Granularity of Locks

- Locking granularity is the size of the data item being locked.
  
  Example:
  - page
  - file
  - tuple (record)
  - field in a tuple
  - a particular field of all tuples (column)

- The granularity of locks is unimportant w.r.t. correctness, but it is important w.r.t. performance.

Granularity And Atomicity Of Reads And Writes

Assume that

- Read/Write is done by blocks
- Locking granularity is record, and
- Block b contains three records r1, r2, r3.
Granularity And Atomicity Of Reads And Writes

The granularity of locking must be at least as coarse as the granularity of the atomic read and write.

- Place another lock on block while read or write is performed; release it when operation completes (not according to 2PL rule).
- Use Multi-Granularity Locking.

Multi-Granularity Locking

- Define a hierarchy of granules where lower level granules are finer:

Database
  | Areas
  | | Files
  | | | Records

An instance of this hierarchy might be:

- Database
- Areas
- Files
- Records
- Blocks
Explicit, Implicit, And Intention Locks

- A lock on a granule \( x \), **explicitly** locks \( x \), and **implicitly** all its descendants in the same mode.

- If \( T_i \) wants to lock a record, say R1.1, all R1.1’s ancestors must be checked for a lock; R1.1 may be implicitly locked.
  - If implicit locking is not available, a transaction \( T_i \) that locks coarse granules should also lock all descendants.
  - This defeats the purpose of introducing multiple granules!

Why?

Explicit, Implicit, And Intention Locks

- An **intention** lock on an item \( x \) means that a transaction performs some operation on a descendant of \( x \).
  - What is the need for intention locks?

- The operation may be determined by the type (mode of the intention lock):
  - irl (intention to read lock)
  - iwl (intention to write lock)
  - riwl (read intention to write lock)

Multi-Granularity 2PL Protocol

<table>
<thead>
<tr>
<th></th>
<th>r</th>
<th>w</th>
<th>ir</th>
<th>iw</th>
<th>riw</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>w</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>ir</td>
<td>y</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>iw</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>y</td>
<td>n</td>
</tr>
<tr>
<td>riw</td>
<td>n</td>
<td>n</td>
<td>y</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

Growing Phase (top down manner)

- The root of hierarchy must be locked first.
- To set rl(x) or irl(x), \( T_i \) must have an irl or iwl on x’s parent.
- To set wl(x) or iwl(x), \( T_i \) must have an iwl on x’s parent.
- To read (write) x, \( T_i \) must have an rl (wl) on x or one of its ancestors (i.e., must be implicitly or explicitly locked).

Shrinking Phase (bottom up manner)

- \( T_i \) cannot release a lock on x if it holds a lock on any of x’s children.
- Once \( T_i \) unlocks at item, it cannot request another lock on any item.
Implementing MGL

To r(l(x)) (or w(l(x))), we must first r(l) (or w(l)) all of x’s ancestors

Who does this?
Who knows the granularity hierarchy in a system?
- How about the Lock Manager?
- How about application programmers?

Scheduler?
- It predicts the need for coarse granularity locks based on the transaction’s recent behavior
- It uses lock escalation.

In the system, where queries are compiled, the compiler may also generate coarse grain requests.

Lock Escalation

Transactions start locking at fine granularity.

When the number of lock requests exceeds a threshold, the scheduler (or TM) may do one of the following:
- Escalate the granularity of the transaction’s lock requests.
  - Escalating lock requests from level $l_k$ to level $l_k - 1$ implies a lock conversion on level $l_k - 1$.
  - Restart the transaction, this time setting coarser grain locks.

<table>
<thead>
<tr>
<th>Old Lock</th>
<th>Requested</th>
<th>Lock</th>
</tr>
</thead>
<tbody>
<tr>
<td>ir, iw, r, riw, w</td>
<td>ir, iw, r, riw, w</td>
<td>r, r, riw, riw, riw, riw, w</td>
</tr>
</tbody>
</table>

Strength: $w > riw > r > iw > ir$
**Timestamp Ordering**

- The basic idea:
  - Each transaction $T_i$ has a timestamp $ts(T_i)$.
  - If the scheduler receives an operation by $T_i$ and it has already processed a conflicting operation by $T_j$ and $ts(T_j) < ts(T_i)$ then $T_i$ is aborted.
  - When a transaction aborts, it must restart with a new (i.e. larger) timestamp.

**Max Read/Write Timestamps**

- To decide whether an operation is in timestamp order, we associate two values with each data item $x$:
  - $\text{max-rts}(x)$: the max $ts$ of transactions that performed a Read on $x$.
    - If $ts(T_i) = \text{max-rts}(x)$ then $T_i$ is the youngest transaction that has read $X$ successfully.
  - $\text{max-wts}(x)$: the max $ts$ of transactions that performed a Write on $x$.
    - If $ts(T_i) = \text{max-wts}(x)$ then $T_i$ is the youngest transaction that has written $X$ successfully.

