Database Systems
DATA 514

Transactions
Concurrency Control

Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
- Unit 3: RDMBS internals and query optimization
- Unit 4: Transactions
- Unit 5: Parallel query processing
- Unit 6: DBMS usability, conceptual design
- Unit 7: Non-relational data
- Unit 8: Advanced topics (time permitting)

Announcements

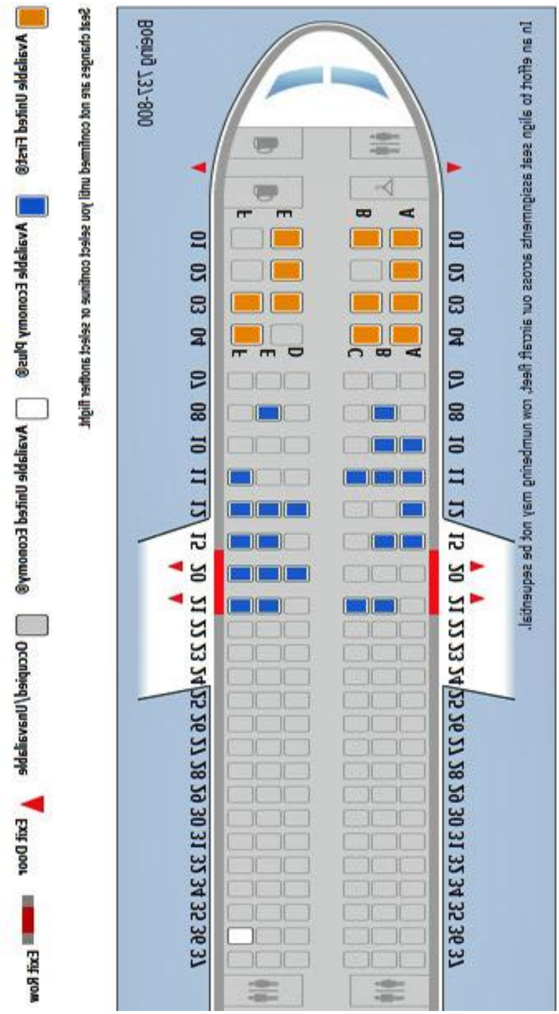
- HW5 due Monday, March 4

Demo
(see Concurrency demo.sql)

Schema: SeatAssignments
(Flightid, Customerid, seat, seatStatus)

Customer 1: views a map showing available seats

customer 1



```
Select seat, seatStatus
From SeatAssignments
Where Flightid = 1 And
seatStatus = 'Free'
```

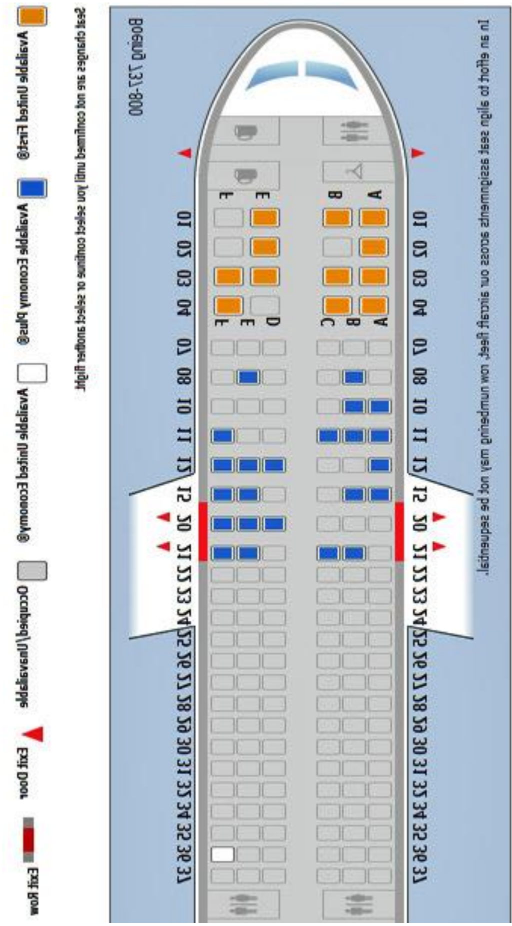
Concurrency

Customer 1: views a map showing available seats

customer 1

customer 2

?

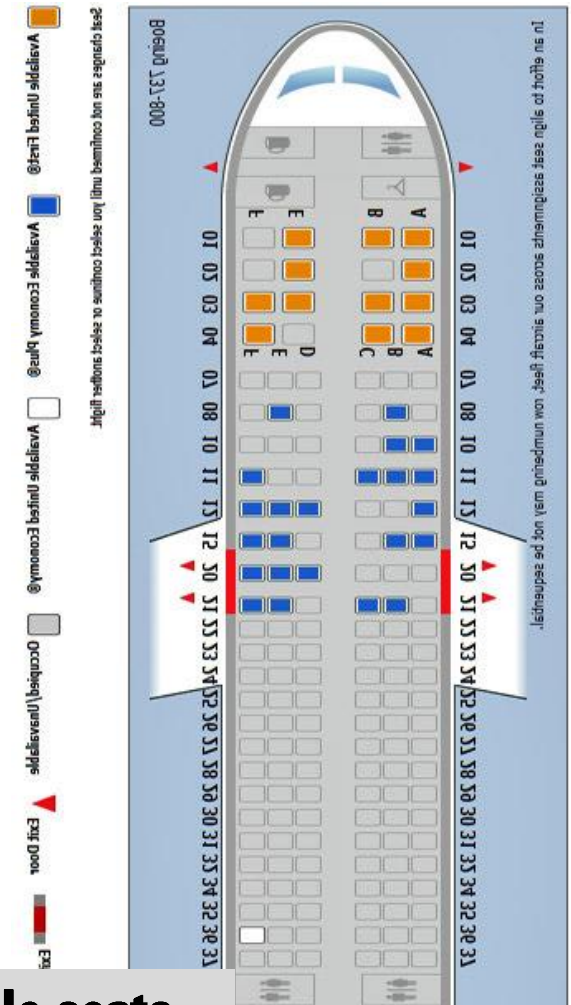


```
Select seat, seatStatus
From SeatAssignments
Where Flightid = 1 And seatStatus = 'Free'
```

Customer 2: requests the same map showing available seats

Concurrency

Customer 1: views a map showing available seats



Customer 2: requests the same map showing available seats

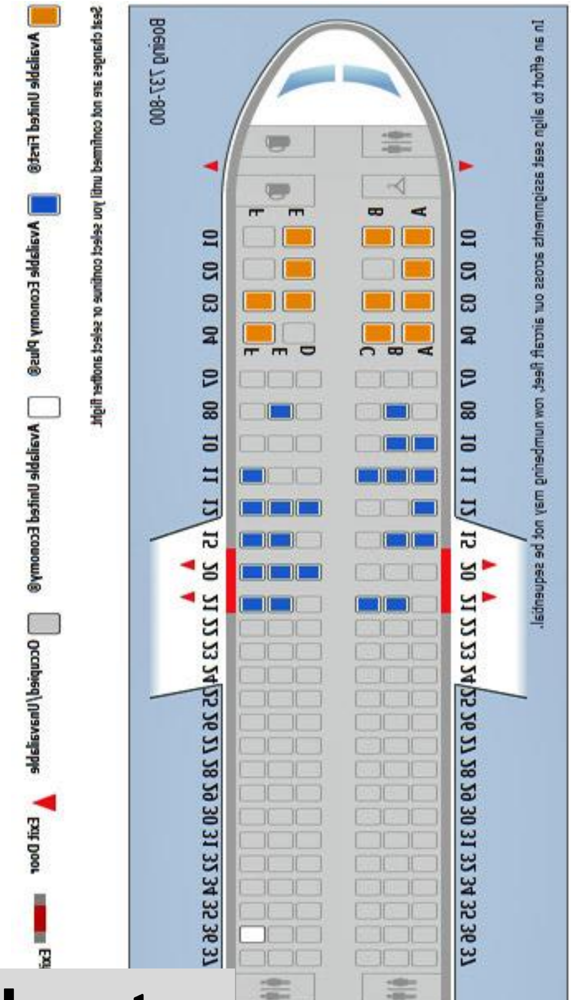
Write:Read Conflict

Customer 1: puts seat 4 on "Hold"

customer 1: seat 4; status = 'Hold'

```sql
Select seat, seatStatus
From SeatAssignments
Where Flightid = 1 And seatStatus = 'Free'
```

customer 2: Dirty Read?



