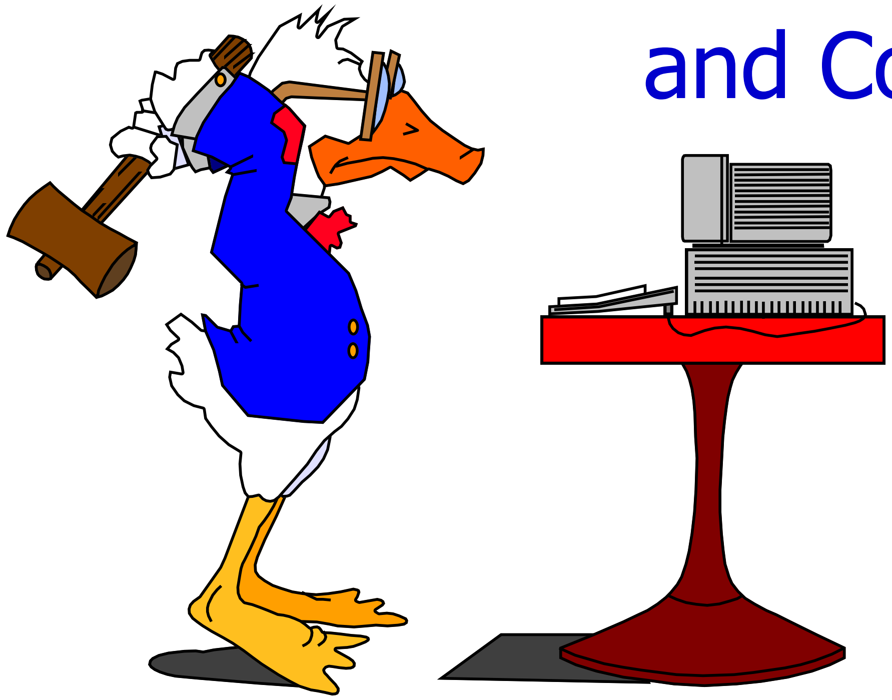
Transaction Overview and Concurrency Control



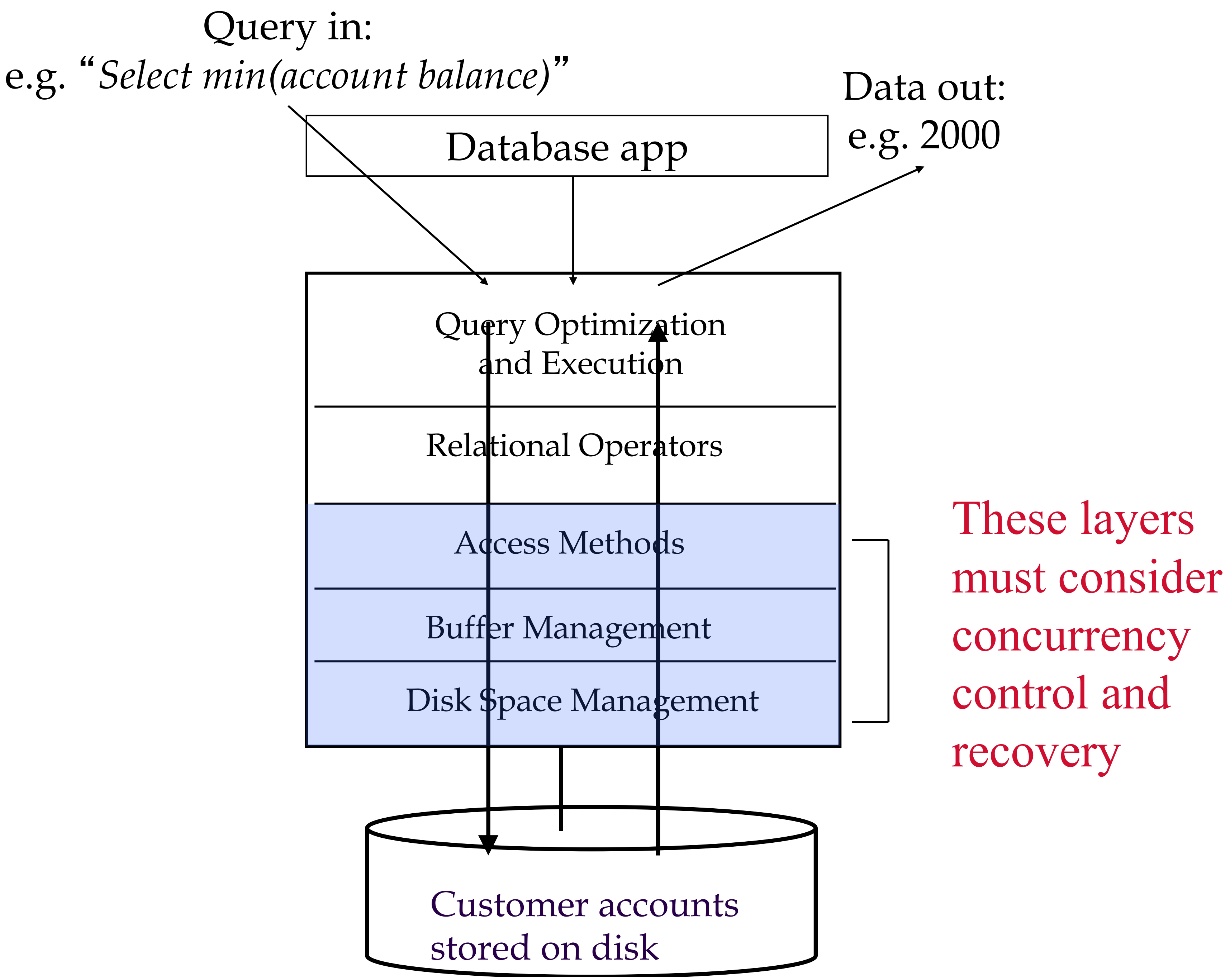
CSC 375

Fall 2017,

Chapters 16 & 17

There are three side effects of acid.
Enhanced long term memory,
decreased short term memory,
and I forget the third.
- Timothy Leary

Structure of a DBMS


Query in:
e.g. "*Select min(account balance)*"
Data out:
e.g. 2000
Database app
Query Optimization and Execution
Relational Operators
Access Methods
Buffer Management
Disk Space Management
Customer accounts stored on disk
These layers must consider concurrency control and recovery

Transactions

- **Concurrent execution essential for good performance.**
  - Because disk accesses are frequent, and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently.
  - Trends are towards lots of cores and lots of disks.
    - e.g., IBM Watson has 2880 processing cores
    - Tegra K1, latest GPU from NVidia, has 192 processing cores
- **A program may carry out many operations, 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.**

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.**
- **Atomic Transactions: a user can think of a transaction as always either executing all its actions, or not executing any actions at all.**
    - One approach: DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.
    - Another approach: *Shadow Pages*
    - Logs approach won because of need for audit trail and for efficiency reasons.