**Read/Write in Basic TO**

- $\text{Read}(x)$
  - if $ts(T_i) < \text{max-wts}(x)$ then Abort $T_i$
  - else send $R(x)$ to DM;
  - $\text{max-rts}(x) = \max(\text{max-rts}(x), ts(T_i))$
  - endif;

- $\text{Write}(x)$
  - if $ts(T_i) < \text{max-rts}(x)$ or $ts(T_i) < \text{max-wts}(x)$ then Abort $T_i$
  - Else send $W(x)$ to DM;
  - $\text{max-wts}(x) = ts(T_i)$
  - endif
Timestamp Table

- These rules assume that each operation runs to completion before the next one is submitted to DM.
- For example, \( S:W_i(x)R_j(x) \), with \( ts(T_i) < ts(T_j) \) is a legal TO schedule.
- However, when the scheduler sends \( R_j(x) \) to DM, it must know that \( W_i(x) \) is finished.
- Thus, we need
  - \( r\text{-in-progress}(x) \): number of transactions reading \( x \)
  - \( w\text{-in-progress}(x) \): number of transactions writing \( x \) (0 or 1)
  - \( \text{waiting-list}(x) \): transactions waiting to access \( x \).

This information is stored in the *timestamp table*.

<table>
<thead>
<tr>
<th>data item</th>
<th>max-rts</th>
<th>max-wts</th>
<th>r-in-progress</th>
<th>w-in-progress</th>
<th>waiting-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>10</td>
<td>4</td>
<td>2</td>
<td>0</td>
<td>( w_1, w_2 )</td>
</tr>
<tr>
<td>y</td>
<td>11</td>
<td>12</td>
<td>0</td>
<td>1</td>
<td>( r_{20}, w_1 )</td>
</tr>
</tbody>
</table>
**Example**

### Admission Scheduling to DM

<table>
<thead>
<tr>
<th>max-rts</th>
<th>max-wts</th>
<th>r-in-progress</th>
<th>w-in-progress</th>
<th>waiting-list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initially</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>R_1(x)</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>R_2(x)</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>W_3(x)</td>
<td>Abort T_2 (because ts(T_2) &lt; max-rts)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W_4(x)</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>R_5(x)</td>
<td>6</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>ack(R_5(x))</td>
<td>6</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ack(R_6(x))</td>
<td>6</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Example**

### max-rts max-wts r-in-progress w-in-progress waiting-list

| R_3(x)  | 6 | 0 | 1 | 0 | W_7, R_8 |
| R_3(x)  | 6 | 0 | 0 | 0 | W_7, R_8 |
| R_3(x)  | 6 | 7 | 0 | 1 | R_8, R_9 |
| R_3(x)  | 6 | 7 | 0 | 1 | R_8, R_9 |
| R_3(x)  | 6 | 7 | 0 | 1 | R_8, R_9 |

**Basic TO and Recovery**

- Basic TO is not strict or ACA
  - does not prohibit overwriting of uncommitted data.
  - We must somehow delay \( W_i(x) \) if \( x \) was previously written by \( T_j \) until \( T_j \) terminates.
  - If we do not want cascading aborts we must also delay read operations on uncommitted data.

- **Solution**
  - The scheduler sets \( \text{w-in-progress} \) to 1 when a \( T_i \) starts the write operation on some \( x \).
  - It resets \( \text{w-in-progress} \) to 0 when \( T_i \) terminates and not when \( T_i \) finishes writing on \( x \).

**Thomas’ Write Rule**

- Consider transactions \( T_1, T_2, \) and \( T_3 \) where \( ts(T_i) = i \).
  - Assume the scheduler has already processed the following sequence of operations:
    \( W_i(x)W_j(x) \)
  - According to basic TO, if the scheduler receives \( W_j(x) \), \( T_j \) should abort.

- **TWR says** …
  - No problem, simply ignore \( T_j’s \) write operation;
    - send an ack that \( W_j(x) \) is successfully performed.
  - What matters is that the last write operation on \( x \) was performed by the transaction with the maximum \( ts \).
Read Operations and TWR

- Assume transactions $T_1$, $T_2$, $T_3$, $T_4$, and $T_5$ and that the scheduler has already received these operations:
  
  $W_1(x)R_3(x)W_5(x)$

- If the scheduler receives $W_4(x)$, could this operation be ignored?
  - Yes. It is like executing: $W_1(x)R_3(x)W_4(x)W_5(x)$

- If the scheduler receives $W_2(x)$, could this operation be ignored?
  - No. The correct schedule would be: $W_1(x)W_2(x)R_3(x)W_5(x)$
    but that's impossible, because $T_2$ already read the write of $T_1$. So $W_2(x)$ should be rejected.

