Transactions

Definition

- The execution of a sequence of one or more operations on a shared database
- The unit of state transition in a database
  - from a consistent state to another consistent state
- Cannot produce side effects in outside world
  - they can't be rolled back (output commit)
- Managed by DBMS

Morphology

- A sequence of read and write operations
  - R(x), W(y)
- Two possible outcomes
  - COMMIT: all transaction's changes saved to database
  - ABORT: none are
    - can be self-inflicted or caused by DBMS

Correctness: ACID

- The Fantastic Four
  - Atomicity
    - All (Commit) or nothing (Abort)
  - Consistency
    - Invariants are maintained
  - Isolation
    - As if running solo
  - Durability
    - Committed changes survive failures

Atomicity: how

- Logging
  - The log knows everything
    - movie vs. snapshots
  - DBMS logs operations, so it can undo them if transaction aborts
- Shadow paging
  - copy on write
  - out of fashion

Consistency: what and how

- Consistency in ACID ≠ consistency in distributed systems
  - though both have to do with correctness
    - in DS, it specifies which values can be read given a set of writes
    - in ACID, it requires the DB to not be delusional
      - consistent with the outside world it is to model
      - internally consistent
  - DS consistency models are general
  - ACID consistency is application specific

Durability

- Changes from committed transactions are persistent
- Implementation:
  - logging
  - shadow paging

Isolation

- Though transactions execute concurrently, it appears to each T as if every other transaction executed either before or after T
- It does not just "happen"...

Initially A and B have $1000 balance

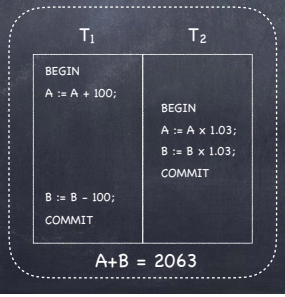
T_1	T_2
BEGIN A := A + 100; B := B - 100; COMMIT	BEGIN A := A x 1.03; B := B x 1.03; COMMIT

In every execution A+B should be $2060

Isolation

- Though transactions execute concurrently, it appears to each T as if every other transaction executed either before or after T
- It does not just "happen"...

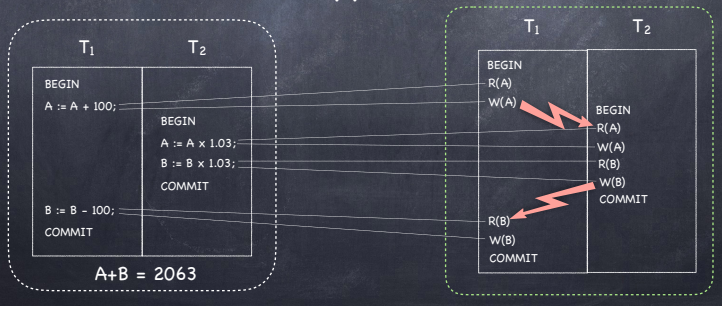
Initially A and B have $1000 balance



In every execution A+B should be $2060

Isolation

- Though transactions execute concurrently, it appears to each T as if every other transaction executed either before or after T
- It does not just "happen"...

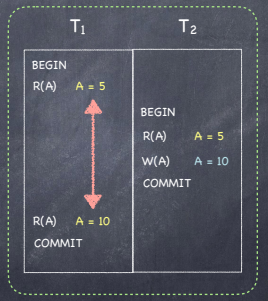


Isolation

- An execution schedule is correct if it is equivalent to a serial schedule (Serializable schedule)
  - S_1 is equivalent to S_2 if they have the same effect on the database
- Enforced through a concurrency control mechanism that handles conflicting operations
  - pessimistic: prevent inconsistencies from arising by blocking conflicting operations
    - in different transactions
    - on the same object, at least one a write
  - optimistic: allow execution to proceed, taking action if conflicts happen

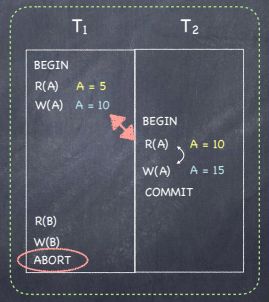
Read-Write Conflicts

- Non-repeatable reads
  - same object read during a transaction
  - no intervening write
  - reads return different values!
- Can't happen in a serializable transaction



