



Transaction Processing: Recovery and Concurrency Control

CS 178



Transactions

- ❖ Concurrent execution of user programs essential for good performance.
 - Keep CPU humming when disk IO takes place.
- ❖ 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.



Why Have Concurrent Processes?

- ❖ Better transaction throughput, response time
- ❖ Done via better utilization of resources:
 - While one process is doing a disk read, another can be using the CPU or reading another disk.
- ❖ **DANGER DANGER!** Concurrency could lead to incorrectness!
 - Must carefully manage concurrent data access.
 - There's (much!) more here than the usual OS tricks!



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*.

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*A, B=1.06*B
```

- ❖ This is OK. But what about:

```
T1: A=A+100, B=B-100
T2: A=1.06*A, B=1.06*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)
```

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.

Transactions

- ❖ Basic concurrency/recovery concept: a *transaction* (*Xact*).
 - A sequence of many actions which are considered to be one atomic unit of work.
- ❖ DBMS "actions":
 - *reads, writes*
 - Special actions: *commit, abort*
 - for now, assume reads and writes are on tuples;
 - Model *Xact* operations only as Read R(A) and Write W(A)

The **ACID** Properties

- ❖ **A**tomicity: All actions in the Xact happen, or none happen.
- ❖ **C**onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- ❖ **I**solation: Execution of one Xact is isolated from that of other Xacts.
- ❖ **D**urability: If a Xact commits, its effects persist.

Passing the ACID Test

- ❖ **Concurrency Control**
 - Guarantees Consistency and Isolation, given Atomicity.
- ❖ **Logging and Recovery**
 - Guarantees Atomicity and Durability.
 - Log file based recovery techniques
- ❖ We'll do Recovery Methods first
 - Assume no concurrency and study recovery methods - Log Based recovery
 - Concurrency control methods will then generate schedules that are "recoverable"

Concept: Log-based Recovery

- ❖ Write to a log file before writing to database
 - Enter log records
- ❖ Transaction states:
 - Commit, Abort, Start
- ❖ Redo and Undo Operations
 - Redo Xact (write new values) if commit record in log
 - Undo Xact (restore old values) if no commit
- ❖ Deferred Modification or Immediate
 - Allow DB Write only after commit record written vs. allow write immediately after record written into log file

Database Recovery

- ❖ Outline recovery process without concurrency
 - Then study Concurrency control techniques that generate schedules that are "recoverable"

Failures in DBMS

- ❖ Computer system subject to failure
 - Power failure: lose contents of main memory
 - Disk crash: could lose non-volatile storage
 - Software errors
- ❖ When failure occurs - info related to DB is lost
- ❖ Recovery scheme responsible for handling failures and restoring database to consistent state

Storage Types

- ❖ Volatile storage: do not survive system crash
 - Main memory, cache
- ❖ Nonvolatile storage: usually survives system crash
 - Disks, tapes
- ❖ Stable storage: information is never lost
 - Need to implement this using redundancy

ACID

- ❖ Recall ACID properties
 - A transaction should be treated as an atomic unit, where all operations must occur or none at all

Recovery Algorithm and Failures

- ❖ Various types of failures can occur
 - To determine how to recover, must also be able to identify failure modes of devices
 - ◆ Logical errors, system errors, system crash, disk crash
 - ◆ Programmer errors, deadlocks, power failure
- ❖ Recovery algorithm has two parts
 - Actions taken during normal operation to ensure system can recover from failure
 - Actions taken after a failure to restore database to consistent state

Data Transfers

- ❖ What is relevant to database consistency?
