \rightarrow T_2 \rightarrow \cdots T_n$ in the SG, and T_1 accesses some data item x in common with T_n, then T_1 accesses the data item before T_n. (proof based on induction on size of dependency chain)
4. The above implies that the SG is acyclic and hence the execution is serializable.

Proof of Path Protocol



- **Base Case:** If $T_1 \to T_n$, then the claim holds by 2.
- **Induction:** Assume claim holds for $\leq n-1$. Show it also holds for n. Since T_1 and T_n access a data item in common, their data accesses are as shown in one of the 4 figures above.
- **Case 1:** Consider T_{n-1}. Since T_{n-1} accesses a data item in common with T_n, it accesses a data item in common with T_1. By IH, T_{n-1} accesses all data items in common with T_1 after T_1 and hence T_n accesses all data items in common with T_1 after T_1. Hence proved.
- **Case 2:** Proof similar to case 1.
- **Case 3:** Let x_1 be the smallest data item accessed by T_1 and x_2 be the largest data item accessed by T_n in the data ordering. Since $T_1 \to T_2 \cdots \to T_n$, there exists a transaction T_k, $k = 2, 3, \cdots, n-1$ such that T_k accesses a data item x' where $x1 \leq x' \leq x2$. By IH, T_1 accesses x' before T_k which accesses x' before T_n. Hence, T_1 accesses all data items in common with T_n before T_n does. Hence proved.
- **Case 4:** similar to case 3.

Generalizations of the Path Protocol

- **Tree protocol:** Assume that the set of data items are structured as nodes of a tree. For such a database CSR is assured if transactions follow the *tree protocol*.
  - The tree protocol differs from the path protocol in that the transaction before it acquires a lock on data item x, it must have a lock on the parent of x, unless x is the first data item accessed by T_i.
- Recall that we have discussed the path and tree protocols without distinguishing between the read and write operations.
- The path and tree protocols can be generalized to the case in which read and write accesses of transactions are distinguished.
- Structured locking protocols are very useful in the context for access method (B-tree) concurrency control.

B-tree Concurrency Control

- Operations over a B-tree:
  - insert(k): The insertion descends from root to leaf searching for the node in which the key value is to be inserted. If insertion causes a page split, then the page split is propogated to higher nodes and the routing information in B-tree is modified.
  - delete(k): The deletion descends from root to leaf and deletes the entry from the leaf node. If deletion causes utilization to go below 1/2, the leaf is merged with its siblings.
  - search(k): search descends from root to leaf.
  - range search(k1, k2): search descends from root to leaf to search for k1. Links between leaf nodes are followed to retrieve all records whose key value k is between k1 and k2.
- B-tree concurrency control algorithm must ensure concurrent operations on a B-tree are isolated from each other.
- Since a large number of transactions access a B-tree, the concurrency control algorithm must provide very high concurrency and freedom from deadlocks.

Strategy 1

- **Search:**

```
 S-lock the root.
 current-node = root.
 While (current-node is not a leaf) /*  lock coupling */
    Place S lock on appropriate child
    Release lock on current-node
    Make child the current node
```

- **Update:**

```
X lock the root;
current-node = root;
While (current-node not leaf)
place X lock on appropriate child;
current-node = child;
if (current-node is safe)
release locks on all ancestors;
```

Strategy 2

- Strategy 1 will cause an inserter to block searchers until the inserter is able to find a safe node in its descent.
- To overcome loss of concurrency, in Strategy 2 updater optimistically descends the tree as if it is a searcher using coupling of S-lock until it reaches the leaf in which case it acquires an X lock.
- If the update does not cause tree modification, nothing more needs to be done. Else, it retries the insertion using strategy 1.

```
S lock the root;
current-node = root;
While (current-node not leaf)
if child is not leaf
   place S lock on appropriate child;
else place X lock on child
release lock on current node;
current-node = appropriate child
if (current-node is not safe)
   release all locks and retry using strategy 1
```

Strategy 3

- Strategy 2 causes multiple tree traversal in case the optimisitic guess is not valid. Strategy 3 overcomes that using a new lock mode.
- U mode lock does not conflict with other searchers but conflicts with other U and X locks.
- Updater descends tree using a U lock and acquires U locks on all nodes on its until it reaches a safe node in which case it releases locks held on anscestors.
- When it reaches a leaf it converts all its U locks to X locks.

```
U lock the root;
current-node = root;
While (current-node not leaf)
place U lock on appropriate child;
current-node = child;
if (current-node is safe)
release locks on all ancestors;
Convert, top down, all U locks to X locks.
```

B-tree Concurrency Control

- Due to the importance of B-trees in databases, and the need for very high concurrency, many concurrency control algorithms for B-trees have been developed.
- The above described strategies are among the first developed for B-tree concurrency control. The three strategies can be combined into a single generalized solution.
- Among the most popular approaches are the ARIES/IM (Mohan) and its generalization – Pi-tree concurrency control (Lomet and Salzberg).
- Concurrency control algorithms for hashing index mechanisms have also been developed – ARIES/LHS (Mohan).
- We will learn that modern applications require novel, new data structures to be incorporated into databases as access methods.
- One of the big impediment in incorporating these data structures into DBMSs is the need to develop concurrency control algorithms for the data structures.
- Recently, concurrency control algorithms have also been developed for R-trees, Grid files, Generalized Search Trees, and hB-trees.
- Pi-tree paper provides a generic mechanism for concurrency control in tree-based data structures.