Transaction Management

F28DM Database Management Systems

Matthew P Aylett

m.aylett@hw.ac.uk

Materials from: Alasdair G J Gray

What is a transaction?

- Series of operations to perform a task
- Logical unit of work

http://www.strategicdealslawblog.com/files/2012/09/dreamstime_m_8745845.jpg

Example: Money Transfer

Move £100 between two accounts

R/W
```
1. UPDATE Account SET
   balance = balance – 100
   WHERE accountNo = 123;
```

R/W
```
2. UPDATE Account SET
   balance = balance + 100
   WHERE accountNo = 124;
```

What would happen if only the first query executed?

Example: Web Purchase

1. Create order
2. Add customer details to order
3. Place items from basket into order
4. Take payment

Why use transactions?

- Ensure data integrity
  - Require multiple updates to appear as one
  - Ensure integrity constraints, e.g. referential integrity (FK)
- Support concurrent access
  - Allow multiple users!
  - More throughput
- Recovery, e.g. system crash
  - Avoid data loss

https://d3glfbbr3jeumb.cloudfront.net/assets/features/concurrent-connections-208bcc5d5db456d66914e808c42a4c05.png

User Interactions

- User Interaction:
  - entering values
  - choosing a value from a list
  - clicking a button
- Users are orders of magnitude slower than computers
  - Reduces throughput

Causes problems for transaction processing!