 - Data being read or written into
 - Model this using the data transfers to DBMS
- ❖ Assume block transfers initiated (by memory management system -OS) through
 - **Input(X)**: transfers block X to main memory
 - **Output(X)**: transfers buffer block X to disk

Database Data Transfers

- ❖ Transactions use **read** and **write** operations
- ❖ **Read(X,xi)**: assigns value of data item X to local variable xi
 - If block with X is not in main memory, then issue **input(X)** and copy to xi
- ❖ **Write(X,xi)**: assign value of local variable xi to item X in database
 - If block with X is not in main memory, then issue **input(X)** and assign value xi to X in main memory
- ❖ Note that **Output(X)** is done by OS
 - Buffer is full, force-output a page to reflect change

Transaction Model

- ❖ Transaction (Xact) is program unit that accesses and possibly updates various data
- ❖ During execution of transaction:
 - **Each data read or written into exactly once**
 - Require Xacts do not violate consistency constraints
 - During execution DB may be temporarily inconsistent

Example Transaction

- ❖ T is a Xact that transfers \$50 from account A to account B. Initially A=1000, B=2000
 - Consistency constraint: sum of A and B is unchanged
- ```
T: read(A,a)
a= a-50;
write(A,a)
read(B,b)
b= b+50
write(B,b)
end-commit
```

## Failures in example

- ❖ Suppose failure occurs after output(A) but before output(B)
  - Database has A=950 and B=2000 – not consistent
- ❖ Note that each program within a transaction does not take the database from consistent state to another consistent state; only Xacts do that
- ❖ Ensuring correctness is responsibility of programmer
- ❖ Ensuring atomicity is responsibility of DBMS
  - If failure, then a transaction may be aborted and system must ensure that aborted Xacts have no effect on DB

## Transaction States

- ❖ Aborted transaction has no effect on DB
  - Therefore database must be *rolled back* to restore to the state before Xact
- ❖ Xact that successfully completes must be *committed* and database updated accordingly
- ❖ Effect of committed Xact cannot be undone by aborting Xact, must use a *compensating* Xact

## Transaction States

- ❖ **Active**: initial state
- ❖ **Partially Committed**: after last statement
- ❖ **Failed**: after discovering normal exec cannot proceed
- ❖ **Aborted**: after Xact rolled back and DB restored to original state
- ❖ **Committed**: after successful completion
  - Can represent above by a finite state diagram

## The Committed Xact

- ❖ Xact enters committed state if it has partially committed (i.e., last statement executed) and it is guaranteed that it is never aborted
  - **This must be ensured by the recovery system!**
  - This is the “*semantics*” of the **commit** instruction

## Dealing with aborted Xacts

- ❖ Restart transaction: if it was aborted due to H/W or system error and not as a result of logical error then treated as new Xact
- ❖ Kill the transaction: if some error which can only be corrected by the application program

## Log-based Recovery Systems

- ❖ Concept: keep a log (i.e., record) of the states of the Xact and refer to log to determine how to update database to maintain consistency
- ❖ **Normal operation**: reflect all changes first in log file
- ❖ **Recovering from failure**: look up log file to figure out next step

## The Log file

- ❖ Reflect all changes first in the log file
  - Output the log file to disk **before** making update to database
  - If system fails, then activities of Xact are recorded in the log file
    - ◆ Look up log file to determine valid changes
- ❖ Info to be stored: info about Xact states and variables modified
  - Log records: store for each Xact the state
    - ◆ <Ti, start>: Xact Ti has started
    - ◆ <Ti, Xj,V1,V2>: Ti performed write on Xj from old value V1 to new value V2
    - ◆ <Ti, Commit>: Ti has committed

## Log file

- ❖ Log record for a write must be created, and written to disk, before database modified
  - Using the old value, we can undo a modification
  - Assume log record is written immediately into log file
    - ◆ Force output the block of log file
  - Log file contains complete record of DB activity
    - ◆ Can get large and needs to be managed efficiently using checkpoints

## Execution of Xact and Log-file entry

- ❖ Before  $T_i$  starts, write  $\langle T_i, \text{start} \rangle$  record into log
- ❖ A write operation by Xact results in writing a new record  $\langle T_i, X_j, V1, V2 \rangle$  into log file
- ❖ When  $T_i$  partially commits, all its statements have been executed: therefore write  $\langle T_i, \text{Commit} \rangle$  record into log file
  - $T_i$  then enters committed state
    - ◆ This is the semantics of commit instruction

## Example

| Xact        | Log File                             | DB(A=1000,B=2000,C=500) |
|-------------|--------------------------------------|-------------------------|
| $T_0$ start | $\langle T_0, \text{start} \rangle$  |                         |
| Read(A,a)   |                                      |                         |
| A=a-50      |                                      |                         |
| Write(A,a)  | $\langle T_0, A, 1000, 950 \rangle$  |                         |
| Read(B,b)   |                                      |                         |
| B=b+50      |                                      |                         |
| Write(B,b)  | $\langle T_0, B, 2000, 2050 \rangle$ |                         |
| End $T_0$   | $\langle T_0, \text{Commit} \rangle$ |                         |
| $T_1$ Start | $\langle T_1, \text{Start} \rangle$  |                         |
| Read(C,c)   |                                      |                         |
| c=c-100     |                                      |                         |
| Write(C,c)  | $\langle T_1, C, 500, 400 \rangle$   |                         |
| End $T_1$   | $\langle T_1, \text{Commit} \rangle$ |                         |

## How to recover ?

- ❖ Two major schemes for log-based recovery
  - Deferred modification
  - Immediate modification
- ❖ Difference
  - When to allow updates to database ?