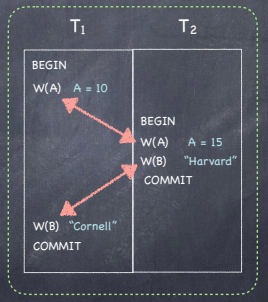
Write-Read Conflicts

- Dirty read
  - committed transaction reads object version created by uncommitted transaction
- Can't happen in a serializable transaction



Write-Write Conflicts

- Dirty writes
  - overwrite committed data
- Can't happen in a serializable transaction

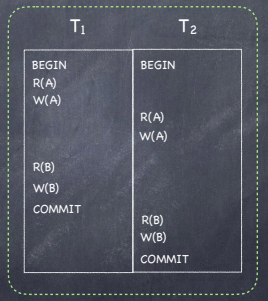


Flavors of Serializability

- Conflict serializability
  - specializes equivalence to conflict equivalence
    - schedules involve same operations of the same transactions
    - every pair of conflicting operations is ordered in the same way as in some serial execution
- View serializability
  - conflict serializable and allows "blind writes"
    - NP-complete - not used in practice

Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions



Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions

T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions

T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions

T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
R(B)	
	W(A)
W(B)	
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions

T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
R(B)	
W(B)	
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions

T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
	R(A)
R(B)	
W(B)	
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions

T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions

T_1	T_2
BEGIN	BEGIN
R(A)	
W(A)	
R(B)	
	R(A)
W(B)	
	W(A)
COMMIT	
	R(B)
	W(B)
	COMMIT

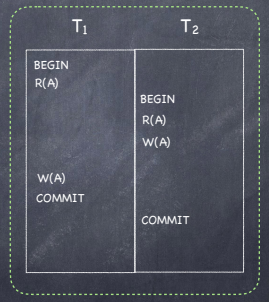
Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions

T_1	T_2
BEGIN	
R(A)	
W(A)	
R(B)	
W(B)	
COMMIT	
	BEGIN
	R(A)
	W(A)
	R(B)
	W(B)
	COMMIT

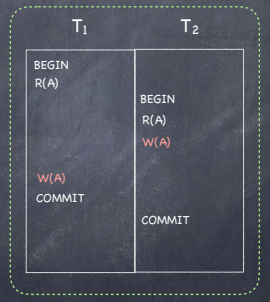
Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions



Conflict serializability: the intuition

- A schedule S is conflict serializable if it can be transformed into a serial schedule by swapping consecutive non-conflicting operations of different transactions

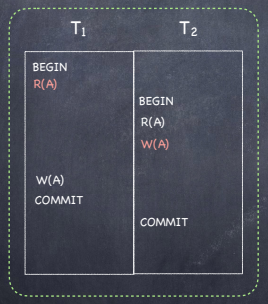


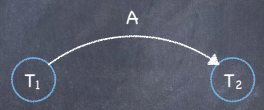
Not conflict serializable

Dependency graphs

- One node per committed transaction
- Edge from T_i to T_j if
  - O_i of T_i conflicts with O_j of T_j
  - O_i appears in schedule before O_j
- A schedule is conflict serializable if and only if its dependency graph is acyclic

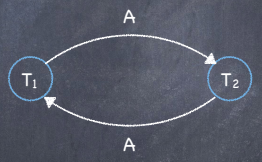
Dependency graphs





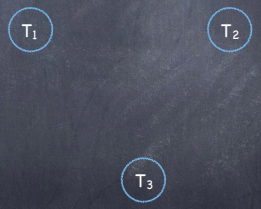
Dependency graphs

T_1	T_2
BEGIN R(A)	
	BEGIN R(A) W(A)
W(A)	
COMMIT	
	COMMIT



Dependency graphs

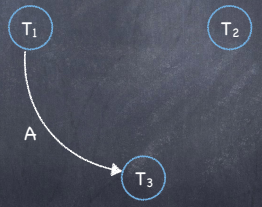
T_1	T_2	T_3
BEGIN R(A) W(A)		
		BEGIN R(A) W(A) COMMIT
	BEGIN R(B) W(B) COMMIT	
R(B) W(B) COMMIT		



Is this equivalent to a serial execution?