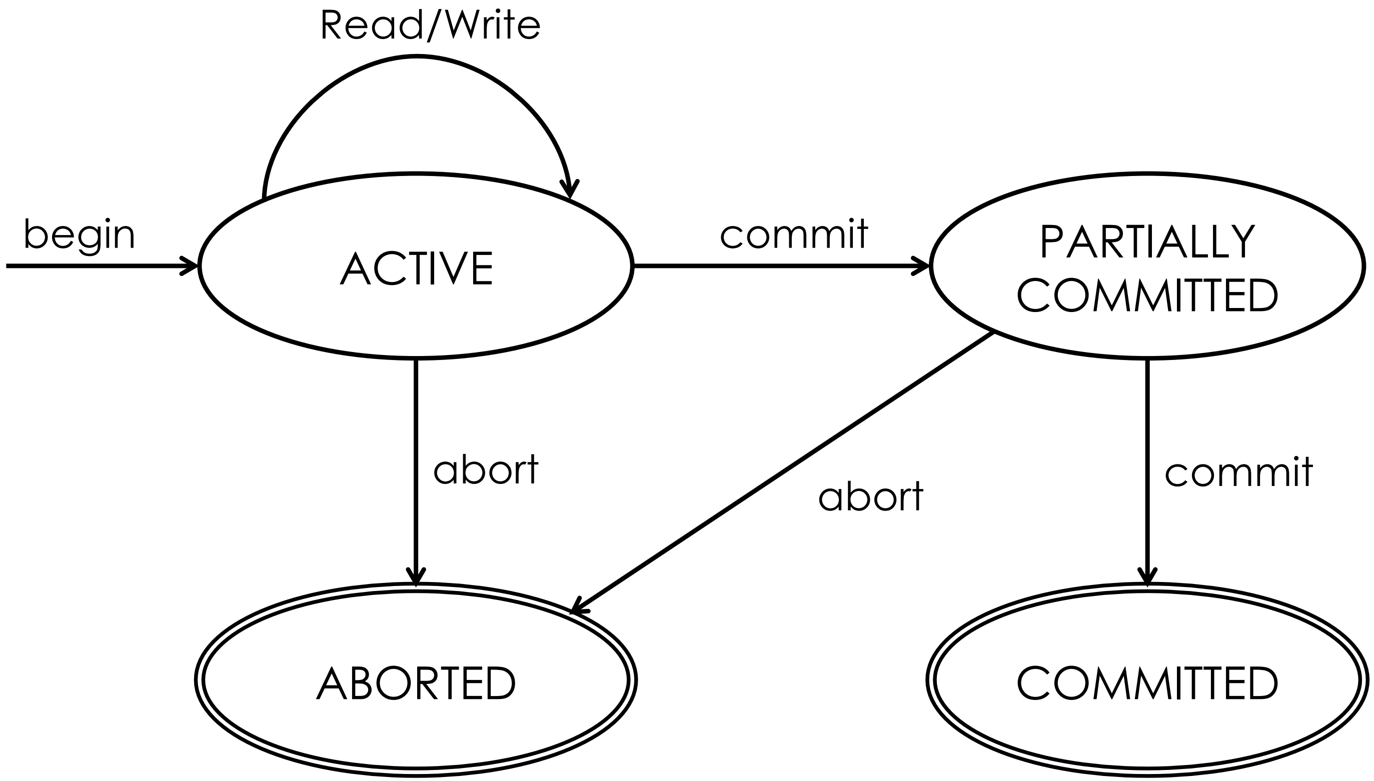
Problem 1: Lost Update

Transaction A	Transaction B	Value V
Get V		5
	Get V	5
Add 10		5
	Add 15	5
Put V		
	Put V	

Transaction Properties: ACID

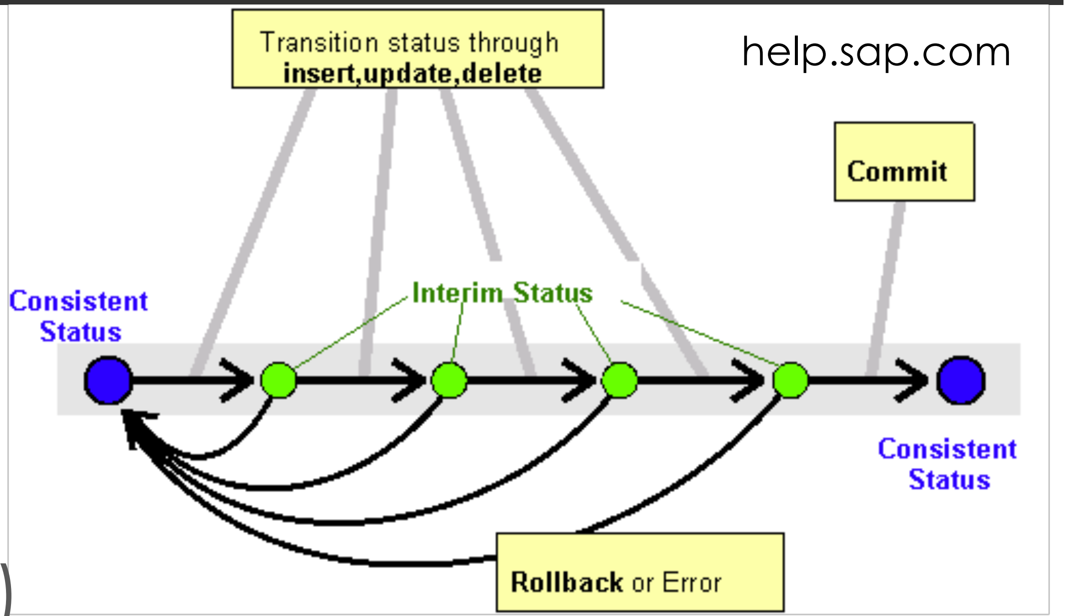
- **Atomicity:** All actions complete or none
  - All or nothing
- **Consistency:** Database finishes in a consistent state, i.e. no integrity constraints are violated
  - Only valid data is saved
- **Isolation:** No interference between concurrent transactions
  - Transactions do not affect each other
- **Durability:** Changes are permanent
  - Written data will not be lost

Transaction Management State Diagram



Rollback

- Transaction is aborted:
  - Failed operation
  - Error
  - Failed integrity constraint
  - Timeout
  - SQL command: ROLLBACK

