Goal: Integrity Across Sequence of Operations

• Update should complete entirely
  – update stipend set stipend = stipend*1.03;
  – What if it gets halfway and the machine crashes?

• What about multiple operations?
  – Withdraw x from Account1
  – Deposit x into Account2

• Simultaneous operations?
  – Print paychecks while stipend being updated

2 April, 2012
Solution: Transaction

- Sequence of operations grouped into a transaction
  - Externally viewed as Atomic: All happens at once
  - DBMS manages so even the programmer gets this view

ACID properties

Transactions have:

- Atomicity
  - All or nothing
- Consistency
  - Changes to values maintain integrity
- Isolation
  - Transaction occurs as if nothing else happening
- Durability
  - Once completed, changes are permanent
Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently.

- A user’s program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.

- A transaction is the DBMS’s abstract view of a user program: a sequence of reads and writes.

Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

- Issues: Effect of interleaving transactions, and crashes.
Atomicity of Transactions

- A transaction might commit after completing all its actions, or it could abort (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are atomic. That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS logs all actions so that it can undo the actions of aborted transactions.

Example

- Consider two transactions (Xacts):

  | T1 | BEGIN A=A+100, B=B-100 END |
  | T2 | BEGIN A=1.06*A, B=1.06*B END |

- Intuitively, the first transaction is transferring $100 from B’s account to A’s account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect must be equivalent to these two transactions running serially in some order.
Example (Contd.)

- Consider a possible interleaving (schedule):

  T1: \( A = A + 100 \), \( B = B - 100 \)
  T2: \( A = 1.06 \times A \), \( B = 1.06 \times B \)

- This is OK. But what about:

  T1: \( A = A + 100 \), \( B = B - 100 \)
  T2: \( A = 1.06 \times A \), \( B = 1.06 \times B \)

- The DBMS's view of the second schedule:

  T1: \( R(A), W(A) \), \( R(B), W(B) \)
  T2: \( R(A), W(A), R(B), W(B) \)

Scheduling Transactions

- **Serial schedule**: Schedule that does not interleave the actions of different transactions.

- **Equivalent schedules**: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.

- **Serializable schedule**: A schedule that is equivalent to some serial execution of the transactions.

(Note: If each transaction preserves consistency, every serializable schedule preserves consistency.)
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, “dirty reads”):

| T1:   | R(A), W(A),     | R(B), W(B), Abort |
| T2:   | R(A), W(A), C   |                 |

- Unrepeatable Reads (RW Conflicts):

| T1:   | R(A), R(A), W(A), C |
| T2:   | R(A), W(A), C       |

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

| T1:   | W(A), W(B), C |
| T2:   | W(A), W(B), C |

Database Management Systems 3ed, R. Ramakrishnan and J. Gehrke
**Aborting a Transaction**

- If a transaction \( Ti \) is aborted, all its actions have to be undone. Not only that, if \( Tj \) reads an object last written by \( Ti \), \( Tj \) must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction’s locks only at commit time.
  - If \( Ti \) writes an object, \( Tj \) can read this only after \( Ti \) commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

**The Log**

- The following actions are recorded in the log:
  - *\( Ti \) writes an object*: the old value and the new value.
    - Log record must go to disk *before* the changed page!
    - *\( Ti \) commits/aborts*: a log record indicating this action.
- Log records are chained together by Xact id, so it’s easy to undo a specific Xact.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.
Recovering From a Crash

- There are 3 phases in the *Aries* recovery algorithm:
  - **Analysis:** Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
  - **Redo:** Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
  - **Undo:** The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)
Chapters 16-17
Concurrency Control

T1          T2          ...          Tn

DB
(consistency constraints)

Example:

T1: Read(A)    T2: Read(A)
    A ← A+100   A ← A×2
    Write(A)    Write(A)
    Read(B)    Read(B)
    B ← B+100  B ← B×2
    Write(B)    Write(B)

