Two Phase Locking, Lecture 3
(BHG, Chap. 3)

The practical foundation
Topics

- Part I: Foundations
- Part II: Beyond Basic Locking
- Part III: Degrees of Isolation
- Part IV: Viewing B+-tree Operations as Transactions
Part I: Foundations
Aggressive and Conservative Schedulers

- Scheduler handling an operation:
  - Send to DM immediately.
  - Delay.
  - Reject.
- Types of schedulers (fuzzy):
  - aggressive – tries not to delay.
  - conservative – delays operations (extreme – serial).
- High contention $\rightarrow$ favor conservative.
- Helpful to know readset and writeset (pre-declare).
- Difficult to do statically in practice (execution dependent).
Locking

- **Basic idea**: lock to protect operations.
- \( H = l1[x] \ r1[x] \ w1[x] \ u1[x] \ c1 \ l2[z] \ r2[z] \ u2[z] \ l2[x] \ w2[x] \ u2[x] \ c2 \)
- \( H' = l2[z] \ l1[x] \ r2[z] \ r1[x] \ w1[x] \ u1[x] \ c1 \ u2[z] \ l2[x] \ w2[x] \ u2[x] \ c2 \)
- Same as T1 T2
- \( H'' = l1[x] \ l2[z] \ r2[z] \ u2[z] \) (\( l2[x] \) can’t cont.)
- Seems to work …
H is not equivalent to a serial history.

**Reason** – release lock and then lock again.
Basic 2PL, notation

- \text{rli}[x], \text{wli}[x]: \text{read (write)} lock owned by Ti.
- \text{oli}[x]: denotes a locking operation.
- Two lock operations conflict: different transactions, same data item, at least one is write.
- We also use \text{rli}[x], \text{wli}[x] to denote operations to obtain such locks.
- \text{rui}[x], \text{wui}[x]: unlock operations.
Basic 2PL, rules

1. Upon pi[x]:
   1. pli[x] conflicts with qlj[x] that’s already set → delay
   2. otherwise, **atomically** set pli[x], send pi[x] to DM.

2. Do not release a lock pli[x] if the DM has not yet acknowledged performing it.

3. Once a lock for a transaction has been released, the transactions will not be assigned any additional locks.
   - May lead to **deadlocks**.
   - A transaction may undergo lock conversion (holds a read and now needs a write).
Correctness

- Characterize histories.
- Show these characteristics $\rightarrow$ SR.
  - We show $SG(H)$ is acyclic.
- We include lock and unlock ops in histories:
  - $oli[x] < oi[x]$ – need a lock before $oi$ on $x$.
  - $oi[x] < oui[x]$ – DM operates while lock is held.
Correctness: H produced by 2PL

- **Proposition 1**: \(oi[x]\) in \(C(H)\) →
  - \(oli[x]\) and \(oui[x]\) are in \(C(H)\)
  - \(oli[x] < oi[x] < oui[x]\)
- **Proof**: All committed transaction actions are in \(C(H)\). By rules (1) and (2).

- **Proposition 2**: \(pi[x]\) and \(qj[x]\) (\(i \neq j\)) are conflicting operations in \(C(H)\) →
  - Either, \(pui[x] < qlj[x]\),
  - or, \(quj[x] < pli[x]\).
- **Proof**: By rule (1), at any point, only one of the locks can be held. Scheduler must release one lock before awarding the other.
Correctness: H produced by 2PL

- **Proposition 3**: pi[x] and qi[y] in C(H) \(\Rightarrow\) 
  - pli[x] < qui[y]

- **Proof**: By the two phase rule (3) every lock op precedes all unlock ops.
**Correctness: H produced by 2PL**

- **Lemma 4**: \( T_i \rightarrow T_j \) in \( SG(H) \)
  - There is a data item \( x \) and operations \( p_i[x] \) and \( q_j[x] \) such that \( p_{ui}[x] < q_{lj}[x] \).

