Locking and Logging

With slides by Ph.Bonnet, J.Ulmann, M.Kifer

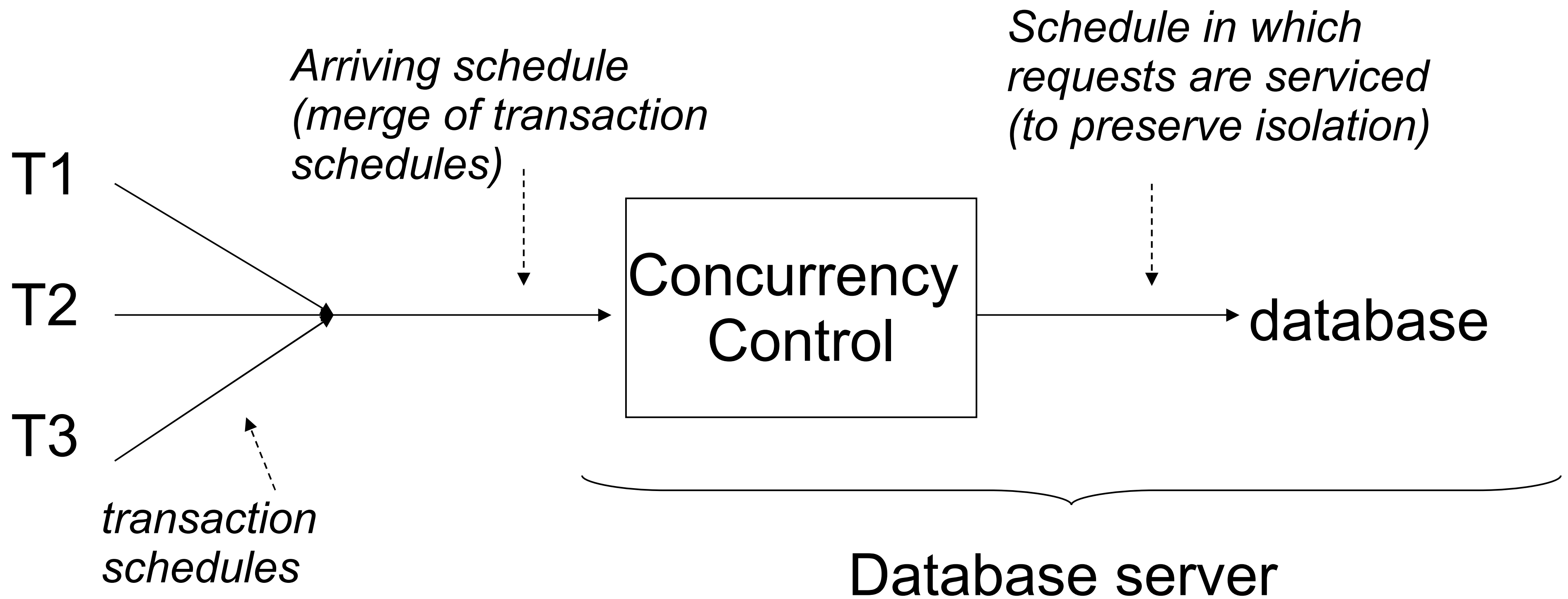
Outline

- Concurrency Control: Implementing Isolation
  - Scheduling
  - Locking
  - Isolation Levels
- Logging: Implementing Atomicity and Durability
  - Buffer Management 101
  - Undo / Redo
  - Write Ahead Logging

Isolation

- Serial execution:
  - Since each transaction is consistent and isolated from all others, schedule is guaranteed to be correct for all applications
  - Inadequate performance
    - Since system has multiple asynchronous resources and transaction uses only one at a time
- Concurrent execution:
  - Improved performance (multiprogramming)
  - Some interleavings produce correct result, others do not (anomalies)
  - We are interested in concurrent schedules that are *equivalent* to serial schedules. These are referred to as *serializable* schedules.

Schedule



Concurrency Control

- Transforms arriving interleaved schedule into a correct interleaved schedule to be submitted to the DBMS
  - Delays servicing a request (reordering) - causes a transaction to wait
  - Refuses to service a request - causes transaction to abort
- Actions taken by concurrency control have performance costs
  - Goal is to avoid delaying or refusing to service a request

Correct Schedules

- Interleaved schedules *equivalent* to serial schedules are the only ones guaranteed to be correct for *all* applications
- Equivalence based on *commutativity* of operations
- **Definition:** Database operations p_1 and p_2 commute if, *for all initial database states*, they (1) *return the same results* and (2) *leave the database in the same final state* when executed in either order.