    - ◆ Defer all updates to DB until Xact finishes writing Commit record
    - ◆ Allow immediate updates as soon as a log record is written to disk

## Deferred Modification Scheme

- ❖ Defer/postpone actual updates to DB until Xact completes successfully and commits.
  - Updates recorded in log file and Xact workspace only (main memory)
- ❖ If system crashes before commit, then ignore transaction in log file
  - After Xact reaches commit point, and the log file is force-outputted into the disk the updates are recorded in the database

## Immediate Modification Scheme

- ❖ Allow database modifications to be output to database while Xact is still in active state – i.e., uncommitted modifications allowed
  - Provisions must be made to **undo** the effect of updates of aborted Xacts, i.e., need capability to **rollback** a Xact
- ❖ Before Output operation into database, log record must be written to disk

## Recovery using Log File

- ❖ When has a transaction committed ?
  - When  $\langle T_i, \text{start} \rangle$  and  $\langle T_i, \text{Commit} \rangle$  record is in the log file
- ❖ When has a transaction failed/aborted ?
  - When  $\langle T_i, \text{Start} \rangle$  is in the log file but no  $\langle T_i, \text{Commit} \rangle$
- ❖ What do we do with a committed Xact?
  - Recall definition: action of committed Xacts must be persistent; i.e., make sure DB reflects these

## Recovery using Log file

- ❖ When system fails, what do we do ?
- ❖ Look through log file
  - Recall that before writing into DB, we have written into log file
- ❖ For each Xact  $T_i$ , if  $\langle T_i, \text{Start} \rangle$  and  $\langle T_i, \text{Commit} \rangle$  records exist, then make Xact effects permanent
  - **Redo** the transaction
- ❖ What if Xact has not committed ?
  - Depends on the modification scheme used

## The Redo and Undo procedures

- ❖ Redo( $T_i$ )
  - Restores the database items to the new values
  - these can be obtained from the log records
- ❖ Undo( $T_i$ )
  - Restores the database to the old values
  - These can be obtained from the log records
  - This is essentially a rollback operation

## Deferred Modification Recovery

- ❖ When recovering from a system failure, examine the log file
- ❖ For each Xact in the log file, IF  $\langle T_i, \text{Start} \rangle$  record and  $\langle T_i, \text{Commit} \rangle$  record exist in the log file then Redo( $T_i$ )
  - Note that if a Xact has not committed (i.e., written  $\langle T_j, \text{Commit} \rangle$  into the log file) then it has not modified the DB and no effects need to be undone

## Example

|            |                                      |                         |
|------------|--------------------------------------|-------------------------|
| Xact       | Log File                             | DB(A=1000,B=2000,C=500) |
| T0 start   | $\langle T_0, \text{start} \rangle$  |                         |
| Read(A,a)  |                                      |                         |
| A=a-50     |                                      |                         |
| Write(A,a) | $\langle T_0, A, 1000, 950 \rangle$  |                         |
| Read(B,b)  |                                      |                         |
| B=b+50     |                                      |                         |
| Write(B,b) | $\langle T_0, B, 2000, 2050 \rangle$ |                         |
| End T0     | $\langle T_0, \text{Commit} \rangle$ |                         |
| T1 Start   | $\langle T_1, \text{Start} \rangle$  | A=950,B=2050,C=500      |
| Read(C,c)  |                                      |                         |
| c=c-100    |                                      |                         |
| Write(C,c) | $\langle T_1, C, 500, 400 \rangle$   |                         |
| End T1     | $\langle T_1, \text{Commit} \rangle$ | A=950,B=2050,C=400      |

## Recovering from Failure

- ❖ Where does failure occur and what action to take (i.e., redo or undo)
  - Based entirely on contents of log file
- ❖ Example:
  - When  $\langle T_0, \text{Start} \rangle$  but no  $\langle T_0, \text{Commit} \rangle$ 
    - ◆ Do nothing
  - When  $\langle T_0, \text{Start} \rangle$  and  $\langle T_0, \text{Commit} \rangle$ , and  $\langle T_1, \text{start} \rangle$  but no  $\langle T_1, \text{Commit} \rangle$ 
    - ◆ Redo( $T_0$ )
  - When both  $T_0$  and  $T_1$  have start and commit records
    - ◆ Redo( $T_0$ ) and Redo( $T_1$ )

## Immediate Modification

- ❖ When recovering from a system failure, examine the log file
- ❖ For each Xact in the log file, IF  $\langle T_i, \text{Start} \rangle$  record and  $\langle T_i, \text{Commit} \rangle$  record exist in the log file then Redo( $T_i$ )