Customer 2: requests the same map showing available seats

Serializability

- When the order in which requests are processed matters!
    - Serialize ⇨ order-preserving!
- Serialization is enforced using locks:
    - Writers acquire EXCLUSIVE locks
    - Readers acquire SHARED locks
        - Readers can opt out if they do not require a view of the data that is guaranteed to be coherent
        - i.e., reflects most recently committed changes, is repeatable, not dirty, etc.

Serializability

- Serialize ⇨ order-preserving!
- Serialization is enforced using locks:
  - Writers acquire **EXCLUSIVE** locks
  - Readers acquire **SHARED** locks
  - Requests block if the LOCK they require cannot be acquired
    - if a resource is currently **SHARED**, then an **EXCLUSIVE** lock request blocks
    - if a resource is currently **EXCLUSIVE**, then a **SHARED** lock request blocks
  - Requests that block long enough time out!

Serializability

- **Serialize ⇨ order-preserving!**
- **Serialization is enforced using locks:**
  - **Writers acquire EXCLUSIVE locks**
    - **Only one request at a time can hold an EXCLUSIVE lock**
  - **Readers acquire SHARED locks**
    - **Multiple Readers can each hold a SHARED lock on the same resource**
- **Lock management in a high volume, transaction processing DB is complex!**

Serializability

- Lock management in a high volume, transaction processing DB is complex!

High overhead!

- Lock contention impacts Request processing time
- Reduced levels of concurrency
- Time-outs
- Deadlocks

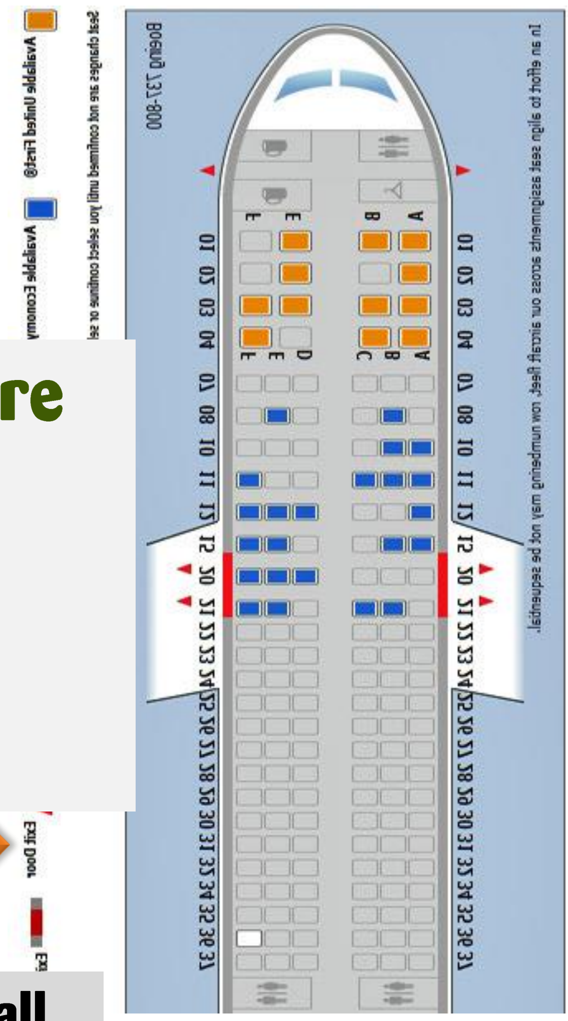
Lock granularity

- Lock management in a high volume, transaction processing DB is complex!
- High overhead:
    - Lock contention
    - Reduced levels of concurrency
- Best Practice: lock the fewest # of resources for the shorted possible period of time
    - However, most locking in a DBMS is implicit!

Serializability

Customer 1: puts seat 4 on "Hold"

customer 1: seat 4; status = 'Hold'



- **When the order in which requests are processed matters!**
- **Serialize ⇨ order-preserving!**

internal customer 2: Dirty Read?

Internal Customer 2: requests seat availability across all flights to between A and B?

Concurrent Read Anomalies

- Dirty Reads
  - Concurrent requests can see changes that have not yet been committed!
- Nonrepeatable Reads
  - Multiple Reads can yield inconsistent results; may not reflect current uncommitted changes
- Phantom Reads
  - Occurs when new rows are added or removed by another transaction to the record set while it is being read.

- Repeatable Reads: requires serializability!

Challenges

- Want to execute many apps concurrently
  - All these apps read and write data to the same DB
  - Simple solution: only serve one app at a time
    - Not very performant!
  - Better: multiple operations need to be executed atomically over the DB where access to data that is subject to change is serialized.

- Serialization:
  - despite handling many Requests in parallel, each Request is executed as if it is a serial process

Challenges

- **Serialization:**
  - despite handling many Requests in parallel, each Request appears to be executing as a serial process
    - i.e., order-preserving
  - the unit of serialization is
    - a single SQL statement
    - multiple SQL statements explicitly flagged as a "transaction"

Why is serialization necessary?

- Manager: balance budgets among projects
  - Remove $10k from project A
  - Add $7k to project B
  - Add $3k to project C

- CEO: check company's total balance
  - `SELECT SUM(money) FROM budget;`

- This is called a "dirty" (i.e., inconsistent) Read
  - aka **WRITE-READ** conflict

What can go wrong?

- **App 1:**
  `SELECT inventory FROM products WHERE pid = 1`

- **App 2:**
  `UPDATE products SET inventory = 0 WHERE pid = 1`

- **App 1:**
  `SELECT inventory * price FROM products`
  `WHERE pid = 1`

- **This is known as an unrepeatable read (aka a READ-WRITE conflict)**

What can go wrong?

Account 1 = $100
Account 2 = $100
Total = $200

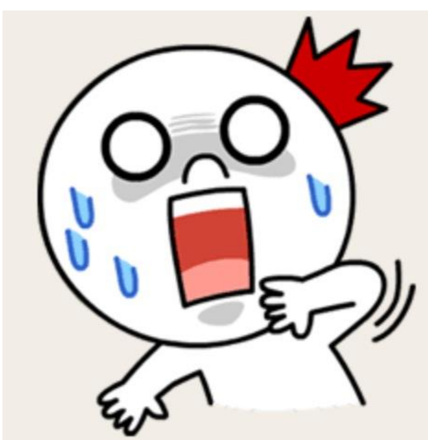
- **App 1:**
  - Set Account 1 = $200
  - Set Account 2 = $0
- **App 2:**
  - Set Account 2 = $200
  - Set Account 1 = $0
- **At the end:**
  - Total = $200

- **App 1:** Set Account 1 = $200
- **App 2:** Set Account 2 = $200
- **App 1:** Set Account 2 = $0
- **App 2:** Set Account 1 = $0
- **At the end:**
  - Total = $0

This is called the lost update, aka a WRITE-WRITE conflict

What can go wrong?

- Buying tickets to the next Cardi B. concert:
  - Fill up form with your mailing address
  - Put in debit card number
  - Click submit
  - Screen shows money deducted from your account
  - [Your browser crashes]



Atomicity: changes to the database may need to be ALL or NOTHING

Concurrent Data Access in SQL

- By default, all **Update**, **Insert** and **Delete** statements are atomic!
- What is the difference between

```
Update SeatAssignments
       Set seatStatus = 'Hold', Customerid = 1
WHERE Flightid = 1 and seat = 4 and seatStatus = 'Free'
```

- And -

```
Update SeatAssignments
       Set seatStatus = 'Hold', Customerid = 1
WHERE Flightid = 1 and seat = 4
```

?

Concurrent Data Access in SQL

- By default, all **Update**, **Insert** and **Delete** statements are atomic
  - implicit Transactions!
  - prevents Race conditions
  - enforced by Locking (Lock mode = **Exclusive**)
    - Note: Locking behavior is not specified explicitly in the SQL language

```
[LOCK mode='Exclusive']
Update SeatAssignments
      Set seatStatus = 'Hold', Customerid = 1
WHERE Flightid = 1 and seat = 4
      and seatStatus = 'Free'
```

Implied behavior

What needs to be Locked?