$p_1 \; p_2 \qquad p_2 \; p_1$

Conventional Operations

- Read
  - *r(x, X)* - copy the value of database variable *x* to local variable *X*
- Write
  - *w(x, X)* - copy the value of local variable *X* to database variable *x*
- We use $r_1(x)$ and $w_1(x)$ to mean a read or write of *x* by transaction T_1

Commutativity of Read and Write

- p_1 commutes with p_2 if
  - They operate on different data items
    - $w_1(x)$ commutes with $w_2(y)$ and $r_2(y)$
  - Both are reads
    - $r_1(x)$ commutes with $r_2(x)$
- Operations that do not commute *conflict*
    - $w_1(x)$ conflicts with $w_2(x)$
    - $w_1(x)$ conflicts with $r_2(x)$

Concurrency Control

- Concurrency control cannot see entire schedule:
  - It sees one request at a time and must decide whether to allow it to be serviced
- Strategy: Do not service a request if:
  - It violates strictness or serializability, or
  - There is a possibility that a subsequent arrival might cause a violation of serializability

Pessimistic Concurrency Control

Idea:

Force conflicts

Rule:

Do not grant a request that imposes an ordering among *active* transactions (*delay* the requesting transaction)

Grant a request that does not conflict with previously granted requests of *active* transactions

Rule can be used as each request arrives

If a transaction's request is delayed, it is forced to wait (but the transaction is still considered active)

Delayed requests are reconsidered when a transaction completes (aborts or commits) since it becomes inactive

Locking

- A transaction can read a database item if it holds a read (shared) lock on the item
- It can read *or* update the item if it holds a write (exclusive) lock
- If the transaction does not already hold the required lock, a lock request is automatically made as part of the (read or write) request

