More Dynamic Databases

- If the set of DB objects changes, Strict 2PL using row or page locks will not ensure serializability (locking whole tables will work but is horribly slow)

  Example:
  - T1 finds oldest sailor for each of rating=1 and rating=2
  - T2 does an insertion and a deletion
    1. T1 locks all pages with rating = 1, finds oldest sailor (age = 71)
    2. Next, T2 inserts a new sailor; rating = 1, age = 96
    3. T2 deletes oldest sailor with rating = 2 (age = 80), commits
    4. T1 locks all pages with rating = 2, and finds oldest (age = 63)

  No serial schedule gives same outcome!

The “Phantom” Problem

- T1 implicitly assumes that it has locked the set of all sailor records with rating = 1
  - Assumption only holds if no sailor records are added while T1 is executing!

  Two mechanisms to address the problem
  - Index locking
  - Predicate locking

Another phantom example

- Table tasks has one row for each worker task, with worker name, task name, number of hours
- Rule that no worker has more than 8 hours total
- Application A to add a task sums hours for worker, adds task if it fits under 8 hours max
  - T1 running A sees ‘Joe’ has 6 hours, adds task of 2 hours
  - Concurrently, T2 running A sees ‘Joe’ has 6 hours, adds task of 1 hour.
  - Joe ends up with 9 hours of work.

- Again, the problem is there is no lock on the set of rows being examined to make a decision

Index Locking

- Assume index on the rating field
  - T1 should lock the index page(s) containing the data entries with rating = 1, and their immediate neighbors
  - If there are no records with rating = 1, T1 must lock the index page where such a data entry would be, if it existed!
  - e.g., lock the page with rating = 0 and beginning of rating=2
  - Or lock pages for just one extra data item on one side, if a lock is understood to cover the key value plus gap to one side.
  - If there is no suitable index, T1 must lock all data pages, and lock the file to prevent new pages from being added

Index Locking

- Assume index on the rating field
  - Row locking is the industry standard now
  - T1 should lock all the data entries with rating = 1 and at least one neighbor (depending on details of protocol)
    - If there are no records with rating = 1, T1 must lock the entries adjacent to where data entry would be, if it existed!
    - e.g., lock the last entry with rating = 0 and beginning of rating=2
  - If there is no suitable index, T1 must lock all the rows and lock the file to prevent new rows from being added, or use a “table lock”.
Predicate Locking

- Grant lock on all records that satisfy some logical predicate
  - But note that a general predicate can depend on data in the row: $\text{salary} > 50000 + 1000 \times \text{years}$
  - Or a whole table: $\text{salary} > (\text{select} \ \text{avg(salary)} \ \text{in} \ \text{emps})$

- Index locking is a special case of predicate locking
  - Index supports efficient implementation of the predicate lock
  - Predicate is specified in WHERE clause

- In general, predicate locking is expensive to implement!
  - Can avoid the runtime cost by using Repeatable Read isolation level, but that opens up anomaly possibilities.

Index Locking, Blow by blow

- Index locking happens in the storage engine, based on FILE calls coming from query processor as directed by the query plan

  - Example: Transaction T1 accesses a heap table with certain index, gets row for certain index key value, say 100. Suppose the next data entry is for another key, 102.
  - Storage engine share locks the accessed data entry for key 100, guarding it and the gap between that key and the next key.
  - Then if another transaction T2 tries to change the row with key 100, can't get necessary X lock, waits. Same with key 101.
  - Original transaction T1 can ask for next key get 102.
  - But if another transaction updates row with key 102 (not guarded by T1's share lock), then T1 has to wait for the next key.

Index Locking Scenario, cont.

- There is an underlying assumption in that story: that all the accesses in fact use the index on this column.
- Well, the important thing is that all accesses that change the column value go through the index. It's OK for another reader to access the value.
- An insert or delete need to change the index, so they are naturally involved.
- An update to this column also needs to change the index, in two places, so it also collides with the old lock.
- You can see this has to be checked out carefully!

Locking vs. Latching

- To protect shared data in memory, multithreaded programs use mutex (semaphores)
  - API: enter_section/leave_section, or lock/unlock
  - Every Java object contains a mutex, for convenience of Java programming: underlies synchronized methods
  - Database people call mutexes and related mechanisms "latches"
  - Need background in multi-threaded programming to understand this topic fully

- The tree structure needs mutex/latch protection
  - Example: split node. No row data is changed, just the details in pages in the buffer pool. No i/o is needed (can't hold a latch across disk i/o without ruining performance.)
  - Latches can be provided by the same lock manager as does 2PL locking, and can have share and exclusive types like locks.
  - In these slides, will use "lock" in quotes to mean non-2PL lock/latch...

Locking for B+ Trees

- Naive solution
  - Ignore tree structure, just lock its pages following 2PL

- Very poor performance!
  - Root node (and many higher level nodes) become bottlenecks
  - Every tree access begins at the root!