Constraint: A=B
### Schedule A

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(A); A ← A+100</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B+100;</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A); A ← A×2;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td>250</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×2;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td>250</td>
</tr>
</tbody>
</table>

### Schedule B

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Read(A); A ← A+100</td>
<td>25</td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B+100;</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read(A); A ← A×2;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td>150</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×2;</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td>150</td>
</tr>
</tbody>
</table>
### Schedule C

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td></td>
<td>125</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>250</td>
<td>250</td>
</tr>
</tbody>
</table>

### Schedule D

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Read(A); A ← A+100</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>Write(A);</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>Read(B); B ← B+100;</td>
<td></td>
<td></td>
<td>50</td>
</tr>
<tr>
<td>Write(B);</td>
<td></td>
<td>150</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>250</td>
<td>150</td>
</tr>
</tbody>
</table>
**Schedule E**

**Same as Schedule D**

but with new T2’

<table>
<thead>
<tr>
<th>T1</th>
<th>T2’</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read(A); A ← A+100</td>
<td>Write(A);</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Write(A);</td>
<td>Read(A); A ← A×1;</td>
<td>125</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(A);</td>
<td></td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>Read(B); B ← B×1;</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Write(B);</td>
<td>125</td>
<td>125</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
A & = 25 \\
B & = 25 \\

\end{align*}
\]

- Want schedules that are “good”, regardless of
  - initial state and
  - transaction semantics
- Only look at order of read and writes

**Example:**

\[
Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)
\]
Example:
\[Sc = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)\]
\[Sc' = r_1(A)w_1(A)r_1(B)w_1(B)r_2(A)w_2(A)r_2(B)w_2(B)\]

However, for Sd:
\[Sd = r_1(A)w_1(A)r_2(A)w_2(A)r_2(B)w_2(B)r_1(B)w_1(B)\]

- as a matter of fact, 
  \[T_2 \text{ must precede } T_1\]
  in any equivalent schedule,
  i.e., \(T_2 \rightarrow T_1\)
• $T_2 \rightarrow T_1$
• Also, $T_1 \rightarrow T_2$

\[
\begin{align*}
T_1 &\leftrightarrow T_2 \\
\implies &\quad \text{Sd cannot be rearranged into a serial schedule} \\
\implies &\quad \text{Sd is not “equivalent” to any serial schedule} \\
\implies &\quad \text{Sd is “bad”}
\end{align*}
\]

**Returning to Sc**

\[
\text{Sc} = r_1(A) w_1(A) r_2(A) w_2(A) r_1(B) w_1(B) r_2(B) w_2(B)
\]

\[
\begin{align*}
T_1 &\rightarrow T_2 \\
T_1 &\rightarrow T_2
\end{align*}
\]

- no cycles $\implies$ Sc is “equivalent” to a serial schedule
  (in this case $T_1, T_2$)
**Concepts**

*Transaction:* sequence of \( r_i(x), w_i(x) \) actions

*Conflicting actions:* \( r_1(A), w_2(A), w_1(A), w_2(A), r_1(A), w_2(A) \)

*Schedule:* represents chronological order in which actions are executed

*Serial schedule:* no interleaving of actions or transactions

---

**What about concurrent actions?**

Ti issues: \( \text{read}(x,t) \)  
System issues: \( \text{input}(x) \)  
Input(\( X \))  
\( t \leftarrow x \)

T2 issues: \( \text{write}(B,S) \)  
System issues: \( \text{input}(B) \)  
\( \text{input}(B) \) completes  
\( B \leftarrow S \)  
System issues: \( \text{output}(B) \)  
\( \text{output}(B) \) completes
So net effect is either
  • $S = \ldots r_1(x) \ldots w_2(b) \ldots$ or
  • $S = \ldots w_2(B) \ldots r_1(x) \ldots$

What about conflicting, concurrent actions on same object?

```
start r_1(A)  end r_1(A)
  ↓         ↓
start w_2(A)  end w_2(A)
  ↑         ↑
  time
```

• Assume equivalent to either $r_1(A) \ w_2(A)$
  or  $w_2(A) \ r_1(A)$

• $\Rightarrow$ low level synchronization mechanism

• Assumption called “atomic actions”
Definition

$S_1$, $S_2$ are conflict equivalent schedules if $S_1$ can be transformed into $S_2$ by a series of swaps on non-conflicting actions.