Transactions in SQL

- **Commit makes permanent any database changes you made during the current transaction. Until you commit your changes, other users cannot see them.**
- **Rollback ends the current transaction and undoes any changes made since the transaction began.**

Transaction – Example

```
BEGIN;     --BEGIN TRANSACTION
   UPDATE accounts SET balance = balance -
     100.00 WHERE name = 'Alice';

   UPDATE branches SET balance = balance -
     100.00 WHERE name = (SELECT branch_name
     FROM accounts WHERE name = 'Alice');

   UPDATE accounts SET balance = balance +
     100.00 WHERE name = 'Bob';

   UPDATE branches SET balance = balance +
     100.00 WHERE name = (SELECT branch_name
     FROM accounts WHERE name = 'Bob');

COMMIT;     --COMMIT WORK
```

Transaction Example (with Savepoint)

```
BEGIN;
UPDATE accounts SET balance = balance - 100.00
    WHERE name = 'Alice';

SAVEPOINT my_savepoint;
UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Bob’;

-- oops ... forget that, and use Wally's
Account ROLLBACK TO my_savepoint;

UPDATE accounts SET balance = balance + 100.00
    WHERE name = 'Wally';
COMMIT;
```

Transactions in MySQL

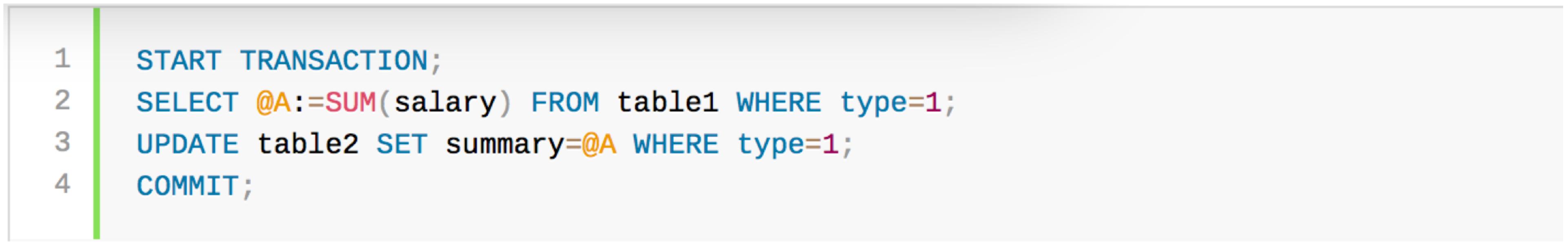
```
START TRANSACTION
    [transaction_characteristic [, transaction_characteristic] ...]

transaction_characteristic:
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY

BEGIN [WORK]
COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE]
SET autocommit = {0 | 1}
```

- START TRANSACTION or BEGIN
  - start a new transaction.
- COMMIT
  - commits the current transaction, making its changes permanent.
- ROLLBACK
  - rolls back the current transaction, canceling its changes.
- SET autocommit
  - disables or enables the default autocommit mode for the current session.

Transactions in MySQL



```sql
START TRANSACTION;
SELECT @A:=SUM(salary) FROM table1 WHERE type=1;
UPDATE table2 SET summary=@A WHERE type=1;
COMMIT;
```

- To disable `autocommit` mode implicitly for a single series of statements, use the START TRANSACTION statement

Transactions in MySQL

- See https://dev.mysql.com/doc/refman/5.7/en/commit.html

The ACID properties of Transactions

- **Atomicity:** All actions in the transaction happen, or none happen.
- **Consistency:** If each transaction is consistent, and the DB starts consistent, it ends up consistent.
- **Isolation:** Execution of one transaction is isolated from that of all others.
- **Durability:** If a transaction commits, its effects persist.

Transaction Consistency

- **Data in DBMS is accurate in modeling real world, follows integrity constraints**
  - User must ensure transaction consistent by itself
  - If DBMS is consistent before transaction, it will be after also.
- **System checks ICs and if they fail, the transaction rolls back (i.e., is aborted).**
  - DBMS enforces some ICs, depending on the ICs declared in CREATE TABLE statements.
  - Beyond this, DBMS does not understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).

Isolation (Concurrency)

- **Multiple users can submit transactions.**
- **Each transaction executes as if it was running by itself.**
  - Concurrency is achieved by DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
- **Many techniques have been developed. Fall into two basic categories:**
  - Pessimistic – don't let problems arise in the first place
  - Optimistic – assume conflicts are rare, deal with them *after* they happen.

Durability - Recovering From a Crash

- System Crash **- short-term memory lost (disk okay)**
  - This is the case we will handle.
- Disk Crash **- “stable” data lost**
  - ouch --- need back ups; raid-techniques can help avoid this.
- **There are 3 phases in ARIES recovery (and most others):**
    - ARIES = Algorithms for Recovery and Isolation Exploiting Semantics
  - *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.
  - *Undo*: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, as found in the log), working backwards in the log.
- **At the end, all committed updates and only those updates are reflected in the database.**
  - Some care must be taken to handle the case of a crash occurring during the recovery process!

Plan of attack (ACID properties)

- **First we will deal with "I", by focusing on concurrency control.**

- **Then will address "A" and "D" by looking at recovery.**

- **What about "C"?**
- Well, if you have the other three working, and you set up your integrity constraints correctly, then you get this for free (!?).