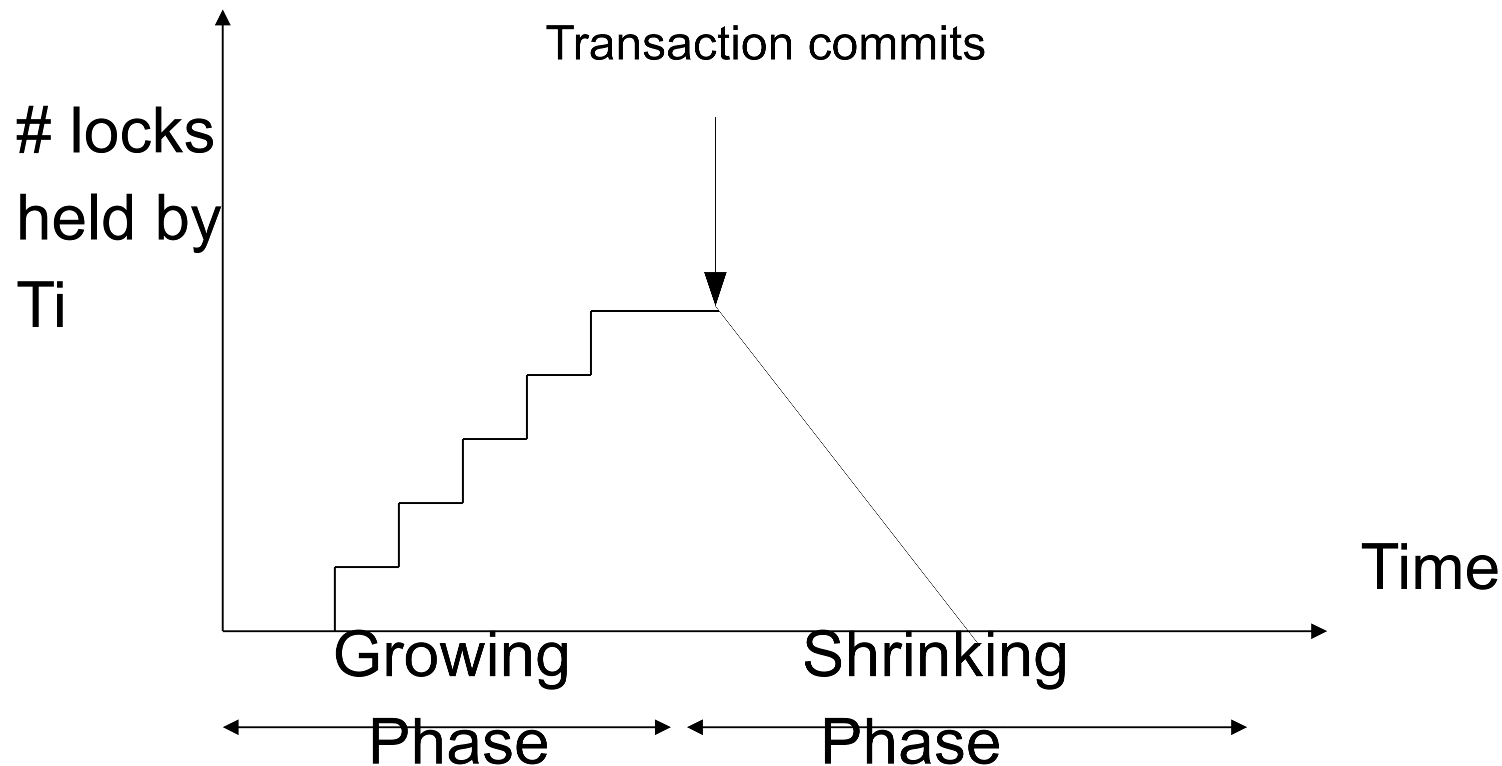
Locking

- Request for read lock on an item is granted if no transaction currently holds write lock on the item
  - Cannot read an item written by an active transaction
- Request for write lock granted if no transaction holds any lock on item
  - Cannot write an item read/written by an active transaction
- Transaction is delayed if request cannot be granted

Requested mode	Granted mode *read*	Granted mode *write*
read		x
write	x	x

2 Phase Locking

- All locks held by a transaction are released when the transaction completes (commits or aborts)



2 Phase Locking

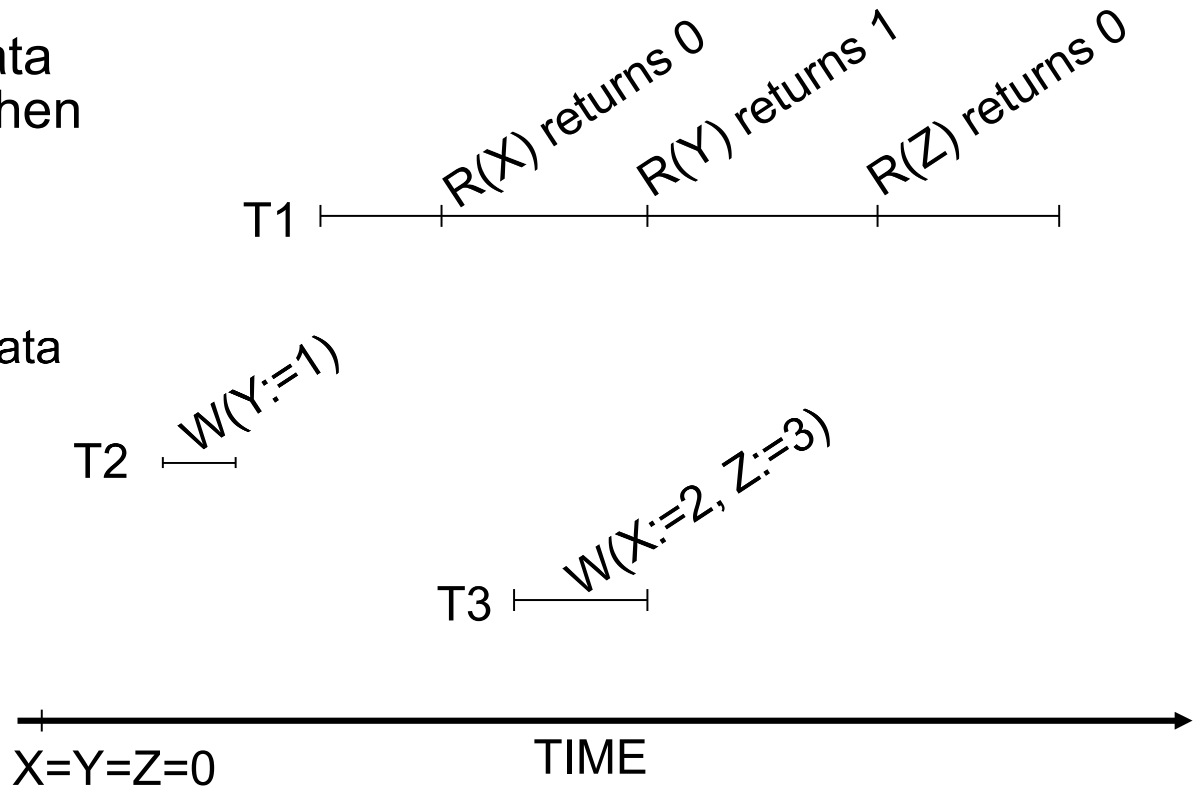
- **Result**: A lock is not granted if the requested access conflicts with a prior access of an active transaction; instead the transaction waits. This enforces the rule:
  - Do not grant a request that imposes an ordering among active transactions (delay the requesting transaction)
- Resulting schedules are serializable and strict