Definition

A schedule is conflict serializable if it is conflict equivalent to some serial schedule.
Precedence graph $P(S)$ ($S$ is schedule)

Nodes: transactions in $S$
Arcs: $T_i \rightarrow T_j$ whenever
- $p_i(A), q_j(A)$ are actions in $S$
- $p_i(A) <_S q_j(A)$
- at least one of $p_i, q_j$ is a write

Exercise:
• What is $P(S)$ for $S = w_3(A) w_2(C) r_1(A) w_1(B) r_1(C) w_2(A) r_4(A) w_4(D)$

• Is $S$ serializable?
Lemma

$S_1$, $S_2$ conflict equivalent $\Rightarrow P(S_1) = P(S_2)$

Proof:
Assume $P(S_1) \neq P(S_2)$

$\Rightarrow \exists T_i: T_i \rightarrow T_j$ in $S_1$ and not in $S_2$

$\Rightarrow S_1 = \ldots p_i(A)\ldots q_j(A)\ldots$ \hspace{1cm} $\left\{ \begin{array}{l} p_i, q_j \\ \text{conflict} \end{array} \right.$

$S_2 = \ldots q_j(A)\ldots p_i(A)\ldots$

$\Rightarrow S_1, S_2$ not conflict equivalent
Note: \( P(S_1) = P(S_2) \not\Rightarrow S_1, S_2 \) conflict equivalent

Counter example:

\[
S_1 = w_1(A) \ r_2(A) \ w_2(B) \ r_1(B)
\]

\[
S_2 = r_2(A) \ w_1(A) \ r_1(B) \ w_2(B)
\]

**Theorem**

\( P(S_1) \) acyclic \( \iff S_1 \) conflict serializable

\( (\iff) \) Assume \( S_1 \) is conflict serializable

\[ \Rightarrow \exists S_s: S_s, S_1 \) conflict equivalent \]

\[ \Rightarrow P(S_s) = P(S_1) \]

\[ \Rightarrow P(S_1) \) acyclic since \( P(S_s) \) is acyclic \]
Theorem

\[ P(S_1) \text{ acyclic} \iff S_1 \text{ conflict serializable} \]

(\(\Rightarrow\)) Assume \(P(S_1)\) is acyclic

Transform \(S_1\) as follows:

1. Take \(T_1\) to be transaction with no incident arcs
2. Move all \(T_1\) actions to the front
   \[ S_1 = \ldots \ q_j(A)\ldots p_1(A) \ldots \]
3. we now have \(S_1 = <T_1 \text{ actions}>\ldots \text{ rest }...>\)
4. repeat above steps to serialize rest!

How to enforce serializable schedules?

**Option 1:** run system, recording \(P(S)\);
at end of day, check for \(P(S)\)cycles and declare if execution was good
How to enforce serializable schedules?

Option 2: prevent P(S) cycles from occurring

\[ T_1 \rightarrow T_2 \rightarrow \ldots \rightarrow T_n \]

Scheduler

DB

A locking protocol

Two new actions:

- lock (exclusive): \( li(A) \)
- unlock: \( ui(A) \)

\[ T_1 \rightarrow T_2 \]

scheduler

lock

table
Rule #1: Well-formed transactions

\[ T_i: \ldots \ l_i(A) \ldots p_i(A) \ldots u_i(A) \ldots \]

Rule #2: Legal scheduler

\[ S = \ldots \ l_i(A) \ldots \ u_i(A) \ldots \]

\[ \text{no } l_j(A) \]
Exercise:

• What schedules are legal?
  What transactions are well-formed?
  S1 = l_1(A)l_1(B)r_1(A)w_1(B)l_2(B)u_1(A)u_1(B)
      r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)
  S2 = l_1(A)r_1(A)w_1(B)u_1(A)u_1(B)
      l_2(B)r_2(B)w_2(B)l_3(B)r_3(B)u_3(B)
  S3 = l_1(A)r_1(A)u_1(A)l_1(B)w_1(B)u_1(B)
      l_2(B)r_2(B)w_2(B)u_2(B)l_3(B)r_3(B)u_3(B)
### Schedule F

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>l₁(A);Read(A)</td>
<td></td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>A←A+100;Write(A);u₁(A)</td>
<td></td>
<td>125</td>
<td></td>
</tr>
<tr>
<td>l₂(A);Read(A)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A←Ax2;Write(A);u₂(A)</td>
<td></td>
<td>250</td>
<td></td>
</tr>
<tr>
<td>l₂(B);Read(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B←Bx2;Write(B);u₂(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>l₁(B);Read(B)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B←B+100;Write(B);u₁(B)</td>
<td></td>
<td>150</td>
<td></td>
</tr>
</tbody>
</table>

4/2/2012
Rule #3  Two phase locking (2PL) for transactions

\[ T_i = \ldots \text{li}(A) \ldots \ldots \text{ui}(A) \ldots \ldots \]

- no unlocks
- no locks

# locks held by Ti

- Growing Phase
- Shrinking Phase

Time

4/2/2012 Chris Clifton - CS541 51
Schedule G

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>l₁(A); Read(A)</td>
<td>l₂(A); Read(A)</td>
</tr>
<tr>
<td>A ← A + 100; Write(A)</td>
<td>A ← A + 100; Write(A)</td>
</tr>
<tr>
<td>l₁(B); u₁(A)</td>
<td>Read(B); B ← B + 100</td>
</tr>
<tr>
<td></td>
<td>Write(B); u₁(B)</td>
</tr>
</tbody>
</table>

- Schedule G

- Schedule G
### Schedule G

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
</table>
| l₁(A); Read(A)  | l₂(A); Read(A)  
| A ← A + 100; Write(A) | A ← A + 2; Write(A) |
| l₁(B); u₁(A)    | l₂(B); u₂(A); Read(B) |
| Read(B); B ← B + 100 | B ← B + 2; Write(B); u₂(B) |
| Write(B); u₁(B) |                           |

### Schedule H (T₂ reversed)

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
</table>
| l₁(A); Read(A)  | l₂(B); Read(B)  
| A ← A + 100; Write(A) | B ← B + 2; Write(B) |
| l₁(B); u₁(A)    | l₂(B); u₂(A); Read(B) |
|                           | delayd              |
|                           | delayd              |
• Assume deadlocked transactions are rolled back
  – They have no effect
  – They do not appear in schedule

E.g., Schedule H =

This space intentionally left blank!

Next step:
Show that rules #1,2,3 ⇒ conflict-serializable schedules
Conflict rules for $l_i(A), u_i(A)$:

- $l_i(A), l_j(A)$ conflict
- $l_i(A), u_j(A)$ conflict

Note: no conflict $< u_i(A), u_j(A)>, < l_i(A), r_j(A)>,...$

**Theorem** Rules #1,2,3 $\Rightarrow$ conflict (2PL) serializable schedule

To help in proof:

**Definition** $\text{Shrink}(Ti) = \text{SH}(Ti) =$ first unlock action of $Ti$
Lemma
Ti $\rightarrow$ Tj in S $\Rightarrow$ SH(Ti) $<_{S}$ SH(Tj)

Proof of lemma:
Ti $\rightarrow$ Tj means that
S = ... $p_i(A)$ ... $q_j(A)$ ...; p,q conflict
By rules 1,2:
S = ... $p_i(A)$ ... $u_i(A)$ ... $l_i(A)$ ... $q_j(A)$ ...
By rule 3: SH(Ti) SH(Tj)
So, SH(Ti) $<_{S}$ SH(Tj)

Theorem  Rules #1,2,3 $\Rightarrow$ conflict (2PL) serializable schedule

Proof:
(1) Assume P(S) has cycle
$T_1 \rightarrow T_2 \rightarrow ... \rightarrow T_n \rightarrow T_1$
(2) By lemma: SH(T_1) $<$ SH(T_2) $<$ ... $<$ SH(T_1)
(3) Impossible, so P(S) acyclic
(4) $\Rightarrow$ S is conflict serializable
Beyond this simple 2PL protocol, it is all a matter of improving performance and allowing more concurrency.