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
```

Transfer $100 from B's account to A's account

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.**
  - Assume at first A and B each have $1000. What are the legal outcomes of running T1 and T2???
    - $2000 *1.06 = $2120

Example (Contd.)

- **Legal outcomes: A=1166,B=954 or A=1160,B=960**
- **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:

The $100 transfer amount is given interest payment twice

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

- **Result: A=1166, B=960; A+B = 2126, bank loses $6**

❖ 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:*** **A schedule that does not interleave the actions of different transactions.**
  - i.e., you run the transactions serially (one at a time)

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

- ***Serializable schedule*:** **A serializable schedule is a schedule that is equivalent to some serial execution of the transactions.**
  - Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time.

Schedules

Schedules represent the chronological order in which instructions are executed in the system.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

A serial schedule and an equivalent interleaved one

Characterizing Schedules Based on Serializability

- **A serializable schedule is <u>not</u> the same as a serial schedule**
- **Being serializable implies that the schedule is a <u>correct</u> schedule.**
  - It will leave the database in a consistent state.
  - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.
- **Serializability is hard to check.**
  - Interleaving of operations occurs in an operating system through some scheduler
  - Difficult to determine beforehand how the operations in a schedule will be interleaved.

Concurrency Control

- **Concurrent execution of user programs is essential for good DBMS performance.**
  - Since disk accesses are frequent and relatively slow, it is important to keep the CPU humming by working on several user programs concurrently.
- **Interleaving actions of different user programs can lead to inconsistency: e.g., check is cleared while account balance is being computed.**
- **Transaction processing ensures such problems do not arise: users can pretend they are using a single-user system.**

Anomalies with Interleaved Execution

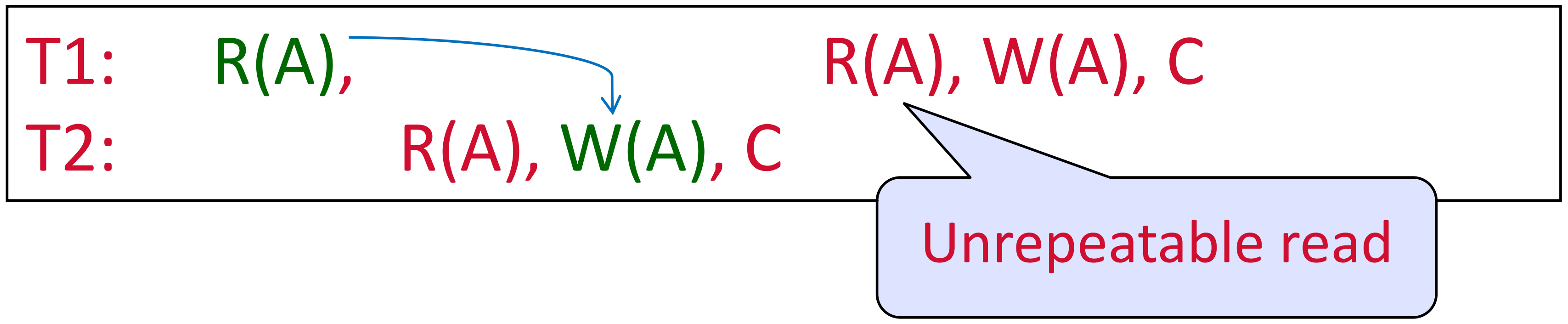
- **Reading Uncommitted Data (WR Conflicts, "dirty reads"):**

$$\begin{tikzcd}
\text{T1:} & \text{R(A), W(A),} \arrow[dr] & & \text{R(B), W(B), Abort} \\
\text{T2:} & & \text{R(A), W(A), C} &
\end{tikzcd}$$

- **Problem**
  - T1 may write some value into A that makes the database inconsistent
  - As long as T1 overwrites this value with a 'correct' value of A before committing, no harm is done if T1 and T2 run in some serial order, because T2 would then not see the (temporary) inconsistency
  - On the other hand, interleaved execution can expose this inconsistency and lead to an inconsistent final database state

Anomalies with Interleaved Execution

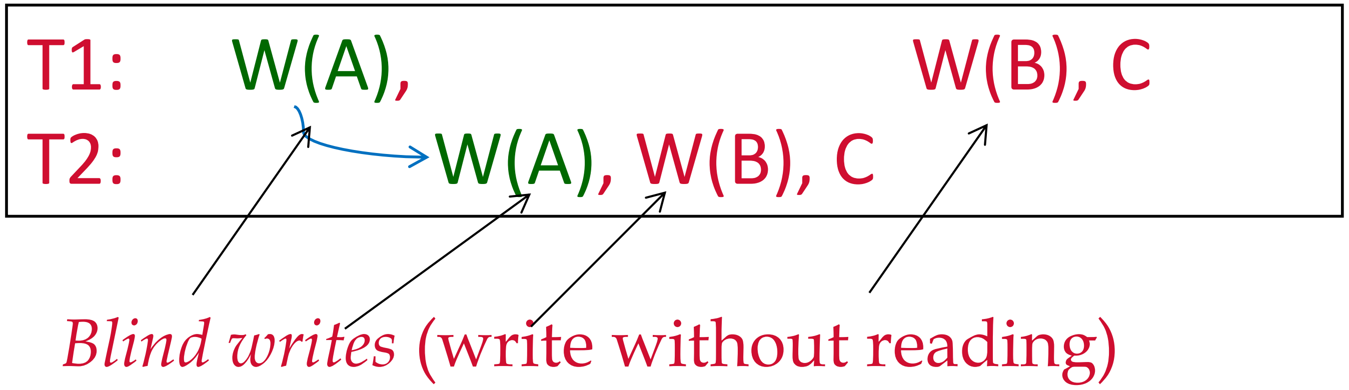
- **Unrepeatable Reads (RW Conflicts):**



- **Transaction T2 changes the value of an object A that has been read by a transaction T1, while T1 is still in progress**

Anomalies: Overwriting Uncommitted Data

- **Overwriting Uncommitted Data (WW Conflicts, blind writes):**



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

- **Transaction T2 overwrites the value of an object A, which has already been modified by a transaction T1 while T1 is still in progress**

Detecting anomalies systematically

- Schedules can be checked automatically for conflict serializability by means of a precedence graph
- Conflict serializable graphs are also serializable
- Locking protocols can guarantee conflict serializable schedules
- However, locking may introduce deadlocks, which can be detected with waits-for graphs

Conflict Serializable Schedules

- **Two operations a1 and a2 are said to conflict, iff**
  - They belong to different transactions
  - Both read or write the same object, e.g. A
  - And either a1 or a2 is a write operation

- **Two schedules are conflict equivalent iff:**

  They involve the same actions of the same transactions, *and*

  every pair of conflicting actions is ordered the same way

Conflict Serializable Schedules

- **If a schedule S can be transformed into a schedule S' by a series of swaps of nonconflicting instructions, we say that S and S are conflict equivalent**

- **Schedule S is conflict serializable if S is conflict equivalent to some serial schedule.**
  - Note, some "serializable" schedules are NOT conflict serializable.

Dependency Graph

- ***Dependency graph*: One node per Xact; edge from *Ti* to *Tj* if an operation of Ti conflicts with an operation of Tj and Ti's operation appears earlier in the schedule than the conflicting operation of Tj.**

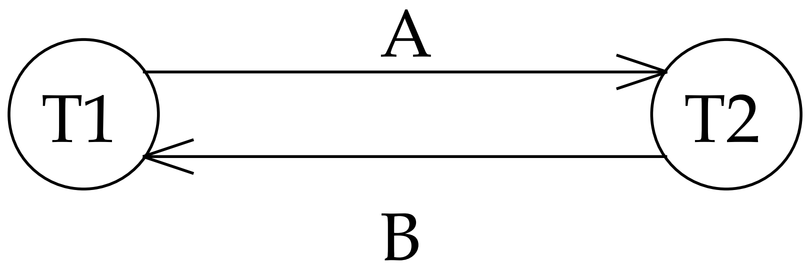
- **It follows that a dependency graph is a graph whose edges Ti →Tj meet one of the following three conditions:**
  - Ti executes write(A) before Tj executes read(A).
  - Ti executes read(A) before Tj executes write(A).
  - Ti executes write(A) before Tj executes write(A).

- **Theorem: Schedule is conflict serializable if and only if its dependency graph is acyclic**

Example

- **A schedule that is not conflict serializable:**

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



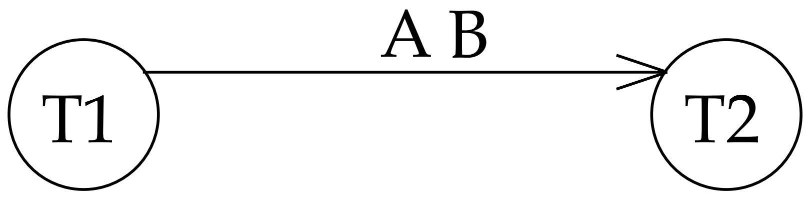
Dependency graph

- **The cycle in the graph reveals the problem. The output of T1 depends on T2, and vice-versa.**

Example

- **A schedule that is conflict serializable:**

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



Dependency graph

- **No Cycle Here!**

View Serializability – an Aside

- **Alternative (weaker) notion of serializability.**
- **Schedules S1 and S2 are view equivalent if:**
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2
- **Basically, allows all conflict serializable schedules + "blind writes"**