- **Proof**: \( T_i \rightarrow T_j \) implies there are conflicting operations \( p_i[x] \) and \( q_j[x] \) s.t. \( p_i[x] < q_j[x] \).
  - By Prop. 1:
    1. \( p_{li}[x] < p_i[x] < p_{ui}[x] \),
    2. \( q_{lj}[x] < q_j[x] < q_{uj}[x] \)
  - By Prop 2, there are two cases to consider:
    - \( p_{ui}[x] < q_{lj}[x] \) which is what we want to prove.
    - \( q_{uj}[x] < p_{li}[x] \). By \( 1+2 \): \( q_j[x] < p_i[x] \). Contradicts \( p_i[x] < q_j[x] \).
**Lemma 5**: $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n$, $n>1$, is a path in $\text{SG}(H)$

- There are data items $x$ and $y$, and operations $\text{pi}[x]$ and $\text{qn}[y]$ in $H$, such that $\text{pu}_i[x] < \text{ql}_n[y]$.

**Proof**: By induction on $n$.

- **Basis** ($n=2$): This is lemma 4.
- **Induction step**:
  - Suppose it holds for $n = k \not\leq 2$, we show it holds for $n = k + 1$.
  - $T_1 \rightarrow \ldots \rightarrow T_k \rightarrow x, z: \text{p}_1[x], \text{o}_k[z], \text{pu}_1[x] < \text{ol}_k[z]$.
  - $T_k \rightarrow T_{k+1} \rightarrow y, \text{o'}_k[y], \text{q}_{k+1}[y], \text{o'}u_k[y] < \text{ql}_{k+1}[y]$.
  - By prop. 3, $\text{ol}_k[z] < \text{o'}u_k[y]$. (two phase rule.)
  - So, $\text{pu}_1[x] < \text{ql}_{k+1}[y]$. 

---

(c) Oded Shmueli 2004
Correctness

- **Theorem 5**: Every 2PL history $H$ is SR.
- **Proof**: Suppose $SG(H)$ contains a cycle $T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \rightarrow T_1$, $n>1$. By lemma 5 there are data items $x$ and $y$ such that $T_1$ unlocked a lock on $x$ and later on obtained a lock on $y$. This contradicts proposition 3.
Deadlocks

- $wl1[x]$
- $wl2[y]$
- $rl1[z]$
- $r1[z]$
- $rl2[w]$
- $r2[w]$
- $rl1[y] \rightarrow \text{wait}$
- $rl2[x] \rightarrow \text{wait}$
Deadlock Resolution

- Timeout.
- WFG, Ti $\rightarrow$ Tj if Ti waits for a lock release by Tj.
- Break deadlock $\rightarrow$ abort transactions (victims).
- Factors: work invested, work remaining, cycles broken, repeated victim.
Deadlock Prevention

- Lock in a predetermined linear order.
- Restart a transaction on each collision.
- Priority scheme ➔ Cyclic restart …
- Timestamps on starting.
- Ti tries to obtain a lock held by Tj:
  - **Wait-Die** – favors young
    - TS(Ti) < TS(Tj) → Ti waits
    - TS(Ti) > TS(Tj) → restart Ti
    - Upon Ti restart, it may be restarted again
    - If T has all locks, it’ll not be aborted
  - **Wound-Wait** – favors old
    - TS(Ti) < TS(Tj) → restart Tj
    - TS(Ti) > TS(Tj) → Ti waits
    - Upon Tj restart it’ll wait
    - **Note**: on restart Tj, it might have terminated…

(c) Oded Shmueli 2004
2PL Variations: Static locking

- Ti obtains all locks prior to execution.
- Needs to predeclare its readset and writeset.
- Various possibilities:
  - If all locks can be granted → execute else put on queue. After each lock release check queue.
  - Alternatively, queue per data item, put Ti on all queues at once. Can start work before all locks are granted, Still, no deadlock.
Strict 2PL

- Release locks only after the DM acknowledges ai or ci.
  - Ensures currently locked data items will no longer be accessed and no new lock requests.
  - Guarantee a strict execution.
    - Suppose \( wi[x] < oj[x] \).
    - \( wli[x] < wi[x] < wui[x] \) and \( olj[x] < oj[x] < ouj[x] \)
    - \( wli[x] \) and \( olj[x] \) conflict.
      - Either \( wui[x] < olj[x] \)
      - or \( ouj[x] < wli[x] \) \( \rightarrow \) \( oj[x] < wi[x] \), contradiction.
      - So, \( wui[x] < olj[x] \).
    - Strict 2PL \( \rightarrow \) \( ai < wui[x] \) or \( ci < wui[x] \).
    - So, either \( ai < oj[x] \) or \( ci < oj[x] \).
Discussion

- The argument is not dependent on keeping read locks till after commit.
- In fact, these can be released subject to the 2PL rule.
- This means, release read locks as soon as the transaction terminates – issues a ci or an ai.
- Write locks remain until after the processing of ci or ai.
Locking Implementation