Dependency graphs

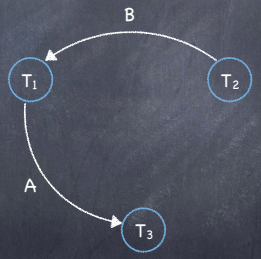
T_1	T_2	T_3
BEGIN R(A) W(A)		
		BEGIN R(A) W(A) COMMIT
	BEGIN R(B) W(B) COMMIT	
R(B) W(B) COMMIT		



Is this equivalent to a serial execution?

Dependency graphs

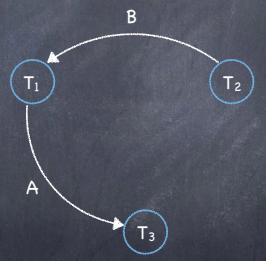
T_1	T_2	T_3
BEGIN R(A) W(A)		
		BEGIN R(A) W(A) COMMIT
	BEGIN R(B) W(B) COMMIT	
R(B) W(B) COMMIT		



Is this equivalent to a serial execution?

Dependency graphs

T_1	T_2	T_3
BEGIN R(A) W(A)		
	BEGIN R(B) W(B) COMMIT	
		BEGIN R(A) W(A) COMMIT
R(B) W(B) COMMIT		



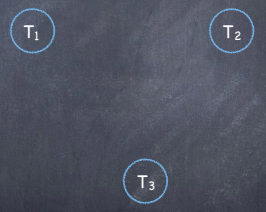
Is this equivalent to a serial execution?

View serializability

- Two schedules S_1 and S_2 if, for each object O
  - if T_i reads the initial value of O in S_1, then T_i also reads the initial value of O in S_2
  - if T_i reads the value of O written by T_j in S_1, then T_i also reads the value of O written by T_j in S_2
  - if T_i writes the final value of O in S_1, then T_i also writes the final value of O in S_2

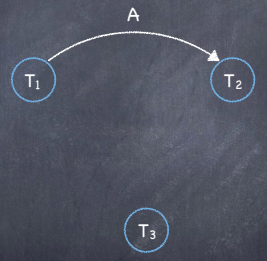
View serializability

T_1	T_2	T_3
BEGIN R(A)		
	BEGIN W(A)	
		BEGIN
W(A)		
COMMIT	COMMIT	W(A) COMMIT



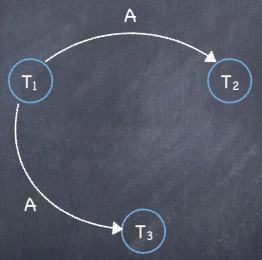
View serializability

T_1	T_2	T_3
BEGIN R(A)		
	BEGIN W(A)	
		BEGIN
W(A)		
COMMIT	COMMIT	W(A) COMMIT



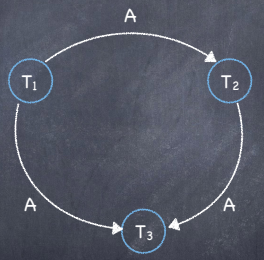
View serializability

T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT



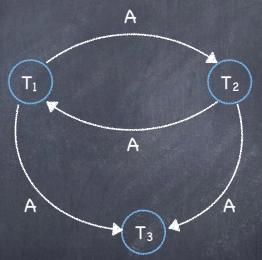
View serializability

T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT



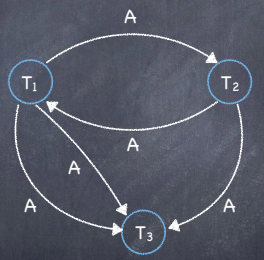
View serializability

T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT



View serializability

T_1	T_2	T_3
BEGIN		
R(A)	BEGIN	
	W(A)	BEGIN
W(A)		
		W(A)
COMMIT	COMMIT	COMMIT



and yet...

View serializability

T_1	T_2	T_3
BEGIN R(A)	BEGIN W(A)	BEGIN
W(A)		W(A)
COMMIT	COMMIT	COMMIT

$\asymp$ view

T_1	T_2	T_3
BEGIN R(A) W(A) COMMIT	BEGIN W(A) COMMIT	BEGIN W(A) COMMIT

- ✓ if T_i reads the initial value of O in S_1, then T_i also reads the initial value of O in S_2
- ✓ if T_i reads the value of O written by T_j in S_1, then T_i also reads the value of O written by T_j in S_2
- ✓ if T_i writes the final value of O in S_1, then T_i also writes the final value of O in S_2

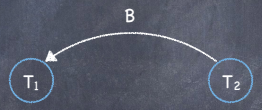
Conflict serializable + blind writes

There are more things in heaven and earth...