- Shared locks
- Multiple granularity
- Inserts, deletes and phantoms
- Other types of C.C. mechanisms

**Shared locks**

So far:

\[ S = \ldots l_1(A) \; r_1(A) \; u_1(A) \; \ldots \; l_2(A) \; r_2(A) \; u_2(A) \; \ldots \]

\[ \text{Do not conflict} \]

Instead:

\[ S = \ldots ls_1(A) \; r_1(A) \; ls_2(A) \; r_2(A) \; \ldots \; us_1(A) \; us_2(A) \]
Lock actions

$l_t(A)$: lock $A$ in $t$ mode ($t$ is $S$ or $X$)
$u_t(A)$: unlock $t$ mode ($t$ is $S$ or $X$)

Shorthand:
$u(A)$: unlock whatever modes
$T_i$ has locked $A$

Rule #1  Well formed transactions

$T_i = \ldots l-S_1(A) \ldots r_1(A) \ldots u_1(A) \ldots$
$T_i = \ldots l-X_1(A) \ldots w_1(A) \ldots u_1(A) \ldots$
• What about transactions that read and write same object?

**Option 1:** Request exclusive lock

\[ T_i = \ldots l-X_1(A) \ldots r_1(A) \ldots w_1(A) \ldots u(A) \ldots \]

• What about transactions that read and write same object?

**Option 2:** Upgrade

(E.g., need to read, but don’t know if will write…)

\[ T_i = \ldots l-S_1(A) \ldots r_1(A) \ldots l-X_1(A) \ldots w_1(A) \ldots u(A) \ldots \]

Think of
- Get 2nd lock on A, or
- Drop S, get X lock
Rule #2  Legal scheduler

\[ S = \ldots l-S_i(A) \ldots u_i(A) \ldots \]
\[ \text{no } l-X_i(A) \]

\[ S = \ldots l-X_i(A) \ldots u_i(A) \ldots \]
\[ \text{no } l-X_i(A) \]
\[ \text{no } l-S_i(A) \]

A way to summarize Rule #2

Compatibility matrix

<table>
<thead>
<tr>
<th>Comp</th>
<th>S</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>X</td>
<td>false</td>
<td>false</td>
</tr>
</tbody>
</table>
Rule # 3  2PL transactions

No change except for upgrades:
(I) If upgrade gets more locks
   (e.g., $S \rightarrow \{S, X\}$) then no change!
(II) If upgrade releases read (shared)
     lock (e.g., $S \rightarrow X$
     - can be allowed in growing phase

Theorem  Rules 1,2,3 $\Rightarrow$ Conf.serializable
          for S/X locks      schedules

Proof: similar to X locks case

Detail:
$l-t_i(A)$, $l-r_j(A)$ do not conflict if $\text{comp}(t,r)$
$l-t_i(A)$, $u-r_j(A)$ do not conflict if $\text{comp}(t,r)$
Lock types beyond S/X

Examples:

(1) increment lock
(2) update lock

Example (1): increment lock

- Atomic increment action: \( \text{IN}_i(A) \)
  
  \{ \text{Read}(A); A \leftarrow A+k; \text{Write}(A) \} 

- \( \text{IN}_i(A) \), \( \text{IN}_j(A) \) do not conflict!