Concurrent Data Access in SQL

- By default, all Update, Insert and Delete statements are atomic!
  - prevents Race conditions
  - enforced by Locking (Lock mode = Exclusive)
- Read Policy (Select) ⇨ Isolation Levels in SQL
  - also enforced by Locking (Lock mode = Shared)
- Locking behavior (Lock Manager):
  - Exclusive lock waits until all prior Shared locks are unlocked
  - Shared locks block while an Exclusive lock is held

Isolation Levels in SQL

1. **"Dirty reads" permitted**

   `SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED`

2. **"Committed reads"**

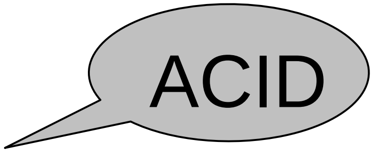
   `SET TRANSACTION ISOLATION LEVEL READ COMMITTED`

3. **"Repeatable reads"**

   `SET TRANSACTION ISOLATION LEVEL REPEATABLE READ`

4. **Serializable transactions**

   `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE`



1. Isolation Level: Read Uncommitted

- WRITE locks (of potentially long duration)
  - Strict 2PL (two-phase Locking)
- No READ locks
  - never delay Read-only requests

Possible problems: dirty and inconsistent reads

2. Isolation Level: Read Committed

- **WRITE** locks (of potentially long duration)
  - Strict 2PL (two-phase Locking)
- "Short duration" **READ** locks
  - acquire locks as necessary while Reading (not 2PL)

- Runs risk of **Unrepeatable reads**
  - i.e., When reading same element twice, may get two different values
  - Use cases?

- **Two-Phase Locking:**
  1. lock acquisition
  2. followed strictly by freeing all the locks acquired

3. Isolation Level: Repeatable Read

- "Long duration" WRITE locks
  - Strict 2PL
- "Long duration" READ locks
  - Strict 2PL

Why ?

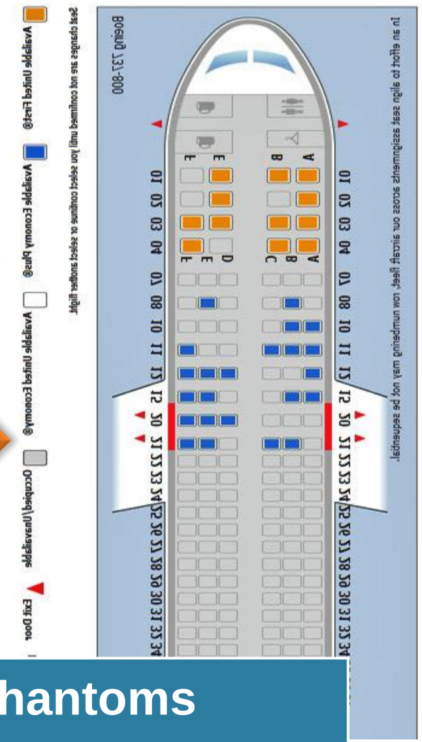
This is not serializable yet !!!

4. Isolation Level Serializable

- "Long duration" **WRITE** locks
  - Strict 2PL
- "Long duration" **READ** locks
  - Strict 2PL
- Predicate locking
  - To deal with phantoms
  - e.g.,

```
Select seat, seatStatus
From SeatAssignments
Where Flightid = 1 And seatStatus = 'Free'
```

Write:Read Conflict



Customer 1: puts seat 4 on "Hold"

customer 1: seat 4; status = 'Hold'

Customer 2?

Isolation Level	Dirty reads	Non-repeatable Reads	Phantoms
Read uncommitted			
Read committed	✗		
Repeatable Read	✗	✗	
Serializable	✗	✗	✗

Consequences of concurrency

- **Correctness Principle (see Molina, p. 847)**

> If a transaction that executes in the absence of any other transactions (and without system errors) starts with a database in a consistent state, then the database is also in a consistent state when the transaction is completed!

- **"Consistent state" refers to:**
  - **Tables/Relations**
  - **Disk blocks/pages (cf., "dirty pages")**
  - **individual Rows/tuples or other objects**

Consequences of concurrency

- Lock contention
  - Two requests executing concurrently that identify the same database object(s)
  - a request that requires a LOCK that cannot be granted is delayed (by the Scheduler)
  - a request that is delayed long enough will time-out!
  - deadlocks are possible:
    - T_a currently Holds(L_1); requests L_2
    - T_b currently Holds(L_2); requests L_1

Consequences of concurrency

- Interleaved execution
  - within a connection, Requests are executed in the order in which they are received
  - across connections, multiple Requests are executed in parallel!
  - creates potential for Resource contention:
    - Pages/Buffers
      - Sharing and “false” sharing
    - Logging

Locking strategies

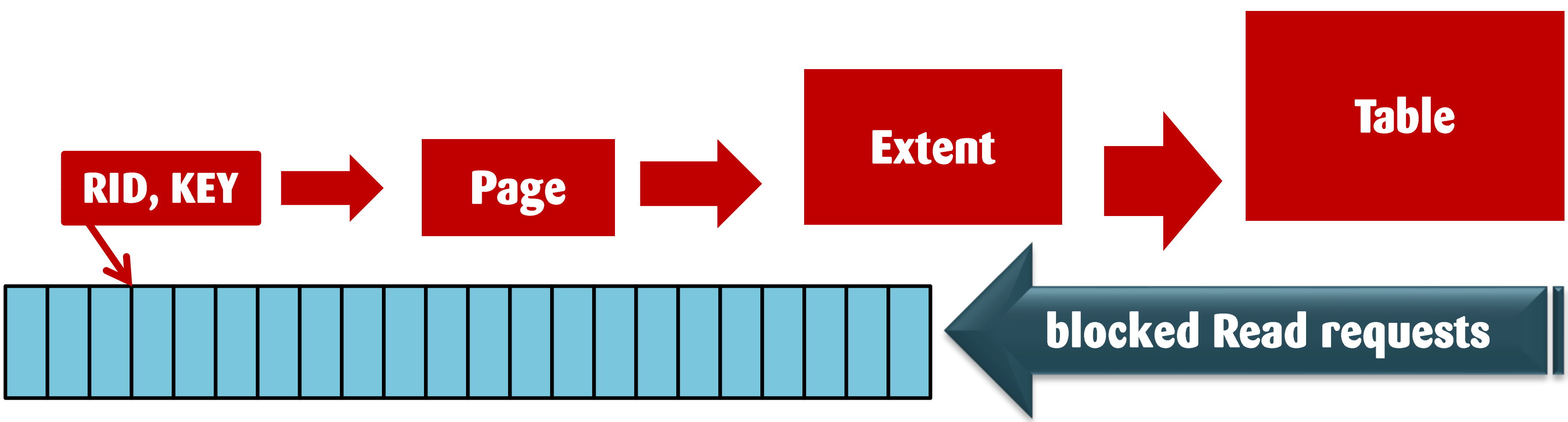
- Pessimistic
  - Acquire Read locks (lock mode = shared)
  - prohibit actions from other Requests that would change the data that the current Request identifies
- Optimistic
  - no Read locks necessary
  - if a conflict occurs due to a Write, rollback the change(s)
  - good performance when lock contention is limited

Concurrency and Locking

- Locking modes
  - SHARED
    - multiple Readers ⇨ many locks
  - EXCLUSIVE
    - only one Request at a time can hold an exclusive LOCK on Resource, r
    - no shared LOCKs for r are granted while an exclusive lock is held on r
  - UPDATE (MS SQL Server): reduces the likelihood of deadlocks

Concurrency and Locking

- **Lock granularity**
  - simple, one-level **SHARED/EXCLUSIVE** schemes do not work well with Indexes
  - Locks on b* trees (e.g., Index range scans) perform much better when there is a **lock hierarchy**
    - automatic LOCK escalation
      - fewer individual LOCKs to manage, but increased (potential) contention



SQL Server hierarchical locking

Resource	Description
RID	A row identifier used to lock a single row within a heap.
KEY	A row lock within an index used to protect key ranges in serializable transactions.
PAGE	An 8-kilobyte (KB) page in a database, such as data or index pages.
EXTENT	A contiguous group of eight pages, such as data or index pages.
HoBT	A heap or B-tree. A lock protecting a B-tree (index) or the heap data pages in a table that does not have a clustered index.
TABLE	The entire table, including all data and indexes.
FILE	A database file.
APPLICATION	An application-specified resource.
METADATA	Metadata locks.
ALLOCATION_UNIT	An allocation unit.
DATABASE	The entire database.

SQL Server lock modes

Lock mode	Description
Shared (S)	Used for read operations that do not change or update data, such as a SELECT statement.
Update (U)	Used on resources that can be updated. Prevents a common form of deadlock.
Exclusive (X)	Used for data-modification operations, such as INSERT, UPDATE, or DELETE. Ensures that multiple updates cannot be made to the same resource at the same time.
Intent	Used to establish a lock hierarchy. The types of intent locks are: intent shared (IS), intent exclusive (IX), and shared with intent exclusive (SIX).
Schema	Used when an operation dependent on the schema of a table is executing. The types of schema locks are: schema modification (Sch-M) and schema stability (Sch-S).
Bulk Update	Used when bulk copying data into a table and the TABLOCK hint is specified.
Key-range	Protects the range of rows read by a query when using the serializable transaction isolation level. Ensures that other transactions cannot insert rows that would qualify for the queries of the serializable transaction if the queries were run again.