TO With TWR

- **Write$_i(x)$:**
  - if $ts(T_i) < max$-rts$(x)$ then abort $T_i$
  - else if $ts(T_i) < max$-wts$(x)$ then ignore $W_i(x)$ (i.e., assume it is done)
  - else if $w$-in-progress$(x) = 0$ and $r$-in-progress$(x) = 0$
    then send $W_i(x)$ to DM
    max$-$wts$(x) = ts(T_i)$
    $w$-in-progress$(x) = 1$
  - else
    insert $W_i$ to waiting-list$(x)$ in timestamp order
  end if

- **Read$_i(x)$:** Same as in Basic TO

Based on the Oldest Transaction

- The scheduler keeps the timestamp of the oldest active transaction $T_{\text{oldest}}$
  - When the table becomes too long, the scheduler removes all $x$ for which max-rts$(x) < ts(T_{\text{oldest}})$ and max-wts$(x) < ts(T_{\text{oldest}})$
  - In this case, we are certain that no transaction should abort when it tries to access a data item which is not in the table.

Timestamp Table Management

- To process an operation on $x$, we need timestamp information for $x$ (for every $x$). Thus, the timestamp table may become too long.

- The solution can be based on the following idea:
  - The scheduler can delete all $x$ for which it can be sure that it will not receive operations on $x$ from a transaction whose $ts$ is less than max-wts$(x)$.
  - Two solutions
    - Based on the $ts$ of the oldest active transaction.
    - Based on timeout.
**Timeout**

- Assume TM uses a real time clock to generate timestamps. Then at a given time $t$, we are almost sure that no transaction is active in the system with a timestamp less than $t-\delta$.

- The scheduler periodically does the following:
  - It sets $t_{min}$ to be $t-\delta$.
  - It removes from the timestamp table all $x$ for which max-rts and max-wts are less than $t_{min}$.
  - It marks the table with $t_{min}$.

- Now, to process some operation on $x$, the scheduler must proceed as follows:
  - If $x$ exists in the table proceed as usual.
  - If $x$ is not in the table and $ts(T) \geq t_{min}$ add $x$ to the table and proceed as usual.
  - If $x$ is not in the table and $ts(T) < t_{min}$ abort $T_i$.

**TO Versus 2PL**

In the following, assume that $ts(T_i) = i$.

- In 2PL, a transaction is never aborted because it submitted an operation too late; it simply waits.
- **Example**: the scheduler receives the following requests $R_2(x)C_2W_1(x)C_1$.
  - In TO, $T_1$ must abort $T_2$ submits $W_i(x)$ too late.
  - In 2PL, it is a legal sequence of operations.
Multi-version Concurrency Control

Assume the following sequence of events.
\[ W_2(x) C_0 W_2(x) R_2(x) C_2 C_1 \]

This sequence CANNOT be produced by a strict 2PL, or Timestamp-Ordering, because
- **Strict 2PL**
  - \( T_1 \) can not read lock \( x \) until after \( C_2 \).
- **TO**
  - Since \( ts(T_1) < ts(T_2) \), \( T_1 \) should abort when it tries to \( R_1(x) \).

An Idea!!
- If we had kept the old version of \( x \) when \( W_2(x) \), then we could avoid having to delay \( T_1 \) in (2PL) or abort \( T_1 \) (in TO) by having \( T_1 \) read the before image of \( x \)
- **Disadvantages?**
  - Complexity
  - Storage space

Basic Idea
- The DM keeps a list of versions for each \( x \).
  - Version \( x_i \) means the version of \( x \) produced by a Write on \( x \) by transaction \( T_i \).
- When the scheduler receives a \( W_i(x) \), it sends a \( W(x) \) to DM. Each \( Write(x) \) produces a new version of \( x \).
- When the scheduler receives a \( R_i(x) \), it must decide when to send the operation to DM and which version of \( x \) to read. A Read operation to the DM will be of the form \( R_i(x) \).
- If a transaction \( T \) is aborted, any version it created is destroyed.
Basic Idea

- **Example:** Assume the scheduler receives:
  \[ W_0(x) \ C_0(x) \ W_2(x) \ R_1(x) \ C_2 \ C_1 \]
  The scheduler sends to DM the following operations:
  \[ W_0(x_0) \ C_0 \ W_2(x_2) \ R_1(x_0) \ C_2 \ C_1 \]
  The above is a legal schedule in both types of schedulers: strict 2PL, TO.

Visibility of Versions

- Versions are under the absolute control of the scheduler and data manager.
  - Users (transactions) still reference data items as usual not by versions.
  - In applications where versions of \( x \) do exist, each version of \( x \) must be considered as an individual item.