```
T1: R(A)        W(A)
T2:       W(A)
T3:                   W(A)
```

view $\equiv$

```
T1: R(A),W(A)
T2:              W(A)
T3:                    W(A)
```

Notes on Conflict Serializability

- **Conflict *Serializability doesn't* allow all schedules that you would consider correct.**
  - This is because it is strictly syntactic - it doesn' t consider the meanings of the operations or the data.
- **In practice, Conflict Serializability is what gets used, because it can be done efficiently.**
  - Note: in order to allow more concurrency, some special cases do get implemented, such as for travel reservations, etc.
- **Two-phase locking (2PL) is how we implement it.**

Locks

- We use "locks" to control access to items.
- Shared (S) locks – multiple transactions can hold these on a particular item at the same time.
- Exclusive (X) locks – only one of these and no other locks, can be held on a particular item at a time.

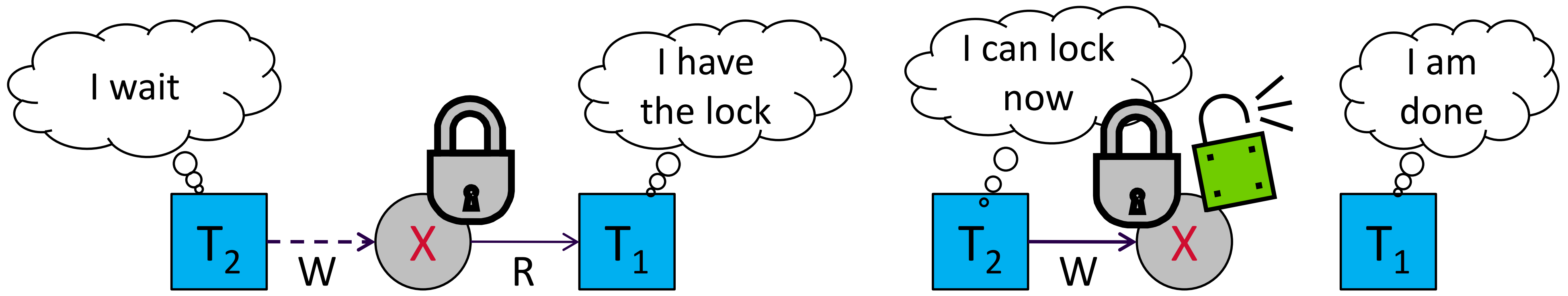
Lock Compatibility Matrix

	S	X
S	√	–
X	–	–

Scheduling Concurrent Transactions

DBMS ensures that execution of {T1, ... , Tn} is equivalent to some *serial* execution T1, ... Tn.

- Before reading/writing an object, a transaction requests a lock on the object, and waits till the DBMS gives it the lock.
- All locks are released at the end of the transaction. (Strict 2-Phase locking protocol)



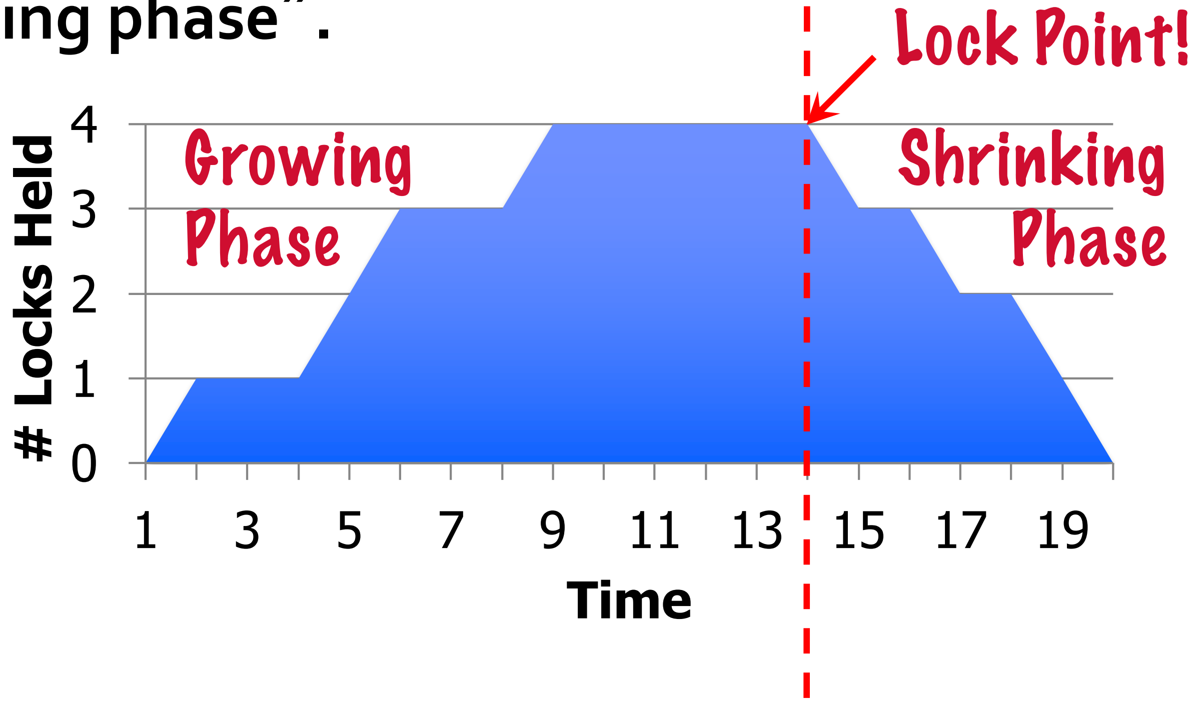
Two-Phase Locking (2PL)

1) Each transaction must obtain:

- a S (*shared*) or an X (exclusive) lock on object before reading,
- an X (*exclusive*) lock on object before writing.

2) A transaction can not request additional locks <u>once it releases</u> any locks.

Thus, each transaction has a "growing phase" followed by a "shrinking phase".



Two-Phase Locking (2PL)

- **2PL on its own is sufficient to guarantee conflict serializability.**
  - Doesn't allow dependency cycles! (note: see "Deadlock" discussion a few slides hence)
  - Schedule of conflicting transactions is conflict equivalent to a serial schedule ordered by "lock point".

The two phase locking rule can be summarized as: never acquire a lock after a lock has been released

Ex 1: A= 1000, B=2000, Output =?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	↓
Write(A)	↓
Unlock(A)	↓ <granted>
	Read(A)
	Unlock(A)
	Lock_S(B) <granted>
Lock_X(B)	
↓	Read(B)
↓ <granted>	Unlock(B)
	PRINT(A+B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	

Is it a 2PL schedule? No, and it is not serializable.

Ex 2: A= 1000, B=2000, Output =?

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	↓
Write(A)	↓
Lock_X(B) <granted>	↓
Unlock(A)	↓ <granted>
	Read(A)
	Lock_S(B)
Read(B)	↓
B := B +50	↓
Write(B)	↓
Unlock(B)	↓ <granted>
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

Is it a 2PL schedule? Yes: so it is serializable.

Avoiding Cascading Aborts – Strict 2PL

- **Problem with 2PL: Cascading Aborts**
- **Example: rollback of T1 requires rollback of T2!**

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

- **Solution: Strict Two-phase Locking (Strict 2PL):**
  - Same as 2PL, except:
  - **All locks held by a transaction are released only when the transaction completes**

Strict 2PL

- ***Strict Two-phase Locking (Strict 2PL) Protocol*:**
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes
  - If a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.

Strict 2PL (continued)