DBMS Locking summary

- Lock acquisition and release is automatic!
  - Best practice: Locks of short duration!
- Example:
  - Automatic maintenance of time-series at the end of each Daily data file Push/DB update run

```
DELETE * FROM machinename@Process
WHERE TimeStamp < lastArchiveDate;
```

What is the scope of the lock?

DBMS Locking summary

- **Lock acquisition and release is automatic!**
  - **Best practice: Locks of short duration!**
- **Example:**

```
DELETE * FROM machinename@Process
WHERE TimeStamp < lastArchiveDate;
```

Risk of time-outs increases *after* the first failure

Improved version:

```
deleteDate = LastArchiveDate
While (deleteDate > lastDateinTable)
{
        DELETE * FROM machinename@Process
        WHERE TimeStamp.Date = deleteDate;

deleteDate = deleteDate.Date – 1;
} Do;
```

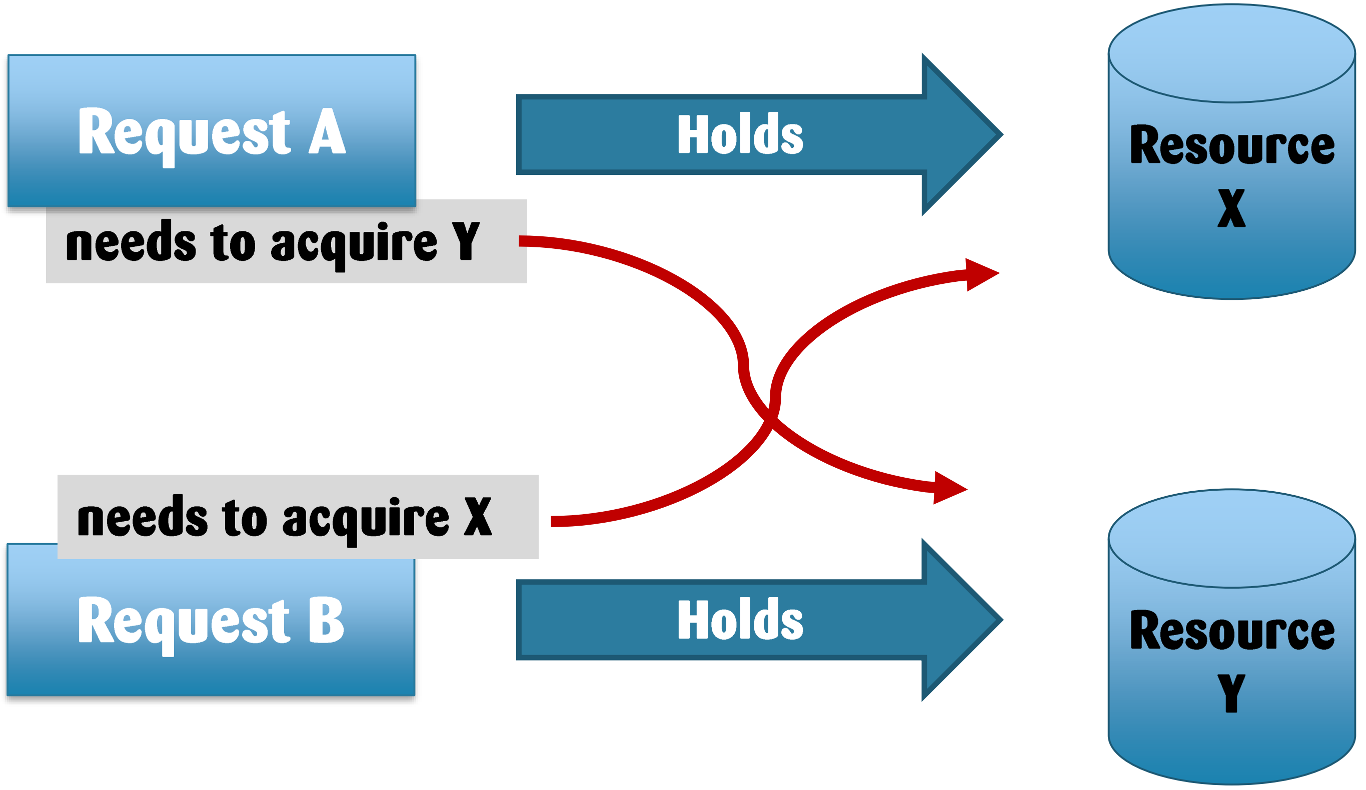
Lock Time-outs

- Requests delayed by contention for locks are subject to time-out.
  - [Shared] blocks [Exclusive]
  - [Exclusive] blocks [Shared]
  - Recover and Retry?

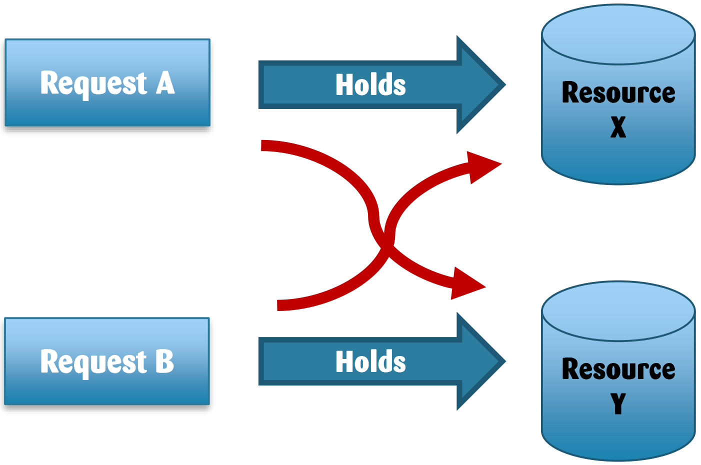
- Note: (long running) Requests can time out for other reasons, too.

Lock Deadlocks

- aka "deadly embrace"



Lock Deadlocks



- **Time-out mechanism:**
  - Request A times out
  - Request B can now proceed
    - Request B becomes more likely to time-out, too.

- **Unraveling the chain of locking events that led to a deadly embrace can be very difficult!**
  - implicit locking
  - lock escalation
  - nested transactions
  - etc.

DBMS Locking summary

- Shared/Exclusive modes
  - additional auxiliary modes can reduce # of deadlocks
  - Optimistic locking policies reduce lock contention, but increases the amount of work the DBMS must perform when contention is detected

- Hierarchical (with automatic escalation)
  - b+ tree data structure (i.e., Indexes)
  - escalation reduces the # of Locks to keep track of,
  - but increases the potential for lock contention

DBMS Locking summary

- Lock contention is often the most serious performance problem for databases that must sustain high transaction processing rates!
  - e.g., real-time trading and other auctions

- Best practice: ensure Write `[Exclusive]` Locks are of short duration!
  - Find out where lock-related Time-outs are occurring
  - Complications from "explicit" (multi-statement) Transactions
  - Do not keep a transaction pending awaiting user input!

DBMS Locking summary

- Best practice: extract data from active production R/W DBs to construct Read-only data warehouses
  - Single Writer task (stream, Bulk update or Batch)
  - Multiple Readers
  - Non-repeatable Reads are OK!
    - no [SHARED] Locks
    - no contention
    - reduced overheads
  - Do not time-out Long Duration reads

Explicit Transactions

- multiple SQL statements can be defined to execute in sequence as a single, atomic unit

```
BEGIN TRANSACTION
     [SQL statement]
     [SQL statement]
     [SQL statement]
     [SQL statement]…

COMMIT     or     ROLLBACK (=ABORT)
```

- Note: explicit transactions can be nested!

Rollback a Transaction

- Initiated by the applications or by the system
- The DB returns to the state prior to the start of the transaction
  - Intermediate DB changes are backed out!
- What are examples?

Lock Time-outs

- Requests delayed by contention for locks are subject to time-out.
  - `[Shared]` blocks `[Exclusive]`
  - `[Exclusive]` blocks `[Shared]`
  - Recover and Retry?

- Complication:
  - Time-out in the middle of a multi-statement Transaction
  - Abort the operation and Rollback?