- Where should locks be placed: impractical on items, centrally.
- Hash table + semaphores on entries.
More on implementation

- Essential – quick lock release.
- Fairness – prevent indefinite postponement.
- Bottleneck – split table into sub-tables.
- Granularity may affect atomicity of read and write ops.
  - Two records updated on the same disk block.
  - Two copies are read into local memories, updated and written. One update gets lost.
  - **Solution**: obtain a short lock on the block, release that lock once the update is done to the block.
• T obtains a read lock on x
• Later on it needs a write lock

• may cause deadlock

<table>
<thead>
<tr>
<th>High priority</th>
<th>waiting for conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>r₁[x]</td>
<td>r₂[x]</td>
</tr>
<tr>
<td>w₁[x]</td>
<td>w₂[x]</td>
</tr>
</tbody>
</table>

Lock Conversion
Part II: Beyond Basic Locking
Phantoms

- Records may be added or deleted.
- Transactions may lock based on value.
  - Select * FROM STUDENT
  - WHERE DEPT = 'Computer Science'
- Does 2PL work in this setting?
Phantoms

- **T1:**
  - Lock Prof where dept = 'CS'
  - Add new faculty members.
  - Update salaries.

- **T2:**
  - Lock Prof where degree = 'Math Logic'
  - Update salaries.

Suppose there are no tuples satisfying
- dept = 'CS' \( \land \) degree = 'Math Logic'

But, what if there are?
Phantoms

- T1 locks
- T2 locks
- T1 adds < ‘CS’, ‘Jones’, ‘Math Logic’ >
- T2 adds < ‘CS’, ‘Smith’, ‘Math Logic’ >
- T1 adds 10% to CS members it knows about
- T2 subtracts 10% from Math Logic degree holders it knows about
- T1 ends
- T2 ends
- **Serializable?**
  - T1 T2: T2 should see Jones, but it does not
  - T2 T1: T1 should see Smith, but it does not
Solving the Phantom Problem

- The problem does not exist:
  - If you follow 2PL rules to the letter…

- Lock whole files.

- Predicate locks:
  - We’ll briefly discuss them (interesting but impractical).
  - Variant: precision locks (not covered).

- Granular Locks:
  - We’ll discuss them.
  - Essentially static predicate locks, covered next.
Granular Locks

- **Idea**: Each transaction should lock at the appropriate level:
  - T1 accesses a few records
  - T2 modifies many records in a file
  - T3 reorganizes the whole database

- **Main Benefit**: Less lock setting.

- **Secondary Benefit**: A predicate lock.

**Hierarchy:**

```
DATABASE
  AREA
  FILES
  RECORDS
```
Granular Locks

- Any object can be locked
- Read/Write Locks apply to descendants as well
Granular Locks

- **Explicit (read or write) lock**: held directly on the item.
- **Implicit (read or write) lock**: held on an ancestor.
- **Lock**: explicit or implicit.
- **Goal**: at no time should two transactions have conflicting locks on a node.
  - This guarantees that locks lock properly.
  - 2PL is still needed to ensure serializability.
  - To verify that locks lock properly it’s enough to consider leaves (bad locking is manifested there).
Granular Locks, locking conflicts

- IR-lock: intends to read descendant(s).
- IW-lock: intends to write descendant(s).
- IR and W conflict.
- IW and W conflict.
Granular Locks, locking rules

1. If v not root, to set IR-lock or R-lock, Ti must own IR or IW on v’s parent.
2. If v not root, to set IW-lock or W-lock, Ti must IW on v’s parent.

Above implies need to set intention locks all the way to the root.

3. To read (write) a node v, Ti must own (implicitly or explicitly) a read (write) lock on v.

4. Ti cannot release IR or IW on v if it currently holds a lock on any child of v.
   - ➔ lock release in leaf to root order.
Granular Locks, correctness

- To verify that locks lock properly it’s enough to consider leaves (bad locking is manifested there).
- Possible cases for two transactions:
  1. implicit R – explicit W
  2. implicit R – implicit W
  3. explicit R – explicit W - impossible
  4. explicit R – implicit W
  5. implicit W – explicit W
  6. implicit W – implicit W
  7. explicit W – explicit W - impossible
Granular Locks, correctness, case 1

implicit R \rightarrow explicit W, conflict at \( v \)