\[
\begin{align*}
A &= 5 & \text{IN}_i(A) & \rightarrow & \text{IN}_j(A) \\
+10 & & +2 & & +10 & & +2 \\
A &= 15 & & A &= 17 & & \\
& & \text{IN}_j(A) & & \text{IN}_i(A) \\
& & & & & & \\
A &= 7 & & & & & \\
& & \text{IN}_i(A) & & & & & \\
\end{align*}
\]
### Comp

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Comp

<table>
<thead>
<tr>
<th></th>
<th>S</th>
<th>X</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>X</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>I</td>
<td>F</td>
<td>F</td>
<td>T</td>
</tr>
</tbody>
</table>
**Update locks**

A common deadlock problem with upgrades:

<table>
<thead>
<tr>
<th></th>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-S₁(A)</td>
<td></td>
<td>L-S₂(A)</td>
</tr>
<tr>
<td>L-X₁(A)</td>
<td></td>
<td>L-X₂(A)</td>
</tr>
</tbody>
</table>

--- Deadlock ---

**Solution**

If $T_i$ wants to read $A$ and knows it may later want to write $A$, it requests update lock (not shared)
### New request

<table>
<thead>
<tr>
<th>Comp</th>
<th>S</th>
<th>X</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Lock already held in

### New request

<table>
<thead>
<tr>
<th>Comp</th>
<th>S</th>
<th>X</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>T</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>X</td>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>U</td>
<td>TorF</td>
<td>F</td>
<td>F</td>
</tr>
</tbody>
</table>

-> symmetric table?
Note: object A may be locked in different modes at the same time...

\[ S_1 = \ldots l-S_1(A)\ldots l-S_2(A)\ldots l-U_3(A)\ldots l-S_4(A)\ldots l-U_4(A)\ldots ? \]

- To grant a lock in mode t, mode t must be compatible with all currently held locks on object

How does locking work in practice?

- Every system is different
  
  (E.g., may not even provide CONFLICT-serializable schedules)

- But here is one (simplified) way ...
Sample Locking System:

(1) Don’t trust transactions to request/release locks
(2) Hold all locks until transaction commits

![Graph showing # of locks over time]

Sample Locking System:

Ti

Scheduler, part I

Scheduler, part II

lock table

DB

Ti

Read(A),Write(B)

I(A),Read(A),I(B),Write(B)…

Read(A),Write(B)
**Lock table** Conceptually

![Lock Table Diagram](image)

<table>
<thead>
<tr>
<th>A</th>
<th>Λ</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Λ</td>
</tr>
</tbody>
</table>

If null, object is unlocked

Lock info for B
Lock info for C

---

**But use hash table:**

A

![Hash Table Diagram](image)

<table>
<thead>
<tr>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

If object not found in hash table, it is unlocked
Lock info for A - example

Object:A
Group mode:U
Waiting:yes
List:

T1 S no →
T2 U no →
T3 X yes 🔒 →

To other T3 records

What are the objects we lock?

Relation A
Relation B
Tuple A
Tuple B
Tuple C
Disk block A
Disk block B
Disk block C
DB
DB
DB

Transaction mode wait? Next T_link
Locking works in any case, but should we choose small or large objects?
If we lock large objects (e.g., Relations)
  – Need few locks
  – Low concurrency
If we lock small objects (e.g., tuples, fields)
  – Need more locks
  – More concurrency

We can have it both ways!!
Ask any janitor to give you the solution...
Example

```
Example
```

```
Example
```

```
Example
```

```
Example
```

```
Example
```

```
Example
```
Multiple granularity

Comp | Requestor
-----|---------
IS   | IX      | S  | SIX | X
----|---------|----|-----|----
IS   |         |    |     |    
Holder
IX   |         |    |     |    
S    |         |    |     |    
SIX  |         |    |     |    
X    |         |    |     |    