Transactions Demo
(see Concurrency demo.sql)

ACID

- Atomic
  - State shows either all the effects of txn, or none of them
- Consistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- Isolated
  - Effect of txns is the same as txns running one after another (i.e., looks like batch mode)
- Durable
  - Once a txn has committed, its effects remain in the database

Atomic

- **Definition: A transaction is ATOMIC if all its updates must happen or not at all.**
- **Example: transfer $100 from A to B**

```
UPDATE accounts SET bal = bal – 100
WHERE acct = A;
UPDATE accounts SET bal = bal + 100
WHERE acct = B;
```

Crash!

```
BEGIN TRANSACTION;
UPDATE accounts SET bal = bal – 100 WHERE acct = A;
UPDATE accounts SET bal = bal + 100 WHERE acct = B;
COMMIT;
```

Isolated

- **Definition: An execution ensures that txns are isolated, if the effect of each txn is as if it were the only txn running on the system.**
  - **Example: Alice deposits $100, Bob withdraws $100 from a joint account**

Alice:

```
BEGIN TRANSACTION;
x = select bal from accounts
    where acct = A;
x = x+100
update accounts
  set bal = x where acct = A;
COMMIT;
```

Bob:

```
BEGIN TRANSACTION;
y = select bal from accounts
    where acct = A;
if y < 100 return "Error"
y = y - 100
update accounts
  set bal = y where acct = A;
COMMIT;
```

Consistent

- Recall: integrity constraints govern how values in tables are related to each other
  - Can be enforced by the DBMS, or ensured by the app
- Example: account.bal >= 0

- How consistency is achieved by the app:
  - App programmer ensures that txns only takes a consistent DB state to another consistent state
  - DB makes sure that txns are executed atomically

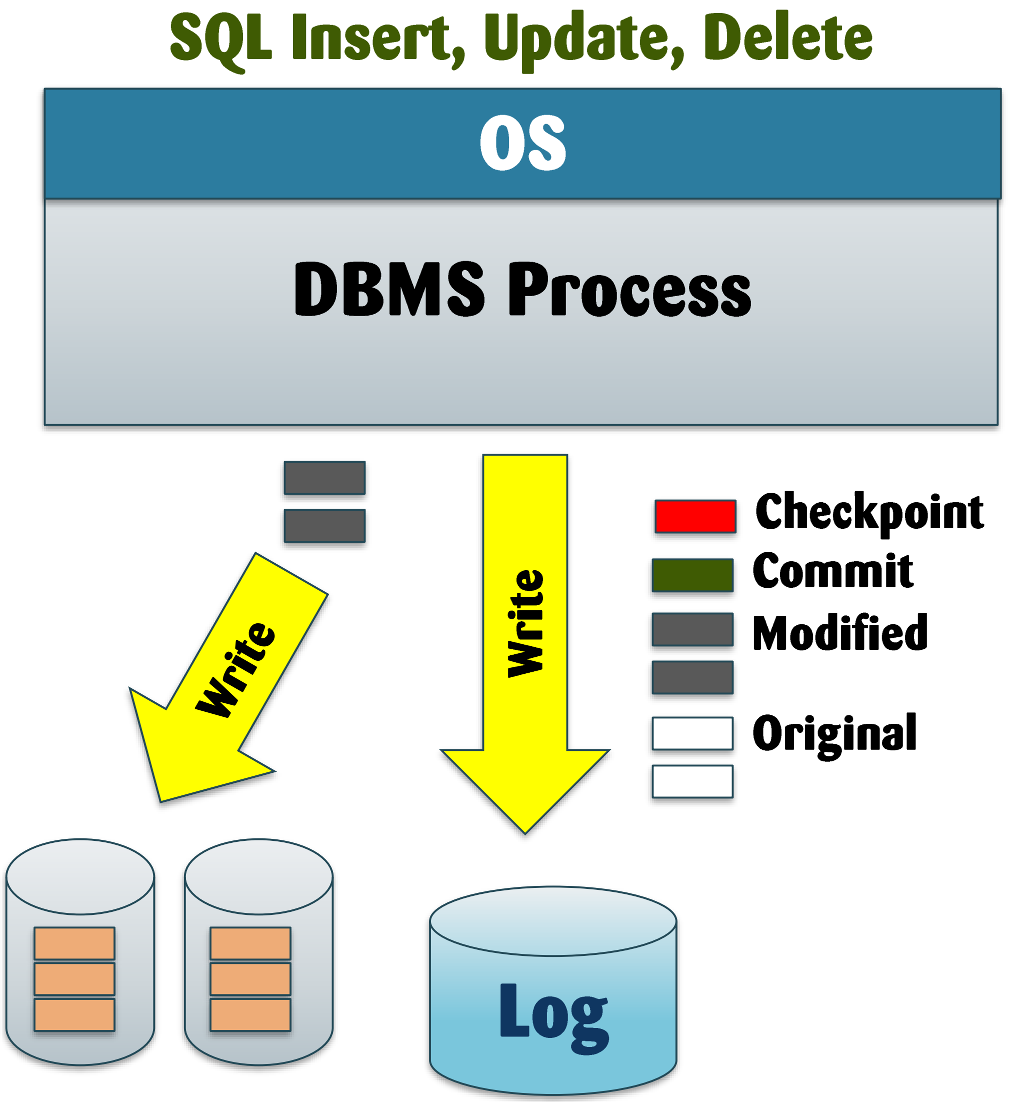
- Can defer checking the validity of constraints until the end of a transaction

Durable

- A transaction is durable if its effects continue to exist after the transaction and even after the program has terminated
- How? By writing to disk

Transaction Recovery Log

- **Writing the Log:**
  - "Before" page images
  - Changed pages
  - Change status
    - Commit
    - Rollback
  - Checkpoint
    - indicates that the DB is in a consistent state!



ACID

Atomic
Consistent
Isolated
Durable

Enjoy this in HW!

Note:

- By default each statement is its own transaction
- If auto-commit=off, then each statement starts a new transaction

Isolation: The Problem

- Multiple transactions are running concurrently
  T_1, T_2, …

- They read/write some common elements
  A_1, A_2, …

- How can we prevent unwanted interference ?
- The SCHEDULER is responsible for that

Schedules

A schedule is a sequence of interleaved actions from multiple transactions

Serial Schedule

- A **serial schedule** is one in which transactions are executed one after the other, in some sequential order

- Fact: nothing can go wrong if the system executes transactions serially
  - But database systems don't do that because we need better performance

Example

A and B are elements in the database
t and s are variables in txn source code

T1	T2
READ(A, t)	READ(A, s)
t := t+100	s := s*2
WRITE(A, t)	WRITE(A,s)
READ(B, t)	READ(B,s)
t := t+100	s := s*2
WRITE(B,t)	WRITE(B,s)

A Serial Schedule

Time ↓

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)

Another Serial Schedule

Time

T1	T2
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(A, t)	
t := t+100	
WRITE(A, t)	
READ(B, t)	
t := t+100	
WRITE(B,t)	

Serializable Schedule

A schedule is *serializable* if it is equivalent to a serial schedule

A Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	
	READ(B,s)
	s := s*2
	WRITE(B,s)

This is a serializable schedule.
This is NOT a serial schedule

A Non-Serializable Schedule

T1	T2
READ(A, t)	
t := t+100	
WRITE(A, t)	
	READ(A,s)
	s := s*2
	WRITE(A,s)
	READ(B,s)
	s := s*2
	WRITE(B,s)
READ(B, t)	
t := t+100	
WRITE(B,t)	

How do We Know if a Schedule is Serializable?

Notation

T_1: $r_1(A)$; $w_1(A)$; $r_1(B)$; $w_1(B)$
T_2: $r_2(A)$; $w_2(A)$; $r_2(B)$; $w_2(B)$

Key Idea: Focus on *conflicting* operations

Conflicts

- Write-Read – WR
- Read-Write – RW
- Write-Write – WW

Conflict Serializability

Conflicts: (it means: cannot be swapped)

Two actions by same transaction T_i: $r_i(X); w_i(Y)$

Two writes by T_i, T_j to same element $w_i(X); w_j(X)$

Read/write by T_i, T_j to same element $w_i(X); r_j(X)$

$r_i(X); w_j(X)$

Conflict Serializability

- A schedule is ***conflict serializable*** if it can be transformed into a serial schedule by a series of swappings of adjacent non-conflicting actions

- Every conflict-serializable schedule is serializable
- A serializable schedule may not necessarily be conflict-serializable

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

↓

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Conflict Serializability

Example:

$r_1(A); w_1(A); r_2(A); w_2(A); r_1(B); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_2(A); r_1(B); w_2(A); w_1(B); r_2(B); w_2(B)$

$r_1(A); w_1(A); r_1(B); r_2(A); w_2(A); w_1(B); r_2(B); w_2(B)$

....