- T1 owns an R-lock on an ancestor \( y \) of \( v \).
- T2 owns an IW-lock on every ancestor of \( v \).
- T2 owns an IW-lock on \( y \). IW and R conflict!
Granular Locks, correctness, case 2

**implicit R – implicit W, conflict at v**

- T1 owns an R-lock on an ancestor y of v.
- T2 owns a W-lock on some ancestor z of v.
  a. $y = z \implies$ R and W concurrently held.
  b. y is an ancestor of z: T2 owns IW-locks on ancestors of z, and hence on y. IW conflicts with R.
  c. z is an ancestor of y: T1 owns IR on z and T2 owns a W on z. W conflicts with IR.
Granular Locks, RIW locks

- Ti would like to read a whole sub-tree and update a few records:
  - W-lock the whole subtree $\Rightarrow$ less concurrency.
  - Lock sub-tree root with IW, lock lower nodes appropriately with R, IR, IW.
  - Obtain RIW = R + IW
    - conflicts with R, W, IW, RIW.
    - does not conflict with IR.
Granular Locks, RIW locks, state

\[ F <T_1: RIW>, <T_2: IR> \]

\[ R_1 <T_2: IR> \]
\[ R_2 \]
\[ R_3 <T_1: W> \]
Update Lock

- Not symmetric.
- Motivation – reduce deadlocks due to conversions.
- Update and IR conflict to prevent deadlocks due to converting IR to R.
Granular Locks, RIW locks, conflicts

<table>
<thead>
<tr>
<th>Request</th>
<th>W</th>
<th>R</th>
<th>IR</th>
<th>IW</th>
<th>RIW</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>R</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>IR</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>IW</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>RIW</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>Update</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

(c) Oded Shmueli 2004
Update Locks – breaking symmetry
Granular Locks, practical issues

- Transaction may ask for additional/different locks.

- Lock conversion
  - if Ti owns pli[x] and requests qli[x] → least lock stronger than both.
  - Compile a lock conversion table (next slide).
  - Use at run-time.
  - For simplicity – a transaction holds a single lock.
  - The granted group is associated with a single lock as well.
Granular Locks, lock conversion table

<table>
<thead>
<tr>
<th>request</th>
<th>W</th>
<th>R</th>
<th>IR</th>
<th>IW</th>
<th>RIW</th>
<th>Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>w</td>
<td>w</td>
<td>w</td>
<td>w</td>
<td>w</td>
<td>w</td>
</tr>
<tr>
<td>R</td>
<td>w</td>
<td>r</td>
<td>r</td>
<td>riw</td>
<td>riw</td>
<td>u</td>
</tr>
<tr>
<td>IR</td>
<td>w</td>
<td>r</td>
<td>ir</td>
<td>iw</td>
<td>riw</td>
<td>u</td>
</tr>
<tr>
<td>IW</td>
<td>w</td>
<td>riw</td>
<td>iw</td>
<td>iw</td>
<td>riw</td>
<td>w</td>
</tr>
<tr>
<td>RIW</td>
<td>w</td>
<td>riw</td>
<td>riw</td>
<td>riw</td>
<td>riw</td>
<td>w</td>
</tr>
<tr>
<td>Update</td>
<td>w</td>
<td>u</td>
<td>u</td>
<td>w</td>
<td>w</td>
<td>u</td>
</tr>
</tbody>
</table>
Granular Locks, lock escalation

- If it can be determined that a transaction will access many records → coarse level.
- Can use past behavior.
- Real time lock escalation. Once “many records” are requested, request a coarser lock.
- Practically, sometimes it’s better to **abort** and restart at a higher granularity.
Rooted DAG Locking

- DAG – directed graph with no cycles.
- Source – vertex with no incoming edges.
- Rooted DAG – A DAG with a unique source.

Idea: “protect” all ways of accessing a record.
Rooted DAG Locking, example

- Index for Tax
  - Index entries A-F
  - Implicit R and W

- Index for Accounts
  - Index entries G-P
  - Index entries Q-Z

- Accounts File
  - Implicit W

- Tax File

- DB
  - Implicit IR
  - Implicit IW

- Area 1
  - Implicit IR
  - Implicit IW

- Area 2
  - Implicit W

- Jerusalem Record
- Haifa Record

(c) Oded Shmueli 2004
Rooted DAG Locking, rules for non-root x

- **Locks:**
  - To set rli[x] or irli[x] – Ti must have at least ir-lock or iw-lock on some parent of x.
  - To set wli[x] or iwli[x] – Ti must have at least iw-lock on all parents of x.