- ❖ For each Xact in the log file, IF  $\langle T_j, \text{Start} \rangle$  record but no  $\langle T_j, \text{Commit} \rangle$  record then Undo( $T_j$ )

## Example

|            |                   |                         |
|------------|-------------------|-------------------------|
| Xact       | Log File          | DB(A=1000,B=2000,C=500) |
| T0 start   | <T0,start>        |                         |
| Read(A,a)  |                   |                         |
| A=a-50     |                   |                         |
| Write(A,a) | <T0,A,1000,950>   | A=950,B=2000,C=500      |
| Read(B,b)  |                   |                         |
| B=b+50     |                   |                         |
| Write(B,b) | <T0,B,2000,2050>  | A=950,B=2050,C=500      |
| End T0     | <T0, Commit>      | A=950,B=2050,C=500      |
| T1 Start   | <T1, Start>       | A=950,B=2050,C=500      |
| Read(C,c)  |                   |                         |
| c=c-100    |                   |                         |
| Write(C,c) | <T1, C, 500, 400> | A=950,B=2050,C=400      |
| End T1     | <T1, Commit>      | A=950,B=2050,C=400      |

## Recovery : Immediate Modification

- ❖ Where does failure occur and what action to take (i.e., redo or undo)
- ❖ Example:
  - When <T0,Start> but no <T0,Commit>
    - ◆ Undo(T0)
  - When <T0,Start> and <T0,Commit>, and <T1,start> but no <T1,Commit>
    - ◆ Redo(T0) , Undo (T1)
  - When both T0 and T1 have start and commit records
    - ◆ Redo(T0) and Redo(T1)

## Checkpoints

- ❖ Recovery system consults log file when failure occurs
  - Search entries in log file
    - ◆ Time depends on number of entries
    - ◆ Why redo Xacts that have already been written to disk
- ❖ Introduce checkpoint records into log file
  - A <checkpoint> record written into log file when system writes out to database
  - All Xacts after last <checkpoint> need to be redone (redo or undo) after a failure
  - To take checkpoint: suspend execution, output all log records to stable storage, output all modified blocks
    - ◆ Need to be careful about when to take checkpoints (system backup time)

## Concurrency in Transaction Processing

## Recall: Transactions

- ❖ Basic concurrency/recovery concept: a *transaction* (*Xact*).
  - A sequence of many actions which are considered to be one atomic unit of work.
- ❖ DBMS “actions”:
  - reads, writes
  - Special actions: commit, abort
  - for now, assume reads and writes are on tuples; we'll revisit this assumption later.
  - Model Xact operations only as Read R(A) and Write W(A)

## The ACID Properties

- ❖ **A**tomicity: All actions in the Xact happen, or none happen.
- ❖ **C**onsistency: If each Xact is consistent, and the DB starts consistent, it ends up consistent.
- ❖ **I**solation: Execution of one Xact is isolated from that of other Xacts.
- ❖ **D**urability: If a Xact commits, its effects persist.

## Concurrency: Schedules

- ❖ Scheduler is a program that controls concurrency execution of Xacts; it produces execution sequence for a set of Xacts
  - **Schedule**
- ❖ Schedule must preserve instruction execution order
- ❖ **Serial schedule** is when transactions are executed sequentially

## 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*A, | B=1.06*B |

- ❖ This is OK. But what about:

|     |                    |         |
|-----|--------------------|---------|
| T1: | A=A+100,           | B=B-100 |
| T2: | A=1.06*A, B=1.06*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) |            |

## Correct Schedules ?

- ❖ Interleaving of instructions can lead to incorrect results
- ❖ Serializability theory - What is a correct schedule ?
  - Easier to reason about serial executions than concurrent schedules
  - If results of a schedule are identical to results of a serial schedule then ?
- ❖ Note: we need only focus on Read and Write operations when modeling concurrency

## Schedules

- ❖ **Schedule**: An interleaving of actions from a set of Xacts, where the actions of any 1 Xact are in the original order.
  - Represents some actual sequence of database actions.
  - Example:  $R_1(A), W_1(A), R_2(B), W_2(B), R_1(C), W_1(C)$
  - In a *complete* schedule, each Xact ends in *commit* or *abort*.
- ❖ Initial State + Schedule → Final State

| <u>T1</u> | <u>T2</u> |
|-----------|-----------|
| R(A)      |           |
| W(A)      |           |
|           | R(B)      |
|           | W(B)      |
| R(C)      |           |
| W(C)      |           |

## Acceptable Schedules

- ❖ One sensible "isolated, consistent" schedule:
  - Run Xacts one at a time, in a series.