$r_1(A); w_1(A); r_1(B); w_1(B); r_2(A); w_2(A); r_2(B); w_2(B)$

Testing for Conflict-Serializability

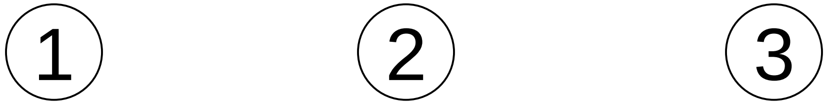
Precedence graph:

- A node for each transaction T_i,
- An edge from T_i to T_j whenever an action in T_i conflicts with, and comes before an action in T_j

- The schedule is serializable iff the precedence graph is acyclic

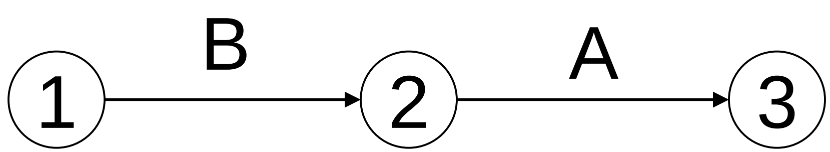
Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



Example 1

$r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B)$



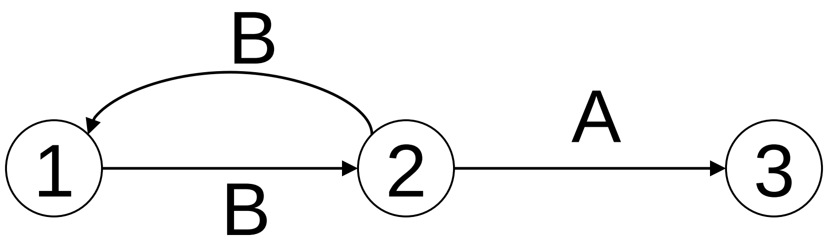
This schedule is conflict-serializable

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$

Example 2

$r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B)$



This schedule is NOT conflict-serializable

Scheduler

- **Scheduler** = is the module that schedules the transaction's actions, ensuring serializability
- Also called **Concurrency Control Manager**
- We discuss next how a scheduler may be implemented

Implementing a Scheduler

Major differences between database vendors

- Locking Scheduler
  - Aka "pessimistic concurrency control"
  - SQLite, SQL Server, DB2
- Multiversion Concurrency Control (MVCC)
  - Aka "optimistic concurrency control"
  - Postgres, Oracle

We discuss only locking

Locking Scheduler

Simple idea:

- Each element has a unique **lock**
- Each transaction must first **acquire** the lock before reading/writing that element
- If the lock is taken by another transaction, then wait
- The transaction must **release** the lock(s)

By using locks scheduler ensures conflict-serializability

What Data Elements are Locked?

Major differences between vendors:

- Lock on the entire database
  - SQLite
- Lock on individual records
  - SQL Server, DB2, etc

Notation

$L_i(A)$ = transaction T_i acquires lock for element A

$U_i(A)$ = transaction T_i releases lock for element A

A Non-Serializable Schedule

T1	T2
READ(A)	
A := A+100	
WRITE(A)	
	READ(A)
	A := A*2
	WRITE(A)
	READ(B)
	B := B*2
	WRITE(B)
READ(B)	
B := B+100	
WRITE(B)	

Example

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$; $L_1(B)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;

Scheduler has ensured a conflict-serializable schedule

But…

T1	T2
$L_1(A)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$;	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A); $U_2(A)$;
	$L_2(B)$; READ(B)
	B := B*2
	WRITE(B); $U_2(B)$;
$L_1(B)$; READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	

Locks did not enforce conflict-serializability !!! What's wrong ?

Two Phase Locking (2PL)

The 2PL rule:

In every transaction, all lock requests must precede all unlock requests

Example: 2PL transactions

T1	T2
$L_1(A)$; $L_1(B)$; READ(A)	
A := A+100	
WRITE(A); $U_1(A)$	
	$L_2(A)$; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; BLOCKED…
READ(B)	
B := B+100	
WRITE(B); $U_1(B)$;	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); $U_2(A)$; $U_2(B)$;

Now it is conflict-serializable

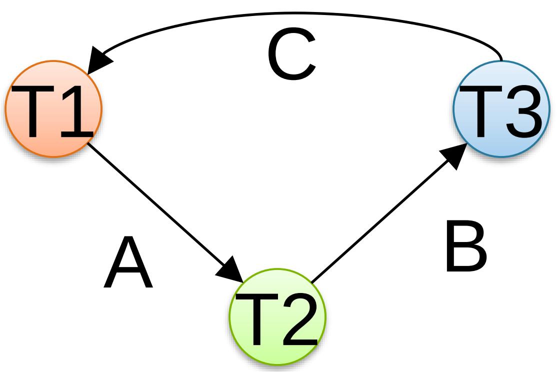
Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

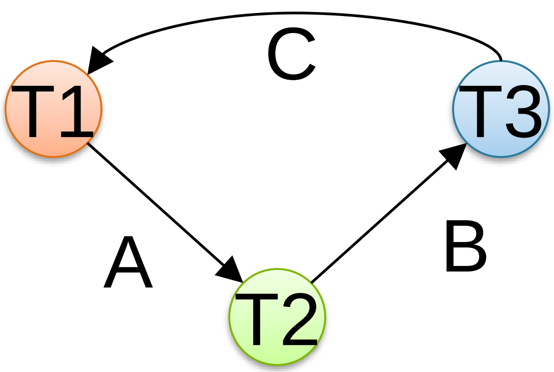
Proof. Suppose not: then there exists a cycle in the precedence graph.



Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.

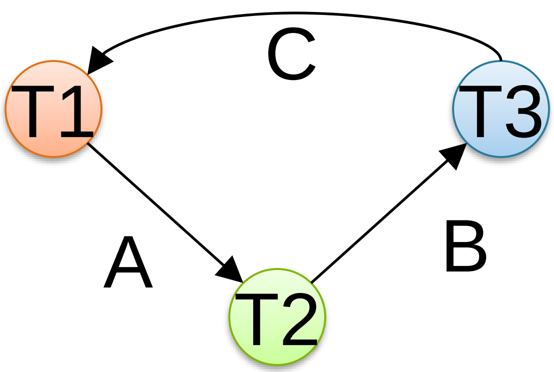


Then there is the following **temporal** cycle in the schedule:

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



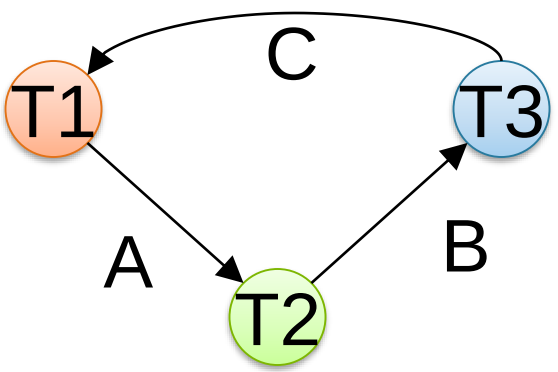
Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$ why?

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following **temporal** cycle in the schedule:

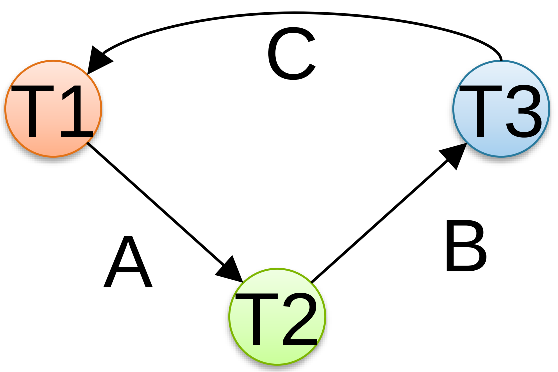
$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$ why?

Two Phase Locking (2PL)

Theorem: 2PL ensures conflict serializability

Proof. Suppose not: then there exists a cycle in the precedence graph.



Then there is the following **temporal** cycle in the schedule:

$U_1(A) \rightarrow L_2(A)$

$L_2(A) \rightarrow U_2(B)$

$U_2(B) \rightarrow L_3(B)$

$L_3(B) \rightarrow U_3(C)$

$U_3(C) \rightarrow L_1(C)$

$L_1(C) \rightarrow U_1(A)$

Contradiction

A New Problem: Non-recoverable Schedule

T1	T2
L_1(A); L_1(B); READ(A)	
A :=A+100	
WRITE(A); U_1(A)	
	L_2(A); READ(A)
	A := A*2
	WRITE(A);
	L_2(B); BLOCKED…
READ(B)	
B :=B+100	
WRITE(B); U_1(B);	
	…GRANTED; READ(B)
	B := B*2
	WRITE(B); U_2(A); U_2(B);
	Commit
Rollback	

Strict 2PL

The Strict 2PL rule:

> All locks are held until the transaction commits or aborts.