All locks held by a transaction are released only when the transaction completes

- **Like 2PL, Strict 2PL allows only schedules whose precedence graph is acyclic, but it is actually stronger than needed for that purpose.**

- **In effect, “shrinking phase” is delayed until:**
  a) Transaction has committed (commit log record on disk), or
  b) Decision has been made to abort the transaction (then locks can be released after rollback).

Strict 2PL (Papadimitrou)

- **Strict Two-phase Locking (Strict 2PL) Protocol:**
  - Each Xact must obtain a S (shared) lock on object before reading, and an X (exclusive) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes (abort or commit)
  - A Xact cannot lock the same object if one of its locks is a write lock held by another transaction
  - A Xact cannot lock an object with a write lock if a read lock on that Xact is held by another Xact
- **Read locks can be held by several transactions.**
- **If a write lock has been set, no read lock can be set and vice versa.**
- **Strict 2PL allows only schedules whose precedence graph is acyclic $\Rightarrow$ it enforces conflict serializable schedules**

Ex 3: A= 1000, B=2000, Output =?

Lock_X(A) <granted>	
Read(A)	**Lock_S(A)**
A: = A-50	
Write(A)	
Lock_X(B) <granted>	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	**<granted>**
	Read(A)
	Lock_S(B) <granted>
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)

Is it a 2PL schedule? Strict 2PL?

Ex 2: Revisited

Lock_X(A) <granted>	
Read(A)	Lock_S(A)
A: = A-50	↓
Write(A)	↓
Lock_X(B) <granted>	↓
Unlock(A)	↓ <granted>
	Read(A)
	Lock_S(B)
Read(B)	↓
B := B +50	↓
Write(B)	↓
Unlock(B)	↓ <granted>
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

Is it Strict 2PL? No: Cascading Abort Poss.

Lock Management

- **Lock and unlock requests are handled by the Lock Manager.**
- **LM contains an entry for each currently held lock.**
- **Lock table entry:**
  - Ptr. to list of transactions currently holding the lock
  - Type of lock held (shared or exclusive)
  - Pointer to **queue of lock requests**
- **When lock request arrives see if anyone else holds a conflicting lock.**
  - If not, create an entry and grant the lock.
  - Else, put the requestor on the wait queue
- **Locking and unlocking have to be atomic operations**
- **Lock upgrade: transaction that holds a shared lock can be upgraded to hold an exclusive lock**
  - Can cause deadlock problems

Ex 4: Output = ?

Lock_X(A) <granted>	
	Lock_S(B) <granted>
	Read(B)
	Lock_S(A)
Read(A)	↓
A: = A-50	↓
Write(A)	↓
Lock_X(B)	↓
↓	↓
↓	↓
↓	↓
↓	↓
↓	↓
↓	↓
↓	↓
↓	↓

Is it a 2PL schedule? Strict 2PL?

Deadlocks

- **Deadlock: Cycle of transactions waiting for locks to be released by each other.**
- **Two ways of dealing with deadlocks:**
  - Deadlock prevention
  - Deadlock detection

- **Many systems just punt and use Timeouts**

Deadlock Prevention

- **Assign priorities based on timestamps. Assume Ti wants a lock that Tj holds. Two policies are possible:**
  - Wait-Die: If Ti is older, Ti waits for Tj; otherwise Ti aborts
  - Wound-wait: If Ti is older, Tj aborts; otherwise Ti waits
- **If a transaction re-starts, make sure it gets its original timestamp**

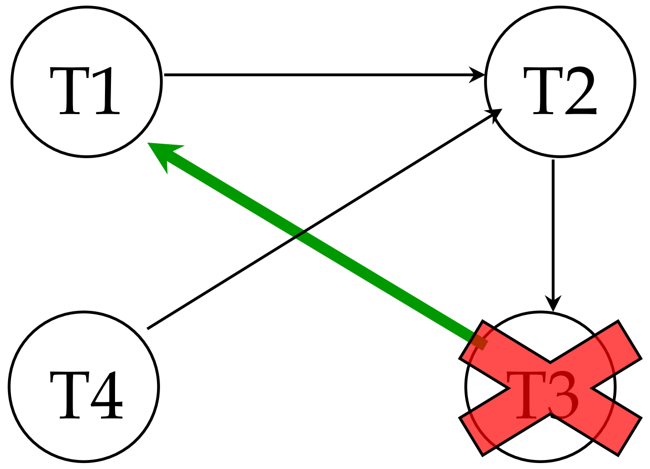
Deadlock Detection

- **Alternative is to allow deadlocks to happen but to check for them and fix them if found.**
- **Create a waits-for graph:**