Isolation Levels

- Read Uncomitted (No lost update)
  - Exclusive locks for write operations are held for the duration of the transactions
  - No locks for read
- Read Committed (No inconsistent retrieval)
  - Exclusive locks for write operations are held for the duration of the transactions.
  - Shared locks are released as soon as the read operation terminates.
- Repeatable Read (no unrepeatable reads)
  - Strict two phase locking: all locks are held until the transaction commits.
- Serializable (no phantoms)
  - Strict two phase locking
  - Table locking or index locking to avoid phantoms

Snapshot isolation

- Each transaction executes against the version of the data items that was committed when the transaction started:
  - No locks for read
  - Locks for writes
  - Costs space (old copy of data must be kept)
- Almost serializable level:
  - T1: x:=y
  - T2: y:= x
  - Initially x=3 and y =17
  - Serial execution: x,y=17 or x,y=3
  - Snapshot isolation: x=17, y=3 if both transactions start at the same time.



Outline

- Concurrency Control: Implementing Isolation
  - Scheduling
  - Locking
  - Isolation Levels
- Logging: Implementing Atomicity and Durability
  - Buffer Management 101
  - Undo / Redo
  - Write Ahead Logging

Handling the Buffer Pool

	No Steal	Steal
Force	Trivial	
No Force		Desired

Handling the Buffer Pool

	No Steal	Steal
Force		Undo
No Force	Redo	Undo Redo

Logging

- REDO and UNDO information in a *log.*
  - Sequential writes to log (put it on a separate disk).
  - Minimal info written to log, so multiple updates fit in a single log page.
- <u>Log</u>: An ordered list of REDO/UNDO actions
  - Log record contains:

    <XID, pageID, offset, length, old data, new data>
  - and additional control info

Current database state = current state of data on disks + log

Write-Ahead Logging (WAL)

The Write-Ahead Logging Protocol:

① Must force the log record for an update *before* the corresponding data page gets to disk.

Guarantees Atomicity

❍ Must write all log records for a Xact *before commit*.

Guarantees Durability

ARIES algorithms, developed by C.Mohan at IBM Almaden in the early 90's

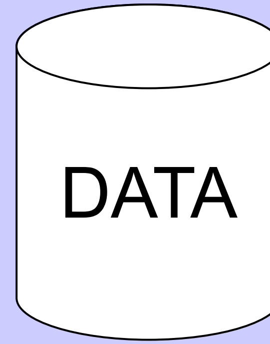
http://www.almaden.ibm.com/u/mohan/ARIES_Impact.html

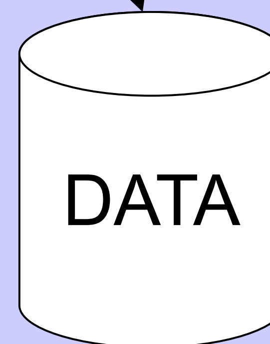
UNSTABLE STORAGE

DATABASE BUFFER

Pi

Pj





STABLE STORAGE

Take Away Points

- Locks are data structures introduced to force conflicts between operations so that the scheduler can take decisions
- Isolation levels impact correctness
- Log used to store before/after images
- Write ahead logging garantees durability and atomicity