Multiple granularity

Comp | Requestor
-----|---------
IS   | IX      | S  | SIX | X
----|---------|----|-----|----
IS   | T       | T  | T   | T   | F
Holder
IX   | T       | T  | F   | F   | F
S    | T       | F  | T   | F   | F
SIX  | T       | F  | F   | F   | F
X    | F       | F  | F   | F   | F
<table>
<thead>
<tr>
<th>Parent locked in</th>
<th>Child can be locked in</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>IS, S</td>
</tr>
<tr>
<td>IX</td>
<td>IS, S, IX, X, SIX</td>
</tr>
<tr>
<td>S</td>
<td>[S, IS] not necessary</td>
</tr>
<tr>
<td>SIX</td>
<td>X, IX, [SIX]</td>
</tr>
<tr>
<td>X</td>
<td>none</td>
</tr>
</tbody>
</table>
Rules

(1) Follow multiple granularity comp function
(2) Lock root of tree first, any mode
(3) Node Q can be locked by Ti in S or IS only if parent(Q) locked by Ti in IX or IS
(4) Node Q can be locked by Ti in X,SIX,IX only if parent(Q) locked by Ti in IX,SIX
(5) Ti is two-phase
(6) Ti can unlock node Q only if none of Q’s children are locked by Ti

• End 11/4
Exercise:

- Can T2 access object f2.2 in X mode? What locks will T2 get?

Exercise:

- Can T2 access object f2.2 in X mode? What locks will T2 get?
Exercise:

- Can T2 access object f3.1 in X mode? What locks will T2 get?

Exercise:

- Can T2 access object f2.2 in S mode? What locks will T2 get?
Exercise:

• Can T2 access object f2.2 in X mode? What locks will T2 get?

Insert + delete operations

```
A
::
Z
α
```

Insert
**Modifications to locking rules:**

1. Get exclusive lock on A before deleting A
2. At insert A operation by Ti, Ti is given exclusive lock on A

Still have a problem: **Phantoms**

Example: relation R (E#, name, …)

constraint: E# is key

use tuple locking

<table>
<thead>
<tr>
<th></th>
<th>E#</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>o1</td>
<td>55</td>
<td>Smith</td>
</tr>
<tr>
<td>o2</td>
<td>75</td>
<td>Jones</td>
</tr>
</tbody>
</table>
\[ T_1: \text{Insert} <99, \text{Gore,} \ldots> \text{ into } R \]
\[ T_2: \text{Insert} <99, \text{Bush,} \ldots> \text{ into } R \]

<table>
<thead>
<tr>
<th>( T_1 )</th>
<th>( T_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1(o_1) )</td>
<td>( S_2(o_1) )</td>
</tr>
<tr>
<td>( S_1(o_2) )</td>
<td>( S_2(o_2) )</td>
</tr>
<tr>
<td>( \text{Check Constraint} )</td>
<td>( \text{Check Constraint} )</td>
</tr>
<tr>
<td>( \vdots )</td>
<td>( \vdots )</td>
</tr>
<tr>
<td>( \text{Insert } o_3[99, \text{Gore,} \ldots] )</td>
<td>( \text{Insert } o_4[99, \text{Bush,} \ldots] )</td>
</tr>
</tbody>
</table>

**Solution**

- Use multiple granularity tree
- Before insert of node Q, lock parent(Q) in X mode
**Back to example**

<table>
<thead>
<tr>
<th>T₁: Insert&lt;99,Gore&gt;</th>
<th>T₂: Insert&lt;99,Bush&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₁(R)</td>
<td></td>
</tr>
<tr>
<td>Check constraint</td>
<td></td>
</tr>
<tr>
<td>Insert&lt;99,Gore&gt;</td>
<td></td>
</tr>
<tr>
<td>U(R)</td>
<td></td>
</tr>
</tbody>
</table>

X₂(R)  
Check constraint  
Oops! e# = 99 already in R!

**Instead of using R, can use index on R:**

Example:

```
    R
   /\    /
Index 0<E#<100  Index 100<E#<200
   \   /       /   /
E#=2 E#=5   E#=107 E#=109
```

...
• This approach can be generalized to multiple indexes...
Next:
- Tree-based concurrency control
- Validation concurrency control

Example
- all objects accessed through root, following pointers

-can we release A lock if we no longer need A??
Idea: traverse like "Monkey Bars"

Why does this work?