- **One-copy serializability (1SR)** is the correctness criterion for Multiversion Concurrency Control.
  - 1SR requires that transaction executions are equivalent to a serial execution of those transactions on a one-copy database.

Alternatives for Storing Multiple Versions

- **Horizontal Redundancy**
  - Extend database schema horizontally
    - Extra "instances" of fields that change
    - 2VNL (2VNL/k)

- **Vertical Redundancy**
  - Extend database schema vertically
    - Extra tuples with modified fields
    - MVNL

- [additional material on the web page]
Multi-version Timestamp Ordering

Each transaction \( T_i \) has a unique timestamp \( ts(T_i) \).

Each version of \( x \) is labeled with the timestamp of the transaction that wrote \( x \).

The scheduler translates operations on data items into operations on versions of these data items.

Scheduling Operations

\( R(x) \)
- Find \( x_k \), the version of \( x \) where \( T_k \) has the largest timestamp less than or equal to \( ts(T_i) \).
- Send \( R(x_k) \) to DM.
  - Therefore, a Read operation is never delayed or rejected.

\( W(x) \)
- If an operation \( R(x_k) \), where \( ts(T_i) \leq ts(T_k) \), has already been processed then reject \( W(x) \), and restart \( T_i \).
- Otherwise, send \( W(x) \) to DM.
  - Write operations may abort.

Can we avoid cascading aborts altogether by using the write-in-progress bit?
Deleting Old Versions

- The scheduler must delete versions from the oldest to the newest.
  - Keep the smallest timestamp, $t_{s,\text{top}}$, of all currently active transactions (i.e., the timestamp of the oldest active transaction).
  - When the oldest transaction $T_i$ terminates, find the most recent $x_i$ such that
    - $k \leq t_s(T_i)$, and
    - $x_i$ is not the most recent version of $x$.
  - Delete all committed $x_j$ for which $j < k$.

Deleting Old Versions

- Example: Assume
  Versions: $x_1$, $x_4$, $x_5$, $x_12$, $x_{20}$
  Active transactions: $T_6$, $T_{10}$, $T_{12}$, $T_{14}$
  If $T_{10}$ commits which version should be deleted?
  If $T_{12}$ commits which version should be deleted?

- Alternatively, delete periodically all versions older than some number.
  - If the scheduler receives $R(x_j)$ and $x_j$ has been deleted, it aborts $T_i$.

Revisiting 2PL

Two Version 2PL (2V2PL)

- The DM keeps one or two versions of each data item $x$.
- When a $T_i$ wants to write $x$, it sets a $w_l(x)$ and it creates a new version of $x$, $x_i$.
- The $w_l(x)$ prohibits other transactions from writing $x$.
- When $T_i$ commits, the $x_i$ version of $x$ becomes $x$'s unique version (the before image of $x$ may now be deleted).
- Readers are allowed to place a $r_l$ on the a write locked $x$ and they read the previous version of $x$ (the before image).
  Therefore, a Read operation is performed on committed updates only (no cascading aborts).
**Commit**

- To delete the before image of \( x_i \) when \( T_i \) commits, we need to know that no other transaction reads \( x \).
- We introduced a third lock, *commit lock*. The compatibility matrix is:

<table>
<thead>
<tr>
<th></th>
<th>rl</th>
<th>wl</th>
<th>cl</th>
</tr>
</thead>
<tbody>
<tr>
<td>rl</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>wl</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
<tr>
<td>cl</td>
<td>n</td>
<td>n</td>
<td>n</td>
</tr>
</tbody>
</table>

**Read/Write Operations**

- **Write\(_i\)(x)**
  - If there is a \( wl \) or \( cl \) on \( x \), place \( W_i \) in *waiting-list(x)*.
  - If \( T_i \) already owns a \( wl \) on \( x \), send \( W_i(x) \) to DM.
  - In any other case (\( x \) is unlocked or read locked), set a \( wl(x) \) and send \( W_i(x) \) to DM. Data item \( x \) remains unaffected.

- **Read\(_i\)(x)**
  - If there is a \( cl \) on \( x \), place \( R_i \) in *waiting-list(x)*.
  - If \( T_i \) already owns a \( wl \) on \( x \) then send \( R_i(x) \) to DM.
  - In any other case (i.e., \( x \) is unlocked or write locked by another transaction), set \( rl \) and send \( R_i(x) \) to DM.

**Discussion**

- The 2V2PL is recoverable and avoids cascading aborts.
- Deadlocks are possible for one more reason:
  - \( T_1 \) tries to convert its \( rl \) on \( x \) to \( wl \)
  - \( T_2 \) tries to convert its \( wl \) on \( x \) to \( cl \)
  - Nothing special here; use any deadlock detection or prevention technique.

- Usually, in 2V2PL, it takes less time to commit a transaction than to execute it.
  - Therefore, commit locks delay Reads less than 2PL's write locks.