With strict 2PL, we will get schedules that are both conflict-serializable and recoverable

Strict 2PL

T1	T2
$L_1(A)$; READ(A)	
A :=A+100	
WRITE(A);	
	$L_2(A)$; BLOCKED…
$L_1(B)$; READ(B)	
B :=B+100	
WRITE(B);	
$U_1(A)$,$U_1(B)$; Rollback	
	…GRANTED; READ(A)
	A := A*2
	WRITE(A);
	$L_2(B)$; READ(B)
	B := B*2
	WRITE(B);
	$U_2(A)$; $U_2(B)$; Commit

Another problem: Deadlocks

- T_1 waits for a lock held by T_2;
- T_2 waits for a lock held by T_3;
- T_3 waits for . . . .
- . . .
- T_n waits for a lock held by T_1

SQL Lite: there is only one exclusive lock; thus, never deadlocks

SQL Server: checks periodically for deadlocks and aborts one TXN

Lock Modes

- **S** = shared lock (for READ)
- **X** = exclusive lock (for WRITE)

Lock compatibility matrix:

	None	S	X
None			
S			
X			

Lock Modes

- **S** = shared lock (for READ)
- **X** = exclusive lock (for WRITE)

Lock compatibility matrix:

	None	S	X
None	✔	✔	✔
S	✔	✔	✘
X	✔	✘	✘

Lock Granularity

- Fine granularity locking (e.g., tuples)
  - High concurrency
  - High overhead in managing locks
  - E.g. SQL Server
- Coarse grain locking (e.g., tables, entire database)
  - Many false conflicts
  - Less overhead in managing locks
  - E.g. SQL Lite
- Solution: lock escalation changes granularity as needed

Sqlite

- SQLite is very simple
- More info: http://www.sqlite.org/atomiccommit.html
- Lock types
  - READ LOCK (to read)
  - RESERVED LOCK (to write)
  - PENDING LOCK (wants to commit)
  - EXCLUSIVE LOCK (to commit)

Sqlite

Step 1: when a transaction begins

- Acquire a **READ LOCK** (aka "SHARED" lock)
- All these transactions may read happily
- They all read data from the database file
- If the transaction commits without writing anything, then it simply releases the lock

Sqlite

Step 2: when one transaction wants to write

- Acquire a **RESERVED LOCK**
- May coexists with many READ LOCKs
- Writer TXN may write; these updates are only in main memory; others don't see the updates
- Reader TXN continue to read from the file
- New readers accepted
- No other TXN is allowed a RESERVED LOCK

Sqlite

Step 3: when writer transaction wants to commit, it needs *exclusive lock*, which can't coexists with *read locks*

- Acquire a **PENDING LOCK**
- May coexists with old READ LOCKs
- No new READ LOCKS are accepted
- Wait for all read locks to be released

Why not write to disk right now?

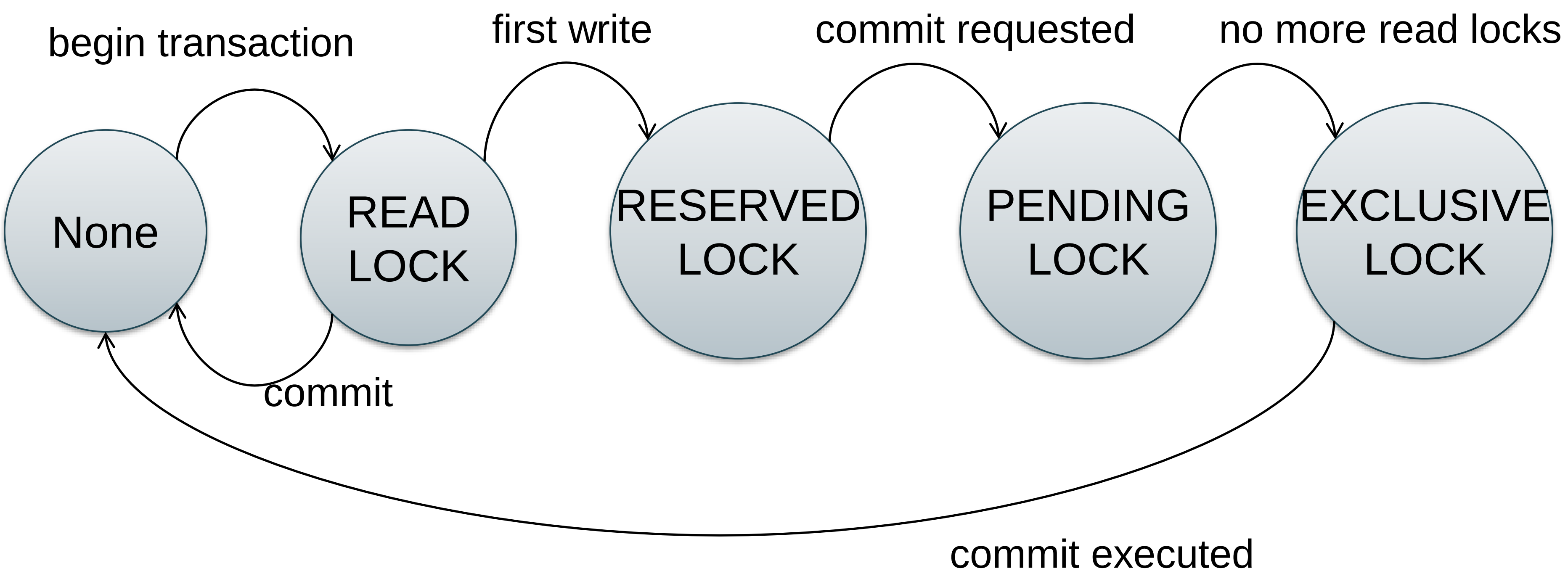
Sqlite

Step 4: when all read locks have been released

- Acquire the **EXCLUSIVE LOCK**
- Nobody can touch the database now
- All updates are written permanently to the database file

- Release the lock and **COMMIT**

Sqlite


begin transaction
first write
commit requested
no more read locks
None
READ LOCK
RESERVED LOCK
PENDING LOCK
EXCLUSIVE LOCK
commit
commit executed

Sqlite Demo

```
create table r(a int, b int);
insert into r values (1,10);
insert into r values (2,20);
insert into r values (3,30);
```

Demonstrating Locking in SQLite

T1:

```
begin transaction;
select * from r;
-- T1 has a READ LOCK
```

T2:

```
begin transaction;
select * from r;
-- T2 has a READ LOCK
```

Demonstrating Locking in SQLite

T1:

update r set b=11 where a=1;

-- T1 has a RESERVED LOCK

T2:

update r set b=21 where a=2;

-- T2 asked for a RESERVED LOCK: DENIED

Demonstrating Locking in SQLite

T3:

```
begin transaction;
select * from r;
commit;
-- everything works fine, could obtain READ LOCK
```

Demonstrating Locking in SQLite

T1:

```
commit;
-- SQL error: database is locked
-- T1 asked for PENDING LOCK -- GRANTED
-- T1 asked for EXCLUSIVE LOCK -- DENIED
```

Demonstrating Locking in SQLite

T3':

begin transaction;

select * from r;

-- T3 asked for READ LOCK-- DENIED (due to T1)

T2:

commit;