  - This is called a *serial* schedule.
  - **NOTE**: Different serial schedules can have different final states; all are "OK" -- DBMS makes no guarantees about the order in which concurrently submitted Xacts are executed.
- ❖ Serial schedule always produces correct results
  - How can we use this to construct a concurrency theory to prove correctness ?

## Serializable Schedules

- ❖ **Serializable** schedules:
  - Final state is what *some* serial schedule would have produced.
  - Aborted Xacts are not part of schedule; ignore them for now (they are made to 'disappear' by using logging).
- ❖ **Serializable schedule produces correct results**

## Serializability Violations

- ❖ Two actions **conflict** when 2 Xacts access the same item:
  - **W-R conflict**: T2 reads something T1 wrote.
  - **R-W and W-W conflicts**: Similar.
- ❖ **WR conflict (dirty read)**:
  - Result is not equal to any serial execution!
- ❖ **RR Conflict** - is this a problem?

Database is inconsistent!

|           | transfer \$100 from A to B | add 6% interest to A & B |
|-----------|----------------------------|--------------------------|
| <b>T1</b> |                            |                          |
| R(A)      |                            |                          |
| W(A)      |                            |                          |
|           |                            | R(A)                     |
|           |                            | W(A)                     |
|           |                            | R(B)                     |
|           |                            | W(B)                     |
|           |                            | Commit                   |
| R(B)      |                            |                          |
| W(B)      |                            |                          |
| Commit    |                            |                          |
| <b>T2</b> |                            |                          |

## More Conflicts

- ❖ **RW Conflicts (Unrepeatable Read)**
  - T2 overwrites what T1 read.
  - If T1 reads it again, it will see something new!
    - ◆ Example when this would happen?
    - ◆ The increment/decrement example.
  - Again, not equivalent to a serial execution.
- ❖ **WW Conflicts (Overwriting Uncommitted Data)**
  - T2 overwrites what T1 wrote.
    - ◆ Example: 2 Xacts to update items to be kept equal.
  - Usually occurs in conjunction w/other anomalies.
    - ◆ Unless you have "blind writes".

## Serializability - Summary

- ❖ If two Xacts T1, T2 refer to same data item Q then need to consider order of Read, Write
- ❖ Two Xacts conflict if they operate on same data and at least one of them is a Write
- ❖ Can a schedule be transformed to a serial schedule?
  - See if you can swap Reads and Writes of different Xacts without changing result, until you get a serial schedule

## Now, Aborted Transactions

- ❖ **Serializable schedule**: Equivalent to a serial schedule of *committed* Xacts.
  - as if aborted Xacts *never happened*.
- ❖ Two Issues:
  - How does one undo the effects of an xact?
    - ◆ Covered by logging/recovery process
  - What if another Xact sees these effects??
    - ◆ Must undo that Xact as well!

## Cascading Aborts

- ❖ Abort of T1 requires abort of T2!
  - **Cascading Abort**
- ❖ What about **WW conflicts** & aborts?
  - T2 overwrites a value that T1 writes.
  - T1 aborts: its "remembered" values are restored.
  - Lose T2's write! We will see how to solve this, too.
- ❖ An **ACA (avoids cascading abort)** schedule is one in which cascading abort cannot arise.
  - A Xact only reads/writes data from committed Xacts.

| <u>T1</u> | <u>T2</u> |
|-----------|-----------|
| R(A)      |           |
| W(A)      |           |
|           | R(A)      |
|           | W(A)      |
| abort     |           |

## Recoverable Schedules

- ❖ Abort of T1 requires abort of T2!
  - But T2 has already committed!
- ❖ A **recoverable** schedule is one in which this cannot happen.
  - i.e. a Xact commits only after all the Xacts it "depends on" (i.e. it reads from or overwrites) commit.
  - Recoverable implies ACA (but not vice-versa!).
    - ◆ Find example ...
- ❖ Real systems typically ensure that only recoverable schedules arise (through **locking**).

| <u>T1</u> | <u>T2</u> |
|-----------|-----------|
| R(A)      |           |
| W(A)      |           |
|           | R(A)      |
|           | W(A)      |
|           | commit    |
| abort     |           |

## Recovery and Concurrency

- ❖ We studied log based recovery
  - Always write first into log-file before writing into database
  - Guarantees Atomicity and Durability
    - ◆ We can recover from failures
- ❖ When can we use the same log-based recovery method when we have concurrency
  - Ask: What type of schedules should be allowed?