T_1	T_2
BEGIN R(A) A := A - 50 W(A)	BEGIN
	R(B) B := B - 10 W(B)
R(B) B := B + 50 W(B)	
	R(A) A := A + 10 W(A)
COMMIT	COMMIT

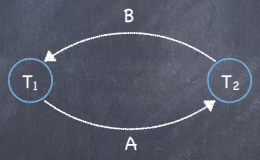
There are more things in heaven and earth...

T_1	T_2
BEGIN R(A) A := A - 50 W(A)	BEGIN
	R(B) B := B - 10 W(B)
R(B) B := B + 50 W(B)	
	R(A) A := A + 10 W(A)
COMMIT	COMMIT



There are more things in heaven and earth...

T_1	T_2
BEGIN R(A) A := A - 50 W(A)	BEGIN
	R(B) B := B - 10 W(B)
R(B) B := B + 50 W(B)	
	R(A) A := A + 10 W(A)
COMMIT	COMMIT



There are more things in heaven and earth...

T_1	T_2
BEGIN	BEGIN
R(A)	
A := A - 50	
W(A)	
	R(B)
	B := B - 10
	W(B)
R(B)	
B := B + 50	
W(B)	
	R(A)
	A := A + 10
	W(A)
COMMIT	COMMIT

- ✓ if T_i writes the final value of O in S_1, then T_i also writes the final value of O in S_2
  - T_1 writes last value of B
  - T_2 writes last value of A
  - can't be view equivalent to any serial execution

and yet it produces same outcome as <T_1, T_2>

How to enforce conflict serializability?

Locks and Latches

- Locks
  - interleave transactions, to protect consistency of database content
  - held for length of transaction
  - must be able to rollback changes
- Latches
  - interleave threads, to protect critical section of internal data structures
  - held for length of operation
  - do not need to rollback changes

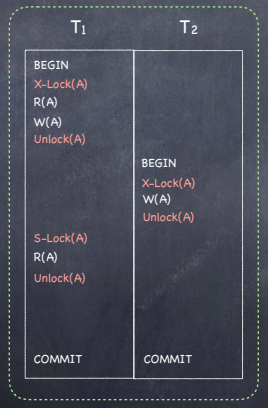
Basic lock types

S-Lock: shared lock for reads
X-Lock: exclusive lock for writes

Compatibilty Matrix

	Shared	Exclusive
Shared	✓	✗
Exclusive	✗	✗

Executing with locks



Non-repeatable reads!

Two-Phase Locking (2PL)

- Determines, on the fly, whether a transaction can acquire a lock
  - without knowing the transaction's future operations

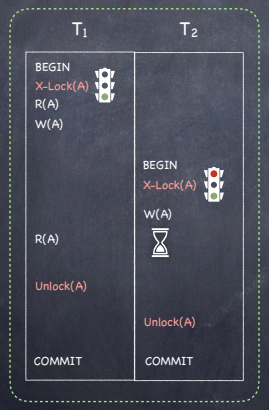
Phase 1: Growing

- transaction requests locks it needs
- lock manager grants/denies

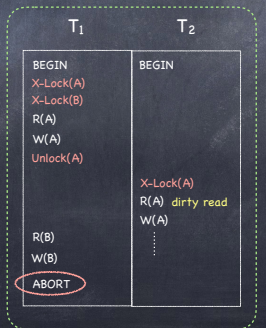
Phase 2: Shrinking

- transaction releases locks it acquired
- no more acquires after first release

Executing with 2PL



- 2PL generates schedules with cycle-free dependency graphs
  - but can suffer from cascading aborts



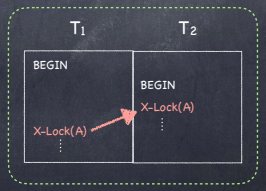
  - and rejects schedules allowed by conflict serializability

Strict 2PL

- As in 2PL, transaction is not allowed to acquire/upgrade locks after end of growing phase
- Unlike 2PL, it cannot release locks until transaction ends
- Rejects more conflict serializable schedules than 2PL

2PL can deadlock

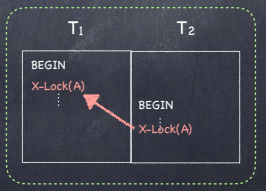
- Deadlock detection
  - Waits-for graph
  - Victim selection criteria
    - age; progress; no. of items locked; no. of transactions that must be rolled back
- Deadlock prevention
  - timestamp transactions
    - Wait-Die
    - Wound-Wait
  - waiting allowed only in one direction