-- releases the last READ LOCK; T1 can commit

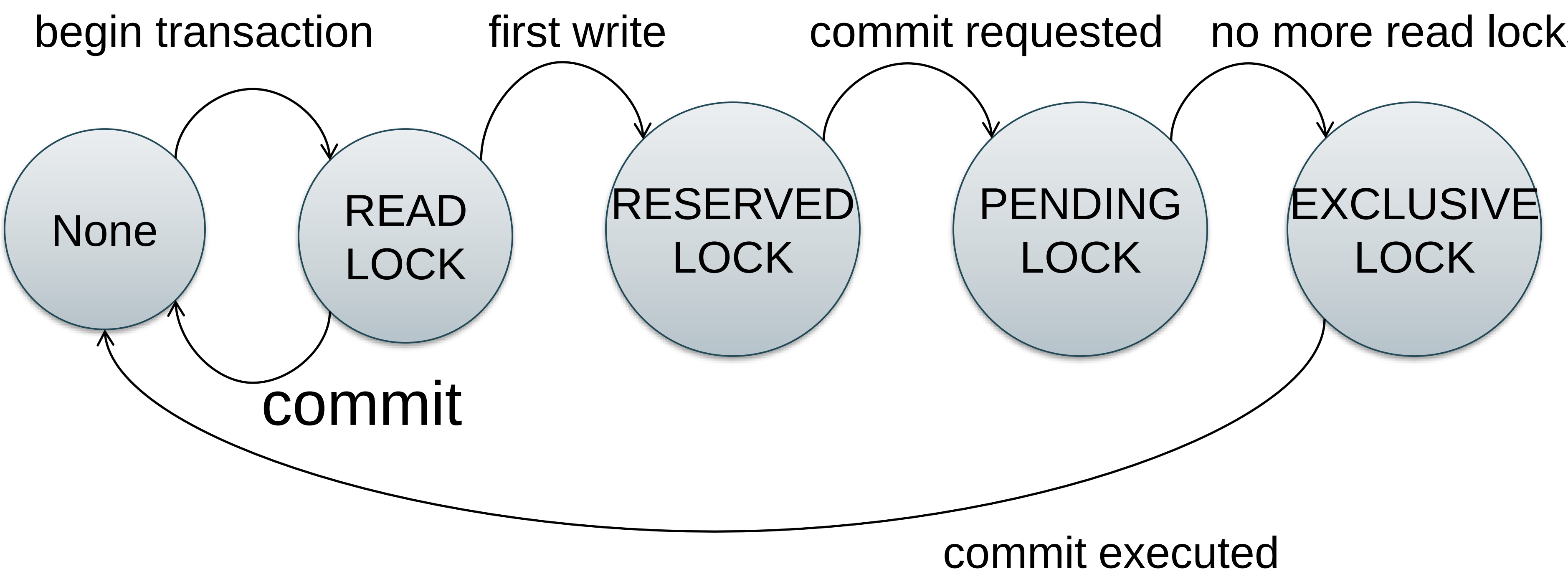
Recap

- What are transactions
  - And why do we need them

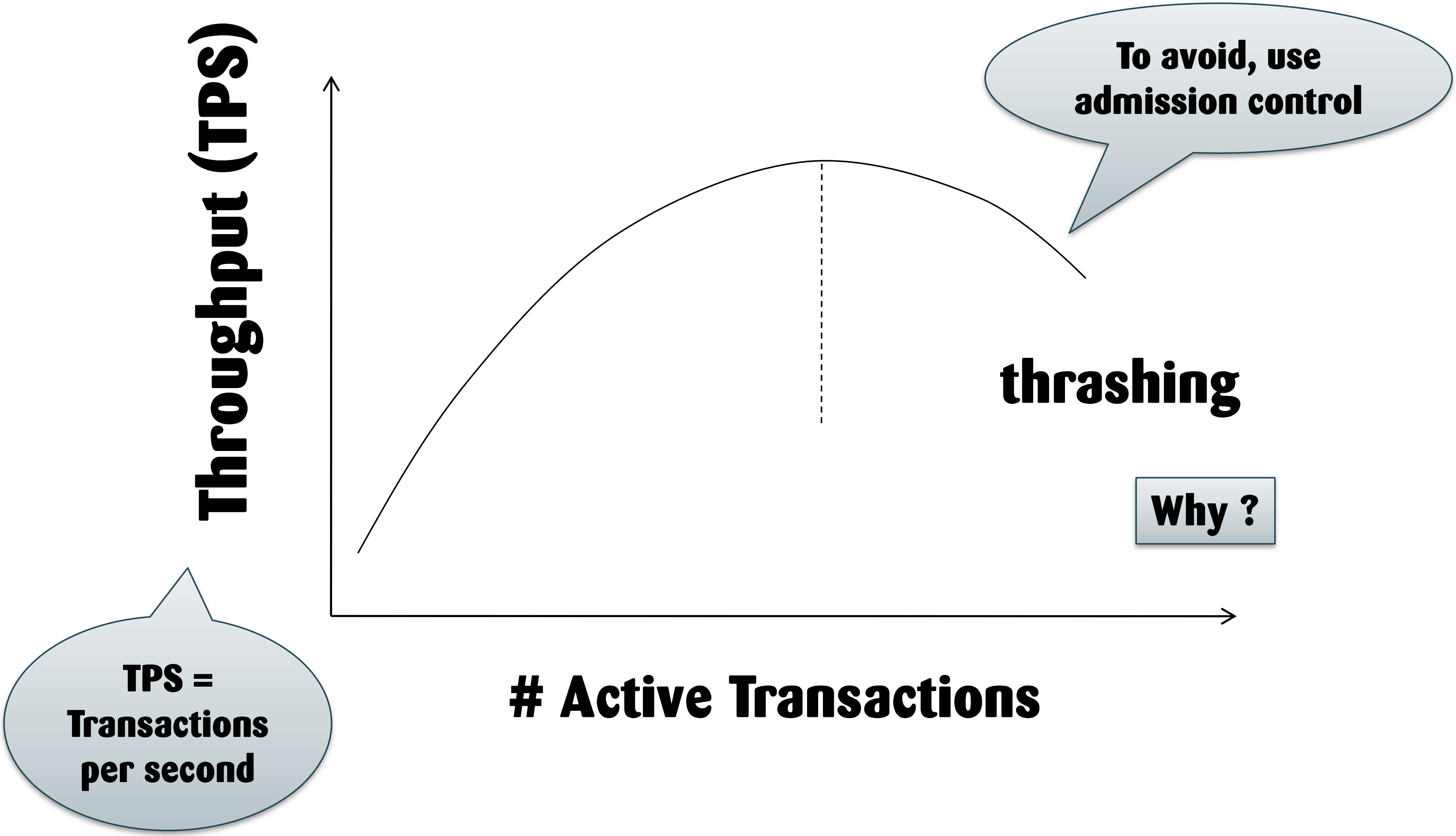
- How to maintain ACID properties via schedules
  - We focus on the isolation property
  - We do not discuss atomicity

- How to ensure conflict-serializable schedules with locks

SQLite


begin transaction
first write
commit requested
no more read lock
None
READ LOCK
RESERVED LOCK
PENDING LOCK
EXCLUSIVE LOCK
commit
commit executed

Lock Performance



Phantom Problem

- So far we have assumed the database to be a **static** collection of elements (=tuples)
- If tuples are inserted/deleted then the **phantom problem** appears

Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘A3’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

Is this schedule serializable ?

Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color='blue'	
	INSERT INTO Product(name, color) VALUES ('A3','blue')
SELECT * FROM Product WHERE color='blue'	

R1(A1),R1(A2),W2(A3),R1(A1),R1(A2),R1(A3)

Suppose there are two blue products, A1, A2:

Phantom Problem

T1	T2
SELECT * FROM Product WHERE color=‘blue’	
	INSERT INTO Product(name, color) VALUES (‘A3’,’blue’)
SELECT * FROM Product WHERE color=‘blue’	

R1(A1),R1(A2),W2(A3),R1(A1),R1(A2),R1(A3)

W2(A3),R1(A1),R1(A2),R1(A1),R1(A2),R1(A3)

Phantom Problem

- A "phantom" is a tuple that is invisible during part of a transaction execution but not invisible during the entire execution

- In our example:
  - T1: reads list of products
  - T2: inserts a new product
  - T1: re-reads: a new product appears !

Dealing With Phantoms

- Lock the entire table
- Lock the index entry for 'blue'
  - If index is available
- Or use predicate locks
  - A lock on an arbitrary predicate

Dealing with phantoms is expensive !

Beware!

In commercial DBMSs:

- Default level is often NOT serializable
- Default level differs between DBMSs
- Some engines support subset of levels!
- Serializable may not be exactly ACID
  - Locking ensures isolation, not atomicity
- Also, some DBMSs do NOT use locking and different isolation levels can lead to different pbs
- Bottom line: Read the doc for your DBMS!

Next two slides: try them on SQL Azure

Demonstration with SQL Server

Application 1:
create table R(a int);
insert into R values(1);
set transaction isolation level serializable;
begin transaction;
select * from R; -- get a shared lock

Application 2:
set transaction isolation level serializable;
begin transaction;
select * from R; -- get a shared lock
insert into R values(2); -- blocked waiting on exclusive lock
-- App 2 unblocks and executes insert after app 1 commits/aborts

Demonstration with SQL Server

Application 1:

```
create table R(a int);
insert into R values(1);
set transaction isolation level repeatable read;
begin transaction;
select * from R; -- get a shared lock
```

Application 2:

```
set transaction isolation level repeatable read;
begin transaction;
select * from R; -- get a shared lock
insert into R values(3); -- gets an exclusive lock on new tuple
        -- If app 1 reads now, it blocks because read dirty
        -- If app 1 reads after app 2 commits, app 1 sees new value
```