- Not needed anyway!
  - Only row data needs 2PL (contents of tree)
  - Tree structure also needs protection from concurrent access
  - But only like other shared data of the server program
  - Note this modern view is not covered in book
  - See Graefe, A Survey of B-tree locking techniques (2010)
  - B-tree locking is a huge challenge!

Locking for B+ Trees (contd.)

- Searches
  - Higher levels only direct searches for leaf pages

- Insertions
  - Node on a path from root to modified leaf must be "locked" in X mode only if a split can propagate up to it
  - Similar point holds for deletions

- There are efficient locking protocols that keep the B-tree healthy under concurrent access, and support 2PL on rows
A Simple Tree Locking Algorithm:
("lock" here is really a latch on tree structure)

- **Search**
  - Start at root and descend: "crabbing down the tree"
  - Repeatedly get S "lock" for child then "unlock" parent, end up with S "lock" on leaf page
  - Later, caller is done with reading row, arranges release of S "lock"

- **Insert/Delete**
  - Start at root and descend, crabbing, obtaining X "locks" as needed
  - Once child is "locked", check if it is safe
  - If child is safe, release "lock" on parent, leaving X "lock" on child
  - Get 2PL X lock on place for new row/old row, insert/delete row, release "lock"

**Safe node:** not about to split or coalesce

- Inserts: Node is not full
- Deletes: Node is not half-empty

When control gets back to QP, transaction only has 2PL locks on rows

---

**Difference from text**

- The algorithm actions described in the text are valid, for example, crabbing down the tree, worrying about full nodes, etc.
- What's different is that the locks for index nodes are shorter lived than described in the text: only 2PL locks on rows are kept until end of transaction, not any locks on index nodes.
- Note that text uses locks and releases them before commit, a sign that they are not actually Strict 2PL locks.
- Note the admission on pg. 564 that the text's coverage on this topic is "not state of the art". Graefe's paper is.

---

**An Example**

```
A

   20
  /   |
35    B
 /     |
23    F
 |     |
 |     |
 |     |
E
```

**Do:**

- Search 38*
- Insert 45*
- Insert 25*
- Delete 38*

**Insert 45 case**

Grab down tree getting X "locks" (really latches)
- "Xlock" A
- B is safe, so "unXlock" A
- C is unsafe, so can't "unXlock" B now
- "Xlock" E (page of rows)
- E is safe, so "unXlock" C

Xlock row (2PL lock) for 45, copy out row or pin buffer, provide row pointer to caller
- "UnXlock" E

Return to QP with 2PL X lock on row with key 45 (or index entry and row)

---

**A Variation on Algorithms**

- **Search**
  - As before

- **Insert/Delete**
  - Set "locks" as if for search, get to leaf, and set 2PL X lock on leaf
  - If leaf is not safe, release all "locks", and restart using previous Insert/Delete protocol

This is what happens if the search down the tree happens on a page that is not in buffer—don’t want to hold a latch across a disk I/O (takes too long)

---

**Lock Management**

- Lock and unlock requests are handled by the lock manager (see Sec. 17.2.1)

  - **Lock table entry:**
    - Name of identifier
    - Number of transactions currently holding a lock
    - Type of lock held (shared or exclusive)
    - Pointer to queue of lock requests
  - Locking and unlocking have to be atomic operations (need mutex protection)
  - Lock table entries are kept in order, to prevent starvation (lots of reads preventing a writer from ever getting a lock, etc.)
Lock Manager Data structure: a multilist

- Need access to lock entry by lock name or transaction id
- Some of these transactions are blocked on the lock

Multiple-Granularity Locks

- Hard to decide what granularity to lock:
  - tuples vs. pages vs. files
  - Inefficient to have a million row locks to scan a relation
  - Shouldn’t have to decide once and for all!
- Data containers are nested:

  Database
  └── Files
      └── Pages
      └── Tuples

New Lock Modes, Protocol

- Allow transactions to lock at each level, but with a special protocol using new **intention locks**
- Before locking an item, must set intention locks on ancestors
- To lock an item with an S lock (X lock), need an IS (IX) lock or stronger on ancestors
- For unlock, go from specific to general (i.e., bottom-up).
- **SIX mode**: Like S & IX at the same time.

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>IS</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>IX</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>S</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>X</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>

New Lock Modes, Protocol

- Lock manager doesn’t care: just make up lock names with table name or item id, use new lock compatibility table
- Protocol makes client check higher level(s) first, then target level, so lock manager itself (or its kernel part) has no responsibility to know relationship between locks

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>--</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>IS</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>IX</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>S</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
<tr>
<td>X</td>
<td>v</td>
<td>v</td>
<td>v</td>
<td>v</td>
</tr>
</tbody>
</table>