## Recap: Scheduling Concurrent Transactions

- ❖ Need to ensure ACID properties
- ❖ Schedule must produce results equivalent to a serial schedule (avoid conflicting R/W)
  - Guarantee serializability
- ❖ Must be able to recover from a failure
  - Guarantee recoverability and avoid cascading aborts
- ❖ What type of schedules ??

## Lock Based Protocols

- ❖ Conflict occurs when two Xacts try to access the same data item
- ❖ Associate a "lock" for each shared data item
  - Similar to mutual exclusion
  - To access a data item, check if it is unlocked else wait
  - Need to worry about the type of operation: Read or Write
    - ◆ Leads to **Lock Modes**: **Shared Lock(S)** for Reads only and **Exclusive Lock(X)** for Writes

## Locks

T1:start  
 lock(A)  
 Read(A)  
 Unlock(A)  
 lock(B)  
 Write(B)  
 Unlock(B)  
 commit

## Scheduling with Locks

| T1:       | T2        | Lock/Concurrency manager |
|-----------|-----------|--------------------------|
| start     |           |                          |
| lock(A)   |           | grant lock               |
| Read(A)   |           |                          |
|           | Start     |                          |
|           | lock(B)   | grant lock               |
|           | Write(B)  |                          |
|           | unlock(B) | unlock                   |
| Unlock(A) |           | unlock                   |
| lock(B)   |           | grant lock               |
|           | lock(A)   | grant lock               |
|           | Write(A)  |                          |
|           | unlock(A) | unlock                   |
|           | commit    |                          |
| Write(B)  |           |                          |
| Unlock(B) |           |                          |
| commit    |           |                          |

## Locking: A Technique for C. C.

- ❖ Concurrency control usually done via **locking**.
- ❖ Lock info maintained by a **"lock manager"**:
  - Stores (XID, RID, Mode) triples.
    - ◆ This is a simplistic view; suffices for now.
  - Mode  $\in \{S, X\}$  (shared/Exclusive)
  - Lock compatibility table:
- ❖ If a Xact can't get a lock, it is suspended on a **wait queue**.

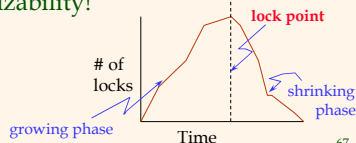
|    |    |   |   |
|----|----|---|---|
|    | -- | S | X |
| -- | ✓  | ✓ | ✓ |
| S  | ✓  | ✓ |   |
| X  | ✓  |   |   |

## Lock Based protocols

- ❖ Using locks as the primitive, construct protocols for using the locks
- ❖ Lock protocol must result in schedules that ensure
  - Correctness, i.e., serializability
  - recoverability

## Two-Phase Locking (2PL)

- ❖ **2PL**:
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - If T releases any lock, it can acquire **no new locks!**
- ❖ Locks are automatically obtained by DBMS.
- ❖ **Guarantees serializability!**
  - Why?



## 2PL- Example

|           |           |                     |
|-----------|-----------|---------------------|
| T1:       | T2        | Concurrency manager |
| start     |           |                     |
| lock(A)   |           | Grant lock          |
|           | Start     |                     |
|           | lock(B)   | Grant lock          |
|           | unlock(B) | unlock B            |
| lock(B)   |           | grant lock          |
| Unlock(A) |           | unlock              |
|           | lock(A)   |                     |
|           | Write(A)  |                     |
|           | unlock(A) | unlock              |
|           | commit    |                     |
| Write(B)  |           |                     |
| Unlock(B) |           | unlock              |
| commit    |           |                     |

## 2PL- Example

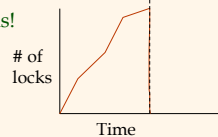
|           |           |                             |
|-----------|-----------|-----------------------------|
| T1:       | T2        | Concurrency manager         |
| start     |           | Grant lock                  |
| lock(A)   | Start     |                             |
|           | lock(A)   | cannot Grant lock/ T2 waits |
|           | unlock(A) |                             |
| lock(B)   |           |                             |
| Unlock(A) | lock(A)   |                             |
|           | Write(A)  |                             |
|           | unlock(A) |                             |
|           | commit    |                             |
| Write(B)  |           |                             |
| Unlock(B) |           |                             |
| commit    |           |                             |

## 2PL – Example: What can go wrong

|           |           |                     |
|-----------|-----------|---------------------|
| T1        | T2        | Concurrency Manager |
| Start     |           | grant lock          |
| X-lock(A) |           |                     |
| S-lock(B) |           |                     |
| Write(A)  |           |                     |
| Unlock(A) |           | unlocks A           |
|           | start     |                     |
|           | lock(A)   | grant lock          |
|           | Write(A)  |                     |
|           | unlock(A) |                     |
|           | commit    |                     |
| Unlock(B) |           |                     |
| commit    |           |                     |

## Strict 2PL

- ❖ **Strict 2PL:**
  - If T wants to read an object, first obtains an S lock.