  - Nodes are transactions
  - There is an edge from Ti to Tj if Ti is waiting for Tj to release a lock
- **Periodically check for cycles in the waits-for graph**
- **If cycle detected – find a transaction whose removal will break the cycle and kill it.**

Deadlock Detection (Continued)

- **Example:**

- **T1: S(A), S(D), S(B)**
- **T2: X(B) X(C)**
- **T3: S(D), S(C), X(A)**
- **T4: X(B)**



The “Phantom” Problem

- **With Insert and Delete, even Strict 2PL (on individual items) will not assure serializability:**
- **Consider T1 – “Find oldest sailor”**
  - T1 locks all records, and finds oldest sailor (*age* = 71).
  - Next, T2 inserts a new sailor; *age* = 96 and commits.
  - T1 (within the same transaction) checks for the oldest sailor again and finds sailor aged 96!!
- **The sailor with age 96 is a “phantom tuple” from T1’s point of view --- first it’s not there then it is.**
- **No serial execution where T1’s result could happen!**

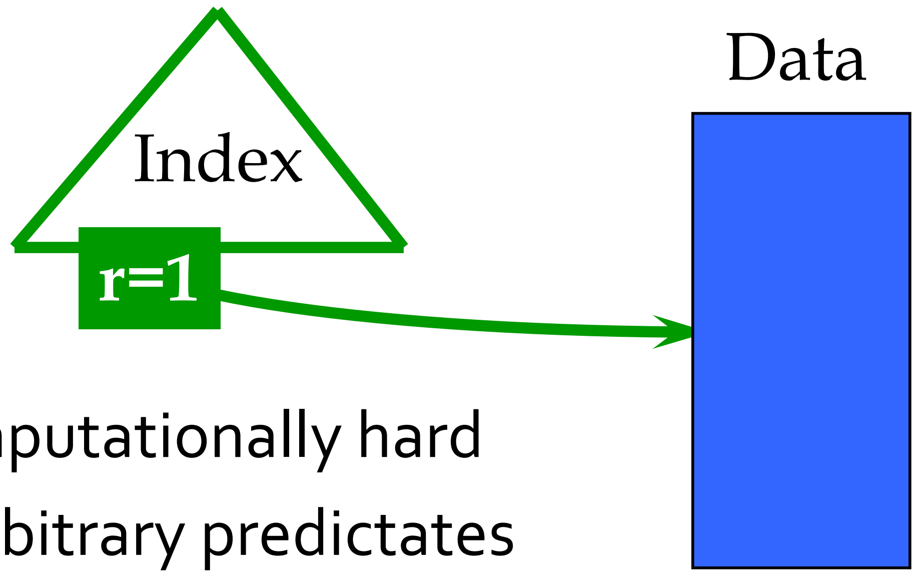
The “Phantom” Problem – example 2

- **Consider T3 – “Find oldest sailor for each rating”**
  - T3 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
  - Next, T4 inserts a new sailor; *rating* = 1, *age* = 96.
  - T4 also deletes oldest sailor with rating = 2 (and, say, *age* = 80), and commits.
  - T3 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- **T3 saw only part of T4’s effects!**
- **No serial execution where T3’s result could happen!**

The Problem

- **T1 and T3 implicitly assumed that they had locked the set of all sailor records satisfying a predicate.**
  - Assumption only holds if no sailor records are added while they are executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- **Examples show that conflict serializability on reads and writes of individual items guarantees serializability only if the set of objects is fixed!**

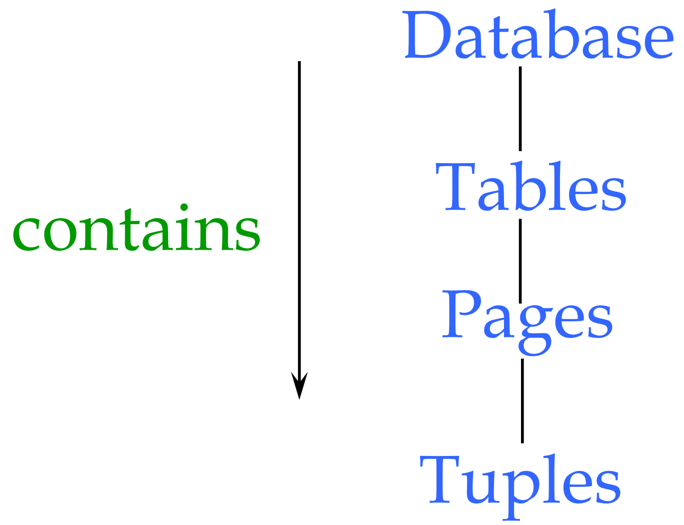
Solution: Index Key Value Locking



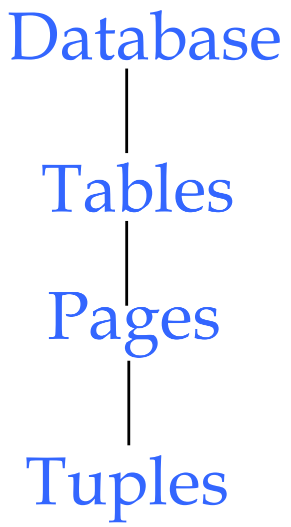
- Locking Predicates directly is computationally hard
  - Need to calculate overlap of arbitrary predictates
- If there is a dense index on the *rating* field using Alternative (2), T3 should lock the index page containing the data entries with *rating* = 1.
  - If there are no records with *rating* = 1, T3 must lock the index page where such a data entry *would* be, if it existed!
- If there is no suitable index, T3 must lock all pages, and lock the file/table to prevent new pages from being added, to ensure that no records with *rating* = 1 are added or deleted.

Multiple-Granularity Locks

- Hard to decide what granularity to lock (tuples vs. pages vs. tables).
- Shouldn't have to make same decision for all transactions!
- Data "containers" are nested:



Solution: New Lock Modes, Protocol



- **Allow Xacts to lock at each level, but with a special protocol using new "intention" locks:**
- **Still need S and X locks, but before locking an item, Xact must have proper intension locks on all its ancestors in the granularity hierarchy.**

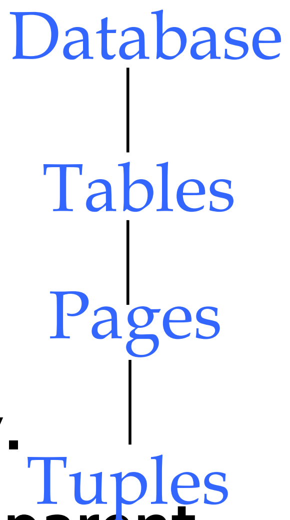
❖ IS – Intent to get S lock(s) at finer granularity.

❖ IX – Intent to get X lock(s) at finer granularity.

❖ SIX mode: Like S & IX at the same time. Why useful?

	IS	IX	SIX	S	X
IS	√	√	√	√	–
IX	√	√	–	–	–
SIX	√	–	–	–	–
S	√	–	–	√	–
X	–	–	–	–	–

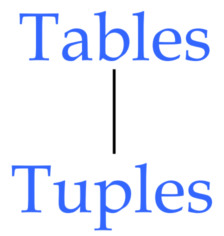
Multiple Granularity Lock Protocol