- **Operations:**
  - To *read* x Ti must own an r-lock or a w-lock on some ancestor of x.
  - To *write* x, for every root-to-x path Ti must own a w-lock on some ancestor of x on that path (cut-set).
  - Ti may not release a lock on x while holding a lock on a child of x.
Rooted DAG Locking, correctness

- \( \text{locksT} : \text{Nodes} \rightarrow \text{LockTypes} \) – formalizes explicitly set locks.
- \( \text{implicitlocksT}(b) = w \) if \( b \) is root and \( T \) holds a \( w \) lock on \( b \), or \( b \) is not root and \( \text{locksT}(bi) = w \) for each node \( bi \) in some node cut-set for \( b \).
- \( \text{implicitlocksT}(b) = r \) if the previous condition is false and \( \text{locksT}(p) \) \( \{r, riw, u, w\} \), for some ancestor \( p \) of \( b \).
- **Explicit Lock Graph** – lock labels from \( \text{locksT} \).
- **Implicit Lock Graph** – lock labels from \( \text{implicitlocksT} \).
- Two explicit lock graphs are compatible if for all nodes \( b \) \( \text{lockTable}(\text{lockT}(b), \text{lockT}'(b)) = y \).
- Two implicit lock graphs are compatible if for all nodes \( b \) \( \text{lockTable}(\text{implicitlocksT}(b), \text{implicitlocksT}'(b)) = y \).
Rooted DAG Locking, Theorem

- **Theorem:** If two explicit lock graphs are compatible then their implicit lock graphs are compatible.

- **Proof:**
  - Let L and L’ be two compatible explicit lock graphs.
  - Suppose their corresponding implicit lock graphs are incompatible at node b.
  - There are two cases to consider.
  - In both cases, we derive a contradiction.
Contradiction since $T'$ has non-null lock.

**Case 1**

$T'$ has riw, r, u or w

$T$ has w on all in cut-set

**Case 2**

$T'$ has riw, r, u or w

$T$ has w or riw or iw

$T$ has w on all in cut-set
Dynamic DAGs

- A record may be updated in an indexed field.
- It needs to be moved to another index interval.
- Need lock on both “old” and “new” interval.
Back to Phantoms

- Suppose we have a DAG representing files, indexes and records.
- Suppose all accesses to records are done using the DAG structure.
- Then, phantoms can be avoided.
- Insert and delete are treated as a write operations.
- **Observations**: there is no need to physically implement indexes. It’s sufficient that each transaction locks the index intervals containing all records the transaction is accessing.
Using DAGs to Avoid Phantoms (static indexes)

- SAL = SAL * 1.1 where DEG = 'math'
- Lock M-Z at least with iw
- Scan records, lock appropriate ranges in I_SAL (source and target).

- SAL = SAL * 1.1 where CITY = 'Haifa'
- Lock A-L
- Scan records, lock appropriate ranges in I_SAL (source and target).
- Same for M-Z.
Using DAGs to Avoid Phantoms (key range locking)

We’ll need **careful navigation**:  
• Use semaphores  
• Lock a leaf  
• Locate next leaf and lock it  
• Release lock on previous leaf.

• Need to avoid phantoms  
• Need to preserve tree’s structure under concurrent modifications

Diagram:

- Attribute value tuple id or key
- Legend:
Using DAGs to Avoid Phantoms (key range locking)

- **Ops:** exact match, range query, insert, delete.
- Locking \( v \) locks all records in \([v, w)\) where \( w \) is the next higher value in the index after \( v \).
- For range query on \( c_1 \leq \text{index}_\text{value} \leq c_2 \):
  - Navigate to relevant leaf. Obtain R semaphore.
  - R-lock the intervals \([c_0, w_1), \ldots, [w_{n-1}, w_n), [w_n, c_3)\) s.t. \( c_0 \) is max less equal to \( c_1 \) and \( w_n \) is max less equal to \( c_2 \).
  - Use long locks, navigate carefully between leaves.
  - Exact match is also viewed as a range query as there may be many records with same value \( c \).
Using DAGs to Avoid Phantoms (key range locking)

- **Idea**: lock old and new intervals, *w-lock* the inserted/deleted records!