• Assume all $T_i$ start at root; exclusive lock
• $T_i \rightarrow T_j \Rightarrow T_i$ locks root before $T_j$

• Actually works if we don’t always start at root
**Rules: tree protocol (exclusive locks)**

1. First lock by $T_i$ may be on any item
2. After that, item $Q$ can be locked by $T_i$ only if parent($Q$) locked by $T_i$
3. Items may be unlocked at any time
4. After $T_i$ unlocks $Q$, it cannot relock $Q$

- Tree-like protocols are used typically for B-tree concurrency control

  ![Tree diagram](image)

  E.g., during insert, do not release parent lock, until you are certain child does not have to split
Validation

Transactions have 3 phases:

(1) **Read**
   - all DB values read
   - writes to temporary storage
   - no locking

(2) **Validate**
   - check if schedule so far is serializable

(3) **Write**
   - if validate ok, write to DB

Key idea

- Make validation atomic
- If $T_1, T_2, T_3, ...$ is validation order, then resulting schedule will be conflict equivalent to $S_s = T_1 T_2 T_3...$
To implement validation, system keeps two sets:

- **FIN** = transactions that have finished phase 3 (and are all done)
- **VAL** = transactions that have successfully finished phase 2 (validation)

Example of what validation must prevent:

- $\text{RS}(T_2) = \{B\}$
- $\text{RS}(T_3) = \{A, B\}$
- $\text{WS}(T_2) = \{B, D\}$
- $\text{WS}(T_3) = \{C\}$

$T_2 \cap T_3 \neq \emptyset$
Example of what validation must prevent:

\[ RS(T_2) = \{B\} \quad RS(T_3) = \{A,B\} \]
\[ WS(T_2) = \{B,D\} \quad WS(T_3) = \{C\} \]

\[ \not\equiv \emptyset \]

Another thing validation must prevent:

\[ RS(T_2) = \{A\} \quad RS(T_3) = \{A,B\} \]
\[ WS(T_2) = \{D,E\} \quad WS(T_3) = \{C,D\} \]

BAD: \( w_3(D) \quad w_2(D) \)
Another thing validation must prevent:

\[ RS(T_2) = \{ A \} \quad RS(T_3) = \{ A, B \} \]
\[ WS(T_2) = \{ D, E \} \quad WS(T_3) = \{ C, D \} \]

Validation rules for \( T_j \):

1. When \( T_j \) starts phase 1:
   \[ \text{ignore}(T_j) \leftarrow \text{FIN} \]

2. At \( T_j \) Validation:
   \[ \text{if check } (T_j) \text{ then} \]
   \[ [ \text{VAL} \leftarrow \text{VAL} \cup \{ T_j \}; \]
   \[ \text{do write phase;} \]
   \[ \text{FIN} \leftarrow \text{FIN} \cup \{ T_j \} ] \]
Check \((T_j)\):

For \(T_i \in \text{VAL - IGNORE } (T_j)\) DO

\[
\text{IF } [ \text{WS}(T_i) \cap \text{RS}(T_j) \neq \emptyset \text{ OR } T_i \notin \text{FIN} ] \text{ THEN RETURN false;}
\]

RETURN true;

Is this check too restrictive?

---

**Improving Check\((T_j)\)**

For \(T_i \in \text{VAL - IGNORE } (T_j)\) DO

\[
\text{IF } [ \text{WS}(T_i) \cap \text{RS}(T_j) \neq \emptyset \text{ OR } (T_i \notin \text{FIN} \text{ AND WS}(T_i) \cap \text{WS}(T_j) \neq \emptyset)]
\]

THEN RETURN false;

RETURN true;
Validation (also called optimistic concurrency control) is useful in some cases:
- Conflicts rare
- System resources plentiful
- Have real time constraints
**Summary**

Have studied C.C. mechanisms used in practice
- 2 PL
- Multiple granularity
- Tree (index) protocols
- Validation