- **Each Xact starts from the root of the hierarchy.**
- **To get S or IS lock on a node, must hold IS or IX on parent node.**
  - What if Xact holds SIX on parent? S on parent?
- **To get X or IX or SIX on a node, must hold IX or SIX on parent node.**
- **Must release locks in bottom-up order.**

Protocol is correct in that it is equivalent to directly setting locks at the leaf levels of the hierarchy.

Examples – 2 level hierarchy



- **T1 scans R, and updates a few tuples:**
  - T1 gets an SIX lock on R, then get X lock on tuples that are updated.
- **T2 uses an index to read only part of R:**
  - T2 gets an IS lock on R, and repeatedly gets an S lock on tuples of R.
- **T3 reads all of R:**
  - T3 gets an S lock on R.
  - OR, T3 could behave like T2; can use lock escalation to decide which.
  - Lock escalation dynamically asks for courser-grained locks when too many low level locks acquired

	IS	IX	SIX	S	X
IS	√	√	√	√	
IX	√	√			
SIX	√				
S	√			√	
X					

Isolation Levels

- **SQL standard offers several isolation levels**
  - Each transaction can have level set separately
  - Problematic definitions, but in best practice done with variations in lock holding
- **Serializable**
  - (ought to be default, but not so in practice)
  - Traditionally done with Commit-duration locks on data and indices (to avoid phantoms)
- **Repeatable Read**
  - Commit-duration locks on data
  - Phantoms can happen
- **Read Committed**
  - short duration read locks, commit-duration write locks
  - non-repeatable reads possible
- **Read Uncommitted**
  - no read locks, commit-duration write locks

Optimistic CC (Kung-Robinson)

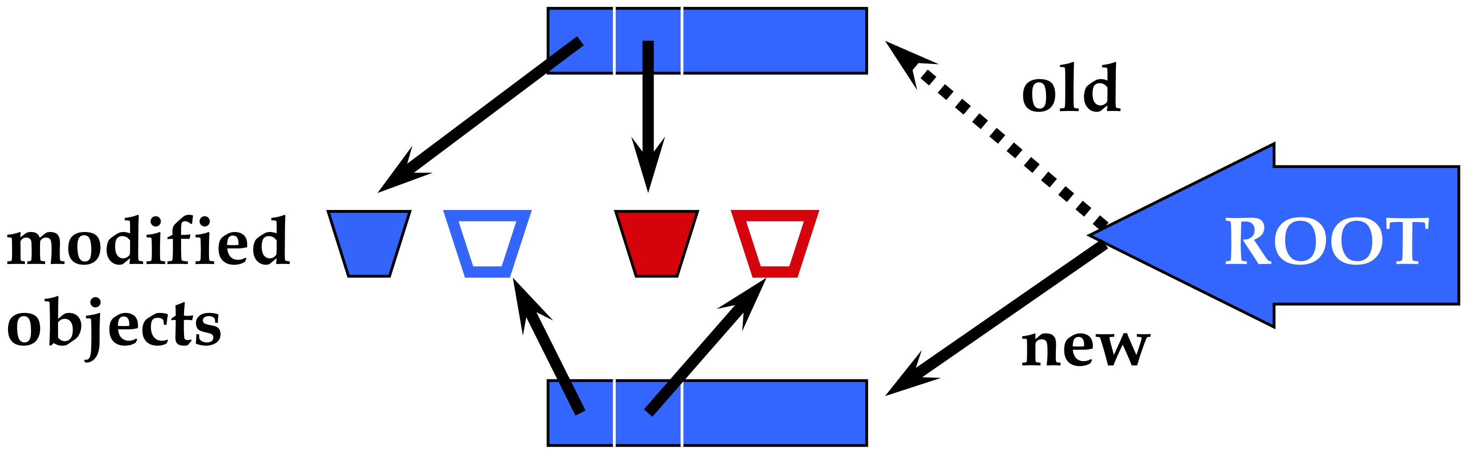
Locking is a conservative approach in which conflicts are prevented. Disadvantages:

  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.

- Locking is "pessimistic" because it assumes that conflicts will happen.
- If conflicts are rare, we might get better performance by not locking, and instead checking for conflicts at commit.

Kung-Robinson Model

- **Xacts have three phases:**
  - READ: Xacts read from the database, but make changes to private copies of objects.
  - VALIDATE: Check for conflicts.
  - WRITE: Make local copies of changes public.

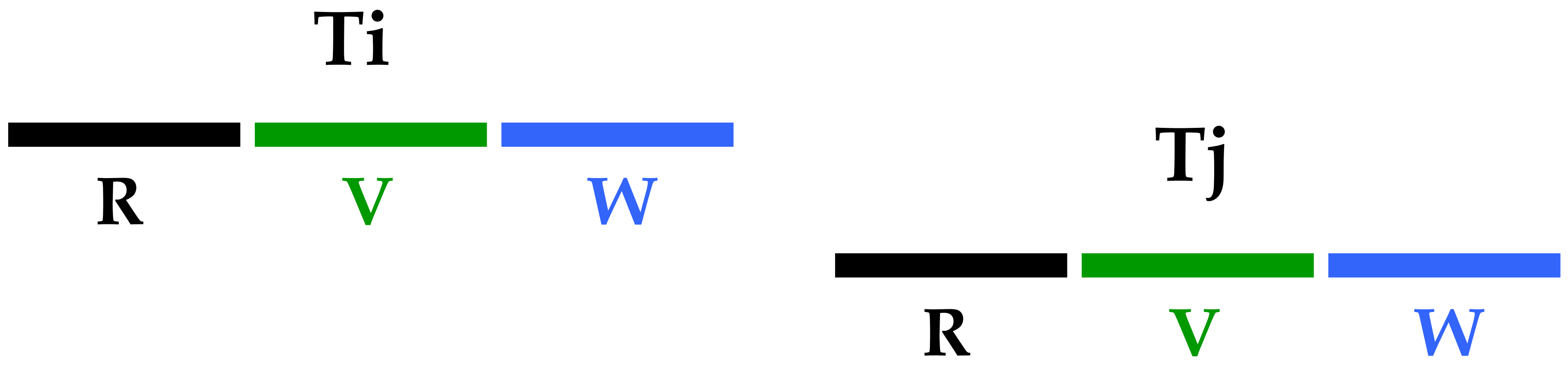


Validation

- **Test conditions that are sufficient to ensure that no conflict occurred.**
- **Each Xact is assigned a numeric id.**
  - Just use a **timestamp (call it T_i)**.
- **Timestamps are assigned at end of READ phase, just before validation begins.**
- **ReadSet(Ti): Set of objects read by Xact T_i**
- **WriteSet(Ti): Set of objects modified by T_i**

Test 1 – non-overlapping

- For all i and j such that Ti < Tj, check that Ti completes before Tj begins.

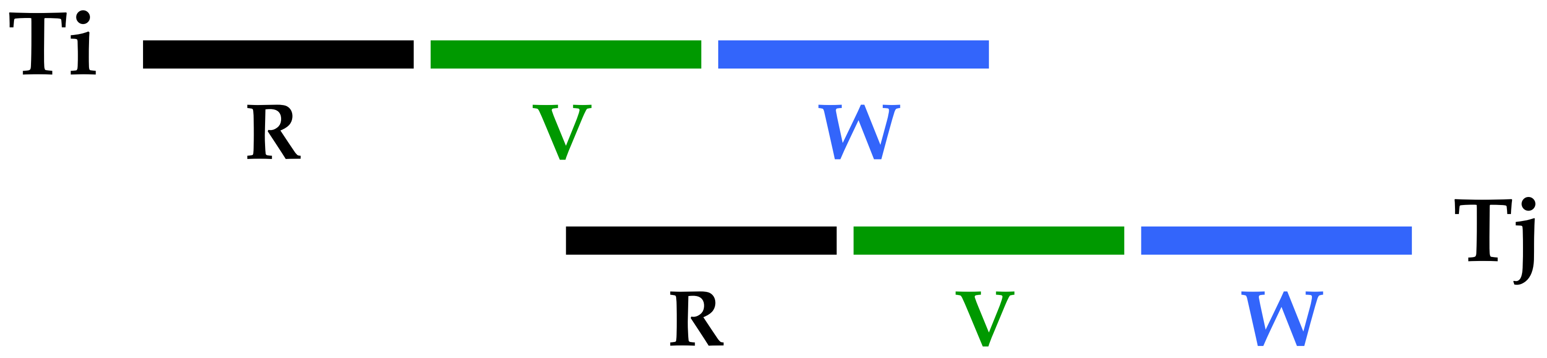


Test 2 – No Write Phase Conflict

- **For all i and j such that Ti < Tj, check that:**

  Ti completes before Tj begins its Write phase

  and WriteSet(Ti) ∩ ReadSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

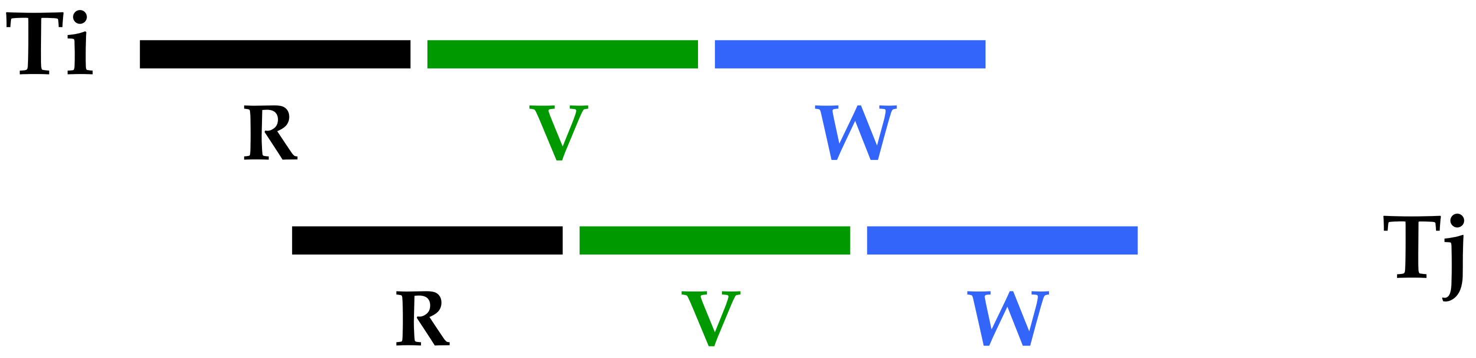
Test 3 – Overlapping Write Phases

- **For all i and j such that Ti < Tj, check that:**

  Ti completes Read phase before Tj does **+**

  WriteSet(Ti) $\cap$ ReadSet(Tj) is empty **+**

  WriteSet(Ti) $\cap$ WriteSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

Applying Tests 1 & 2: Serial Validation

- **To validate Xact T:**