  - If T wants to modify an object, first obtains X lock.
  - **Hold all locks until end of transaction.**
- ❖ Guarantees serializability, and **recoverable schedule**, too!
  - also avoids WW problems!



## Strict 2PL- Example

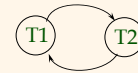
|           |           |                         |
|-----------|-----------|-------------------------|
| T1        | T2        | Concurrency Manager     |
| Start     |           | grant lock              |
| X-lock(A) |           |                         |
| S-lock(B) |           |                         |
| Write(A)  |           |                         |
|           | start     |                         |
|           | lock(A)   | cannot grant lock /wait |
|           | Write(A)  |                         |
|           | unlock(A) |                         |
|           | commit    |                         |
| Unlock(A) |           |                         |
| Unlock(B) |           |                         |
| commit    |           |                         |

## Strict 2PL-Example

| T1        | T2        | Concurrency Manager |
|-----------|-----------|---------------------|
| Start     |           |                     |
| X-lock(A) |           | grant lock          |
| S-lock(B) |           | grant lock          |
| Write(A)  |           |                     |
| Unlock(A) |           |                     |
| Unlock(B) |           | unlock all          |
| Commit    |           |                     |
|           | start     |                     |
|           | lock(A)   | grant lock          |
|           | Write(A)  |                     |
|           | unlock(A) |                     |
|           | commit    | unlock all          |

## Precedence Graph

- ❖ A **Precedence (or Serializability) graph**:
  - Node for each committed Xact.
  - Arc from  $T_i$  to  $T_j$  if an action of  $T_i$  precedes and conflicts with an action of  $T_j$ .
- ❖ T1 transfers \$100 from A to B, T2 adds 6%
  - $R_1(A), W_1(A), R_2(A), W_2(A), R_2(B), W_2(B), R_1(B), W_1(B)$



## Conflict Serializability

- ❖ 2 schedules are **conflict equivalent** if:
  - they have the same sets of actions, and
  - each pair of conflicting actions is ordered in the same way.
- ❖ A schedule is **conflict serializable** if it is conflict equivalent to a serial schedule.
  - **Note:** Some serializable schedules are not conflict serializable!

## Conflict Serializability & Graphs

- ❖ **Theorem:** A schedule is conflict serializable iff its precedence graph is acyclic.
- ❖ **Theorem:** 2PL ensures that the precedence graph will be acyclic!
- ❖ **Strict 2PL** improves on this by avoiding cascading aborts, problems with undoing WW conflicts; i.e., ensuring recoverable schedules.

## Locks – what else can go wrong?

|           |           |                        |
|-----------|-----------|------------------------|
| T1:       | T2        | Concurrency manager    |
| start     |           | Grant lock             |
| lock(A)   | Start     | Grant lock             |
|           | lock(B)   | cannot grant/ T1 waits |
| lock(B)   |           | cannot grant/T2 waits  |
|           | lock(A)   |                        |
|           | Write(A)  |                        |
|           | unlock(A) | unlock                 |
|           | unlock(B) |                        |
|           | commit    |                        |
| Write(B)  |           |                        |
| Unlock(B) |           |                        |
| Unlock(A) |           |                        |
| commit    |           |                        |

## Deadlocks

- ❖ Lock protocols can result in deadlocks
  - Transactions waiting for each other to release locks
- ❖ Deadlocks lead to suspended processes
- ❖ How to deal with deadlocks
  - Prevent deadlocks
  - Detect and recover
  - Periodically check for deadlocks

## Deadlock Prevention

X1(A), X2(B), S1(B), S2(A)

- ❖ Assign a timestamp to each Xact as it enters the system. “Older” Xacts have priority.
- ❖ Assume Ti requests a lock, but Tj holds a conflicting lock.
  - **Wait-Die:** If Ti has higher priority, it waits; else Ti aborts.
  - **Wound-Wait:** If Ti has higher priority, abort Tj; else Ti waits.
  - **Note:** After abort, restart with original timestamp!
  - Both guarantee deadlock-free behavior! Pros and cons of each?

## An Alternative to Prevention

- ❖ In theory, deadlock can involve many transactions:
  - T1 waits-for T2 waits-for T3 ...waits-for T1
- ❖ In practice, most “deadlock cycles” involve only 2 transactions.
- ❖ Don’t need to prevent deadlock!
  - What’s the problem with prevention?
- ❖ Allow it to happen, then notice it and fix it.