- **Insert** a new record with value $c$:
  - Navigate to relevant *leaf*. Obtain $W$ semaphore.
  - Make sure others cannot read the interval containing $c$ by placing an *iw* lock on the interval $c_1$ s.t. $c_1$ is max less equal to $c$.
  - If a new interval is formed, protect it with a *w* lock, otherwise the iw lock on the old interval is sufficient.
  - Lock the inserted record itself with a *w* lock.
  - **Optimization**: a short iw lock suffices if $c \neq c_1$. This checks that no readers are currently seeing the soon to be split interval.
Using DAGs to Avoid Phantoms (key range locking)

Delete a record with value c:

- Navigate to relevant leaf. Obtain W semaphore.
- Make sure others cannot read the interval containing c by placing a w lock on the interval c1 s.t. c1 is the entry immediately to the left (next-lower entry).
- That is, protect the old interval from reading with a w lock.
- Protect the c interval with a w lock.
- Lock the deleted record itself with a w lock.
- May need to navigate through leaves until actual record is found.
- **Optimization**: if \( c \neq c1 \) then a long iw lock on c1 suffices. It will allow for concurrent updates.
Using DAGs to Avoid Phantoms (key range locking)

- Retrieve $17 \leq k \leq 23$, r lock 16, 20
- Insert (21,K8), iw lock 20, w lock 21
- Insert (20,K8), iw lock 20
- Delete (20,K4), w lock 20
- Delete (16,K2), w lock 14, w lock 16
Predicate Locks

- \((R, P, a)\) – **predicate lock**
  - \(R\) a relation, \(P\) predicate, \(a \in \{r, w\}\)
  - Example: \((\text{Prof.}, \text{dept} = \text{‘cs’}, w)\)

- \((R, t, a)\) – **action**
  - \(R\) a relation, \(a \in \{r, w\}\), \(t\) over \(R\)
  - Example: \((\text{Prof.}, < \text{Smith, 26, 30000, cs }>, w )\)

- **Operations**
  - \((R, t, w)\): **Insert** \(t\) into \(R\)
  - \((R, t, w)\): **Delete** \(t\) from \(R\)
  - \((R, t, w)\) and \((R, t’, w)\): **Replace** \(t\) with \(t’\) in \(R\)
Predicate Locks, covering and conflicts

- A predicate lock **covers** an action:
  - \(( R', P', a' ) \) **covers** \(( R, t, a ) \) if
    \[( a' = w \lor a = r ) \land P'(t) \land (R = R') \]
  - Intuitively, this lock allows to perform the action.

- An action **conflicts** with a predicate lock:
  - \(( R', P', a' ) \) **conflicts** with \(( R, t, a ) \) if
    \[( a' = w \lor a = W) \land P'(t) \land (R = R') \]
  - Intuitively, if T1 holds lock and T2 performs action, there is a conflict.
Predicate Locks, lock conflicts

Two predicate locks conflict if there is an action covered by one and conflicting with the other:

- \(( R', P', a' )\) and \(( R, P, a )\) conflict if
  \[(R = R') \land (a' = w \lor a = w) \land (\otimes t) [ P(t) \land P'(t) ]\]
Predicate Locks, mode of operation

- Award a lock only if it does not conflict with existing locks.
- An action is allowed only if it is covered by a lock owned by its issuing transaction.
- **Access via a predicate** $P'$ by a transaction $T$ is allowed if:
  - $T$ owns a relevant lock $(R, P, a)$, and
  - $\forall$ tuples $t$, if $P'(t) = \text{true}$ then $P(t) = \text{true}$, i.e., $P' \rightarrow P$.
  - That is, $\exists t: P'(t) \land \neg P(t)$ is unsatisfiable.
Predicate Locks, checking lock conflicts

- **Problem**: Is there a tuple \( t \) s.t. \( P(t) \land P'(t) \).
  - In general, **undecidable**.
  - Also, fairly pessimistic.
  - Need to consider integrity constraints.
  - **Precision locks** help here. Check for *actual* conflicts.

- **Simple sentence** - a comparison.
  - \((sal > 100), (dept = 'cs')\)

- **Simple predicate** - Boolean combination of simple sentences:
  - \(((\text{rank} = 'FP') \land (sal > 27000)) \lor (dept = 'cs')\)

- **To check**:
  - Transform to DNF, of the form
    \((P11 \land P12 \land \ldots) \lor (P21 \land P22 \land \ldots) \ldots \lor (P_{k1} \land \ldots)\).
  - Check each disjunct ( ….. ).