Wait-Die
T_1 waits
Wound-wait
T_2 aborts

2PL can deadlock

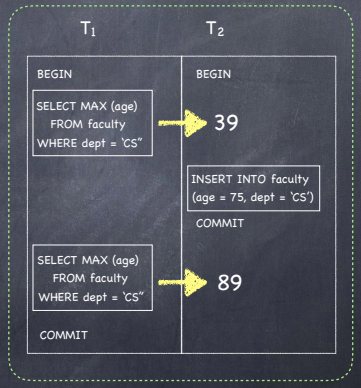
- Deadlock detection
  - Waits-for graph
  - Victim selection criteria
    - age; progress; no. of items locked; no. of transactions that must be rolled back
- Deadlock prevention
  - timestamp transactions
    - Wait-Die
    - Wound-Wait
  - waiting allowed only in one direction



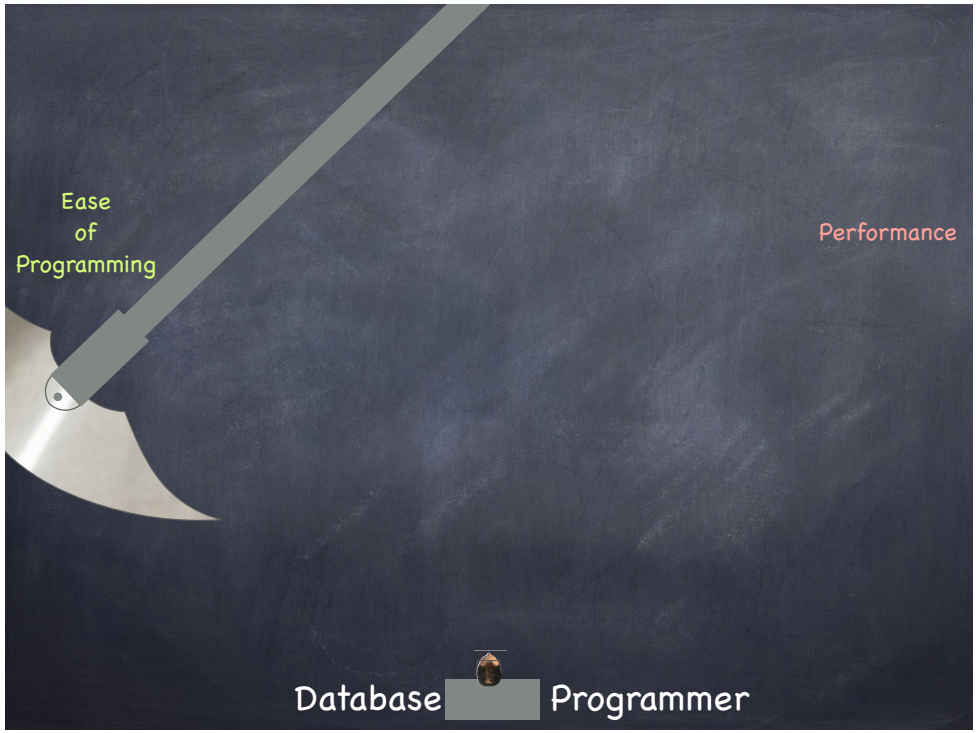
Wait-Die
T_2 aborts
Wound-wait
T_2 wait

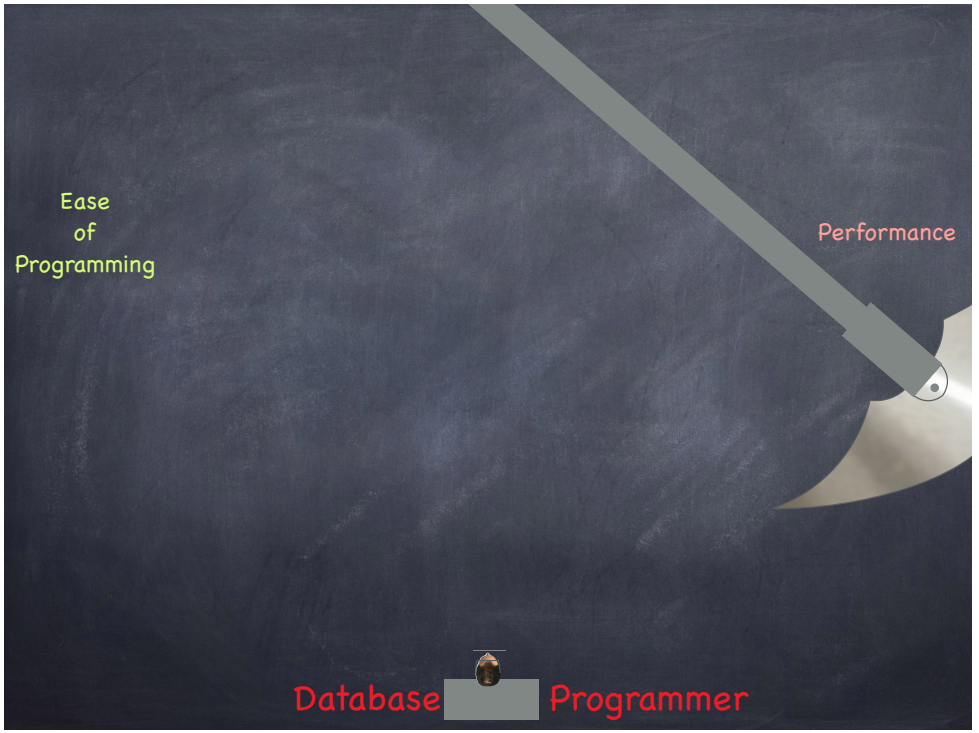
The Phantom Menace

- An update in T_i can change result of serial scan in T_j
- Conflict Serializability on R/W
  - serializability only if set of objects is fixed
- Solution: predicate locking
  - lock records that satisfy a logical predicate
    - difficult
- Alternatives: index locking, repeated scan to check for inconsistencies



A moment of silence





Weaker levels of Isolation

- Repeatable Reads
  - Phantoms may happen
- Read Committed
  - Phantoms and non-repeatable reads
- Read Uncommitted
  - Phantoms, non repeatable reads and dirty reads
- It-that-shall-not-be-named
  - all of the above, and dirty writes

Both transactions set x = y

W1[x]	W1[y]	C1
W2[x]	W2[y]	C2

Weaker levels of Isolation

- Repeatable Reads
  - Phantoms may happen
- Read Committed
  - Phantoms and non-repeatable reads
- Read Uncommitted
  - Phantoms, non repeatable reads and dirty reads
- It-that-shall-not-be-named
  - all of the above, and dirty writes

