Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones.
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol).
- When a transaction locks a node in the tree explicitly, it implicitly locks all the node's descendants in the same mode.
- Granularity of locking (level in tree where locking is done):
  - Fine granularity (lower in tree): high concurrency, high locking overhead
  - Coarse granularity (higher in tree): low locking overhead, low concurrency
Example of Granularity Hierarchy

The levels, starting from the coarsest (top) level are:
- database
- area
- file
- record

Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

<table>
<thead>
<tr>
<th></th>
<th>IS</th>
<th>IX</th>
<th>S</th>
<th>SIX</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>IX</td>
<td>true</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>S</td>
<td>true</td>
<td>false</td>
<td>true</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>SIX</td>
<td>true</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td>X</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
Multiple Granularity Locking Scheme

- Transaction $T_i$ can lock a node $Q$, using the following rules:
  1. The lock compatibility matrix must be observed.
  2. The root of the tree must be locked first, and may be locked in any mode.
  3. A node $Q$ can be locked by $T_i$ in S or IS mode only if the parent of $Q$ is currently locked by $T_i$ in either IX or IS mode.
  4. A node $Q$ can be locked by $T_i$ in X, SIX, or IX mode only if the parent of $Q$ is currently locked by $T_i$ in either IX or SIX mode.
  5. $T_i$ can lock a node only if it has not previously unlocked any node (that is, $T_i$ is two-phase).
  6. $T_i$ can unlock a node $Q$ only if none of the children of $Q$ are currently locked by $T_i$.

- Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.
- Lock granularity escalation: in case there are too many locks at a particular level, switch to higher granularity S or X lock.

Insert/Delete Operations and Predicate Reads

- Locking rules for insert/delete operations
  - An exclusive lock must be obtained on an item before it is deleted
  - A transaction that inserts a new tuple into the database I automatically given an X-mode lock on the tuple

- Ensures that
  - reads/writes conflict with deletes
  - Inserted tuple is not accessible by other transactions until the transaction that inserts the tuple commits
Phantom Phenomenon

- Example of **phantom phenomenon**.
  - A transaction T1 that performs **predicate read** (or scan) of a relation
    - `select count(*) from instructor where dept_name = 'Physics'`
  - and a transaction T2 that inserts a tuple while T1 is active but after predicate read
    - `insert into instructor values ('11111', 'Feynman', 'Physics', 94000)` (conceptually) conflict in spite of not accessing any tuple in common.
  - If only tuple locks are used, non-serializable schedules can result
    - E.g. the scan transaction does not see the new instructor, but may read some other tuple written by the update transaction
  - Can also occur with updates
    - E.g. update Wu’s department from Finance to Physics

Insert/Delete Operations and Predicate Reads

- **Another Example**: T1 and T2 both find maximum instructor ID in parallel, and create new instructors with ID = maximum ID + 1
  - Both instructors get same ID, not possible in serializable schedule

**Schedule**

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(instructor where dept_name='Physics')</td>
<td>Insert Instructor in Physics</td>
</tr>
<tr>
<td></td>
<td>Insert Instructor in Comp. Sci.</td>
</tr>
<tr>
<td></td>
<td>Commit</td>
</tr>
<tr>
<td>Read(instructor where dept_name='Comp. Sci.')</td>
<td></td>
</tr>
</tbody>
</table>
Handling Phantoms

- There is a conflict at the data level
  - The transaction performing predicate read or scanning the relation is reading information that indicates what tuples the relation contains.
  - The transaction inserting/deleting/updating a tuple updates the same information.
  - The conflict should be detected, e.g. by locking the information.

- One solution:
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item.
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item. (Note: locks on the data item do not conflict with locks on individual tuples.)

Above protocol provides very low concurrency for insertions/deletions.

Index Locking To Prevent Phantoms

- **Index locking protocol** to prevent phantoms
  - Requires that every relation must have at least one index.
  - A transaction can access tuples only after finding them through one or more indices on the relation.
  - A transaction $T_i$ that performs a lookup must lock all the index leaf nodes that it accesses, in S-mode
    - Even if the leaf node does not contain any tuple satisfying the index lookup (e.g. for a range query, no tuple in a leaf is in the range)
  - A transaction $T_i$ that inserts, updates or deletes a tuple $t_i$ in a relation $r$
    - Must update all indices to $r$
    - Must obtain exclusive locks on all index leaf nodes affected by the insert/update/delete
  - The rules of the two-phase locking protocol must be observed

Guarantees that phantom phenomenon won’t occur
Next-Key Locking to Prevent Phantoms

- Index-locking protocol to prevent phantoms locks entire leaf node
  - Can result in poor concurrency if there are many inserts

- **Next-key locking protocol**: provides higher concurrency
  - Lock all values that satisfy index lookup (match lookup value, or fall in lookup range)
  - Also lock next key value in index
    - even for inserts/deletes
  - Lock mode: S for lookups, X for insert/delete/update

- Ensures detection of query conflicts with inserts, deletes and updates
  
  Consider B+-tree leaf nodes as below, with query predicate $7 \leq X \leq 16$.

  Check what happens with next-key locking when inserting: (i) 15 and (ii) 7

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>8</td>
<td>11</td>
<td>14</td>
<td>18</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>24</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>38</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>55</td>
</tr>
</tbody>
</table>