  - Deadlock detection.

## Deadlock Detection

- ❖ Lock Mgr maintains a “Waits-for” graph:
  - Node for each Xact.
  - Arc from  $T_i$  to  $T_j$  if  $T_j$  holds a lock and  $T_i$  is waiting for it.
- ❖ Periodically check graph for cycles.
- ❖ “Shoot” some Xact to break the cycle.

To lock such rascal counters from his friends,  
Be ready, gods, with all your thunderbolts:  
Dash him to pieces!  
-- Shakespeare, *Julius Caesar*

## Deadlock Detection

- ❖ Simpler hack: *time-outs*.
  - $T_1$  made no progress for a while? Shoot it.

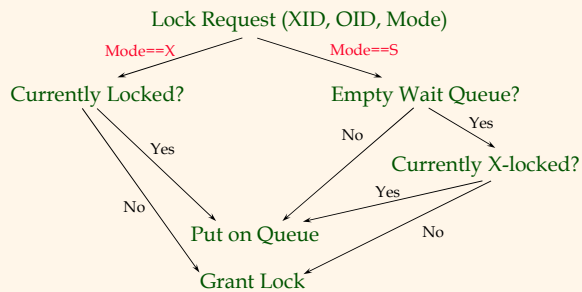
## Prevention vs. Detection

- ❖ Prevention might abort too many Xacts.
- ❖ Detection might allow deadlocks to tie up resources for a while.
  - Can detect more often, but it's time-consuming.
- ❖ The usual answer:
  - Detection is the winner.
  - Deadlocks are pretty rare.
  - If you get a lot of deadlocks, reconsider your schema/workload!

## Lock Manager Implementation

- ❖ **Question 1:** What are we locking?
  - Tuples, pages, or tables?
  - Finer granularity increases concurrency, but also increases locking overhead.
- ❖ **Question 2:** How do you “lock” something??
- ❖ **Lock Table:** A hash table of Lock Entries.
  - *Lock Entry:*
    - ◆ OID
    - ◆ Mode
    - ◆ List: Xacts holding lock
    - ◆ List: Wait Queue

## Handling a Lock Request



Database Management Systems

85

## More Lock Manager Logic

- ❖ On lock release (OID, XID):
  - Update list of Xacts holding lock.
  - Examine head of wait queue.
  - If Xact there can run, add it to list of Xacts holding lock (change mode as needed).
  - Repeat until head of wait queue cannot be run.
- ❖ **Note:** Lock request handled atomically!
  - via **latches** (i.e. semaphores/mutex; OS stuff).

Database Management Systems

86

## Lock Upgrades

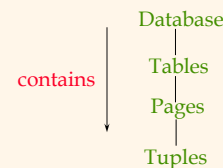
- ❖ Think about this scenario:
  - T1 locks A in S mode, T2 requests X lock on A, T3 requests S lock on A. *What should we do?*
- ❖ In contrast:
  - T1 locks A in S mode, T2 requests X lock on A, T1 requests X lock on A. *What should we do?*
- ❖ Allow such **upgrades** to supersede lock requests.
  - Consider this scenario:
    - ◆ S1(A), X2(A), X1(A): **DEADLOCK!**
- ❖ BTW: Deadlock can occur even w/o upgrades:
  - X1(A), X2(B), S1(B), S2(A)

Database Management Systems

87

## Multiple-Granularity Locks

- ❖ Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- ❖ Shouldn't have to decide!
- ❖ Data "containers" are nested:



Database Management Systems

88

## *Summary of C.C.*

- ❖ Concurrency control key to a DBMS.
  - More than just mutexes!
- ❖ Transactions and the ACID properties:
  - C & I are handled by concurrency control.
  - A & D coming soon with logging & recovery.
- ❖ Conflicts arise when two Xacts access the same object, and one of the Xacts is modifying it.
- ❖ Serial execution is our model of correctness.

## *Summary, cont.*

- ❖ Serializability allows us to “simulate” serial execution with better performance.
- ❖ 2PL: A simple mechanism to get serializability.
  - Strict 2PL also gives us recoverability.
- ❖ Lock manager module automates 2PL so that only the access methods worry about it.
  - Lock table is a big main-mem hash table
- ❖ Deadlocks are possible, and typically a deadlock detector is used to solve the problem.

## *Summary of Basic DBMS Architecture*

- ❖ Logical Level
  - Relational Model: formal languages, SQL, Shema design
- ❖ Physical Design
  - Index structures and file organization
  - Query processing - links logical and physical design
- ❖ Concurrency and Recovery process
  - Log based recovery, transaction support