Both transactions set x = y

W1[x] W2[x] W2[y] C2 W1[y] C1

Locks and Isolation levels

- Serializable
  - obtains all locks and index locks first; strict 2PL
- Repeatable Reads
  - as above, but no index locks
- Read Committed
  - as above, but shared locks released immediately
- Read Uncommitted
  - as above, but dirty reads (no shared locks)

Timestamp Ordering

- Timestamps determine the serializability order of transactions
  - no locks
- If $TS(T_i) < TS(T_j)$ then DBMS ensures equivalent schedule to a serialization where T_i precedes T_j
  - different schemes to determine transactions' timestamps
  - different timestamp implementations

Basic Timestamp Ordering

- Every object O tagged with TS of last transaction that performed Read/Write
  - W-TS(O); R-TS(O)
- Before each operation, DBMS verifies timestamps
  - a transaction that tries to access an object from the future is aborted and restarted

Basic TO: Reads

- If $TS(T_i) < W\text{-}TS(O)$, then T_i is trying to read a value from a later transaction
  - T_i is aborted
- Otherwise:
  - T_i reads O
  - $R\text{-}TS(O) := \max(R\text{-}TS(O), TS(T_i))$
  - Make local copy of O to ensure repeatable reads for T_i

Basic TO: Writes

- If $TS(T_i) < W\text{-}TS(O)$ or $TS(T_i) < R\text{-}TS(O)$
  - T_i is aborted
- Otherwise:
  - T_i write O
  - $R\text{-}TS(O) := \max(W\text{-}TS(O), TS(T_i))$
  - Make local copy of O to ensure repeatable reads for T_i