- The problem is NPC for simple predicates.
- Not considered a realistic locking mechanism (inner loop).
- However, granular locks are in essence predicate locks!
Part III: Degrees of Isolation (Consistency)

- Serializability is not always needed (e.g., a statistical query).
- **Idea**: fit each transaction with its own degree of consistency and make sure it does not harm other transactions.
- **Problem**: consistency is not ensured.
- In many systems 02 is the default.
- **Cursor stability**: A variation. keep an r lock on currently scanned record.
Degrees of Isolation, definition

- Degree 0: short w locks.
- Degree 1: long w locks, recoverable.
- Degree 2: short r locks (will not read “dirty” data), long w locks.
- Degree 3: long r locks and long w locks.
Degrees of Isolation, concepts

REPORT  SCREEN  MEAN  SD

E1

:  

E10  

} readings

T0 (0): circular updates, short w locks

T1 (1): calculate approx. MEAN and SD. Dirty reads. Long w locks. May see “garbage” (ignores).

T2 (2): read the MEAN. Print to screen. Short r locks. Sees “good” data. Long w locks.

T3 (3): Write “consistent” MEAN and SD in a report. Long r locks. Long w locks. Has repeatable reads.
## Degrees of Isolation and SQL

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Isolation Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>$0^0 = 1^0$</td>
</tr>
<tr>
<td></td>
<td>Allowed for read only transactions. <em>Browse</em>.</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>$2^0$ approx. <em>cursor stability</em>.</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>$2.9999^0$, Almost $3^0$ without phantom protection. <em>Repeatable reads</em>.</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>$3^0$</td>
</tr>
</tbody>
</table>
Part IV: Viewing B+-tree Operations as Transactions

- Must first understand the tree locking protocol (not covered here).
B+-Tree Locking, reminder

- Operations of interest: Search, Insert, Delete
  - a pointer b pair implies that pointer points all keys k s.t. a ≤ k < b
  - Navigation for search.
  - Insert into leaf, if there is space.
  - Split leaf and insert minimum key of new leaf into parent, repeat recursively if needed.
  - Delete from leaf. If “too small” may merge leaf nodes. May repeat recursively
B+-Tree Locking, problem

- **The problem**: implement insert, delete and search as transactions:
  - We really care about the leaves level.
  - Non-leaves levels – redundant search structure.
  - We do not care about the search structure itself, it’s extra data; we still need it to be a consistent data structure.

- These transactions may be sub-transactions of more complex transactions.

- 2PL is ineffective – root is always locked.

- Can use tree locking, problems with insert – how long should a lock be held depends on number of recursive inserts.
B+-Tree Locking, algorithm

- When “going down” through node N, ask for a write lock on N.
  - read N.
  - If N is not full, it will not be split. Release the write locks held on any ancestors of N.
  - Upon reaching the leaf, insertion may proceed safely, all write locks are held at that point.

- This solution – too many delays due to write locks.
B+-Tree Locking, lock conversion

- When “going down” through node N, ask for a read lock on N.
- Set a write lock on leaf L.
- If L is full convert necessary read locks into write locks.
- Start at the node closest to the root that needs a write lock.
- Proceed down the tree, converting read locks into write locks.
- Can lead to deadlock. Solution:
  - Introduce new lock type: might write.
  - It conflicts with write and might write. No deadlock.
B+-Tree Locking, using links

- `link(N)`: pointer to right sibling.
- If no right sibling, point to first child of right sibling of parent.
- If parent has no sibling, point to first grandchild of right sibling of grandparent .
- End result: linked in key order at each level.
B+-Tree Locking, using links

- **Insert**: if no space, lock node, split content, “fit” into chain, release lock, insert pointer into parent...

- **Search** – proceed down, no need for lock coupling:
  - Obtain read lock on N
  - Determine child to continue, C
  - Release the lock
  - Proceed to C

- **Problem**: an insert in the meantime updates C and N:
  - It follows the search w.r.t. node N.
  - It precedes it w.r.t. node C.
  - So, it seems not serializable.

- **Idea**: use link if C no longer holds expected keys.

No deadlock.
B+-Tree Locking: What Could Go Wrong?

- **t=5**: 60 Deleted
- **t=4**: D
- **t=3**: T1
- **t=2**: 25
- **t=1**: T2
- **t=0**: 10 20 30 40 50 60 70

(file records)