```
valid = true;
// S = set of Xacts that committed after Begin(T)
//    (above defn implements Test 1)
//The following is done in critical section
< foreach  Ts in S do {
   if ReadSet(T) intersects WriteSet(Ts)
         then valid = false;
   }
   if valid then { install updates; // Write phase
                      Commit T } >
             else Restart T
```

start of critical section

end of critical section

Comments on Serial Validation

- **Applies Test 2, with T playing the role of Tj and each Xact in Ts (in turn) being Ti.**
- **Assignment of Xact id, validation, and the Write phase are inside a** critical section**!**
  - Nothing else goes on concurrently.
  - So, no need to check for Test 3 --- can't happen.
  - If Write phase is long, major drawback.

- **Optimization for Read-only Xacts:**
  - Don't need critical section (because there is no Write phase).

Overheads in Optimistic CC

- **Must record read/write activity in ReadSet and WriteSet per Xact.**
  - Must create and destroy these sets as needed.
- **Must check for conflicts during validation, and must make validated writes ``global''.**
  - Critical section can reduce concurrency.
  - Scheme for making writes global can reduce clustering of objects.
- **Optimistic CC restarts Xacts that fail validation.**
  - Work done so far is wasted; requires clean-up.

Snapshot Isolation (SI)

- **A multiversion concurrency control mechanism was described in SIGMOD ’95 by H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O’Neil, P. O’Neil**
  - Does not guarantee serializable execution!
- **Supplied by Oracle DB, and PostgreSQL (before rel 9.1), for “Isolation Level Serializable”**
- **Available in Microsoft SQL Server 2005 as “Isolation Level Snapshot”**

Snapshot Isolation (SI)

- **Read of an item may not give current value**
- **Instead, use old versions (kept with timestamps) to find value that had been most recently committed at the time the txn started**
  - Exception: if the txn has modified the item, use the value it wrote itself
- **The transaction sees a "snapshot" of the database, at an earlier time**
  - Intuition: this should be consistent, if the database was consistent before

First committer wins (FCW)

- **T will not be allowed to commit a modification to an item if any other transaction has committed a changed value for that item since T' s start (snapshot)**
- **Similar to optimistic CC, but only write-sets are checked**
- **T must hold write locks on modified items at time of commit, to install them.**
  - In practice, commit-duration write locks may be set when writes execute.

Benefits of SI

- **Reading is never blocked, and reads don' t block writes**
- **Avoids common anomalies**
  - No dirty read
  - No lost update
  - No inconsistent read
  - Set-based selects are repeatable (no phantoms)
- **Matches common understanding of isolation: concurrent transactions are not aware of one another' s changes**
- **On the downside – it turns out that it doesn't fully guarantee Serializablity (but Prof. Alan Fekete & team have fixed this in PostgreSQL 9.1+)**

Other Techniques

- Timestamp CC: **Give each object a read-timestamp (RTS) and a write-timestamp (WTS), give each Xact a timestamp (TS) when it begins:**
  - If action ai of Xact Ti conflicts with action aj of Xact Tj, and TS(Ti) < TS(Tj), then ai must occur before aj. Otherwise, restart violating Xact.

- Multiversion CC: **Let writers make a “new” copy while readers use an appropriate “old” copy.**
  - Advantage is that readers don’t need to get locks
  - Oracle and PostgreSQL use a simple form of this.

Summary

- **Correctness criterion for isolation is "serializability".**
  - In practice, we use "conflict serializability", which is somewhat more restrictive but easy to enforce.
- **Two Phase Locking, and Strict 2PL: Locks directly implement the notions of conflict.**
  - The lock manager keeps track of the locks issued. Deadlocks can either be prevented or detected.
- **Must be careful if objects can be added to or removed from the database ("phantom problem").**
- **Index locking common, affects performance significantly.**
  - Needed when accessing records via index.
  - Needed for locking logical sets of records (index locking/predicate locking).

Summary (Contd.)

- **Multiple granularity locking reduces the overhead involved in setting locks for nested collections of objects (e.g., a file of pages);**
  - should not be confused with tree index locking!
- **Optimistic CC aims to allow progress when conflicts are rare or getting locks is expensive (e.g. distributed sys)**
- **Optimistic CC has its own overheads however; most real systems use locking or Snapshot Isolation.**
- **Snapshot Isolation is a practical approach that let's readers run without locks, by looking at (possibly) older snapshots.**