TO Write optimization: Thomas Write Rule

- If $TS(T_i) < R\text{-}TS(O)$
  - T_i is aborted
- If $TS(T_i) < W\text{-}TS(O)$
  - ignore write and allow T_i to proceed
    - no one will read it anyway!
- Otherwise
  - T_i write O
  - $R\text{-}TS(O) := \max(W\text{-}TS(O), TS(T_i))$
  - Make local copy of O to ensure repeatable reads

Musings on Basic TO

- Generates conflict serializable schedules
  - but not if using Thomas Write Rule
- Allows non-recoverable schedules
  - dirty reads; to avoid, delay committing
- Performance
  - Updating timestamps has high overhead
  - Long running transactions can be starved
  - If concurrency is high, timestamp allocation can become a bottleneck

Optimistic Concurrency Control Kung, Robinson (1981)

- Motivation
  - If conflicts are rare and transactions shortlived, locks are inefficient
    - optimize for the common case!
- OCC
  - each transaction T creates a private workspace
    - read objects are copied in the workspace
    - writes are only applied to the workspace
  - at T's commit time, DBMS verifies whether T's write set conflicts with other transactions
    - if not, the changes are atomically applied to shared database
    - otherwise, T is aborted

One protocol, three phases

- Read Phase
  - T is executed. Values copied and writes applied to T's private workspace
- Validation Phase
  - Before deciding on T's outcome, DBMS checks whether other transactions conflict with T
- Write Phase
  - If validation succeeds, writes in private workplace are applied to shared database

One protocol, three phases

- Validation Phase
  - transaction receive a timestams at the beginning of Validation phase
  - ensure all conflicts go in the same direction, either forward or backward
  - Before deciding on T's outcome, DBMS checks whether other transactions conflict with T
  - DBMS keeps track of read and write set of concurrently running transactions
  - Validate and Write phases executed inside a protected critical section

Validation

- Backward
  - check whether committing would cause a conflict with already committed transaction
- Forward
  - check whether committing would cause a conflict with currently running transactions
- Abort if validation fails

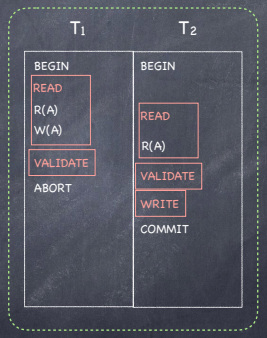
Conditions for successful Validation

- T_i succeeds if for all T_j running concurrently either
  - T_i completes all three phases before T_j begins

Conditions for successful Validation

- T_i succeeds if for all T_j running concurrently either
  - T_i completes all three phases before T_j begins
  - T_i completes its Write phase before T_j begins its Write phase, and

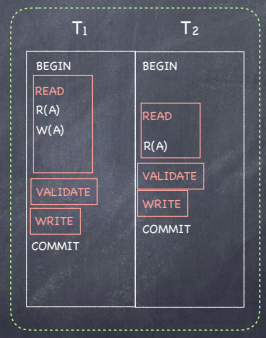
    $WriteSet(T_i) \cap ReadSet(T_j) = \varnothing$



Conditions for successful Validation

- T_i succeeds if for all T_j running concurrently either
  - T_i completes all three phases before T_j begins
  - T_i completes its Write phase before T_j begins its Write phase, and

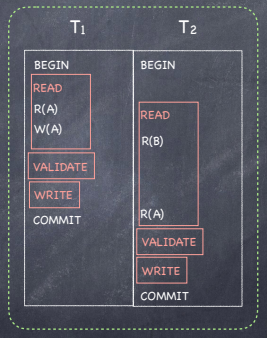
    $WriteSet(T_i) \cap ReadSet(T_j) = \varnothing$



Conditions for successful Validation

- T_i succeeds if for all T_j running concurrently either
  - T_i completes all three phases before T_j begins
  - T_i completes its Write phase before T_j begins its Write phase, and

    $WriteSet(T_i) \cap ReadSet(T_j) = \varnothing$



  - T_i completes its Read phase before T_j completes its Read phase, and

    $WriteSet(T_i) \cap ReadSet(T_j) = \varnothing$

    $WriteSet(T_i) \cap WriteSet(T_j) = \varnothing$

Musings on OCC

- Good when conflicts are few
  - transactions are read only
  - transactions access disjoint subsets of data
- Performance issues
  - high overhead for maintaining local workspace
  - Validation/Write phase can be a bottleneck
  - Aborts are more wasteful, because they occur after the entire transaction has executed
  - Allocating timestamps is a bottleneck