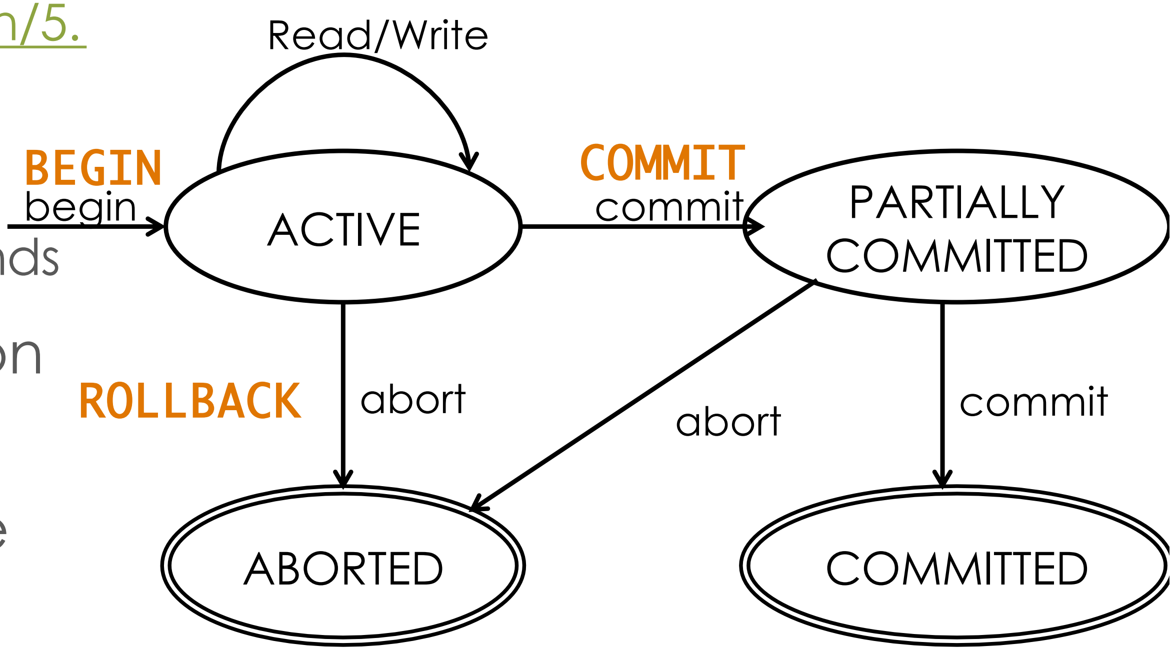


- All operations undone (A)
  - Appear like the transaction never took place
- Must not interfere with other transactions (I)

SQL: Transaction Control Language

Control statements

- http://dev.mysql.com/doc/refman/5.6/en/commit.html
- BEGIN: Start a transaction
  - Issue prior to running SQL commands
- COMMIT: Complete a transaction
  - Issue once all SQL commands run
- ROLLBACK: Undo the work since last commit
  - Issue to return to previous state



Schedule (Definitions)

- **Schedule:** Set of read and write operations from one or more transactions

- **Serial schedule:** Set of read and write operations from a one transaction take place before any operations of another transaction.
  i.e. All operations of T1, then all operations of T2
  - No interleaving of operations
  - Low throughput
  - No ACID violations

Transaction Processing

Aim to interleave operations from multiple transactions

- Maximise database throughput

- **Serializable schedule:** Interleaving of operations from more than one transaction so that it appears as if one transaction takes place before another.
  - High throughput
  - Potential for ACID violations
  - Equivalent serial schedule, but less efficient
  - For n transactions there are $n!$ possible serial schedules
    - Unfeasible to check them all

Schedules

Serial Schedule

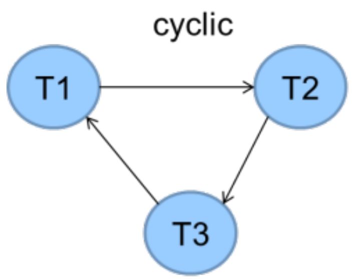
Transaction A	Transaction B
Get V1	
Add 5	
Put V1	
Get V2	
Add 5	
Put V2	
	Get V1
	Get V2

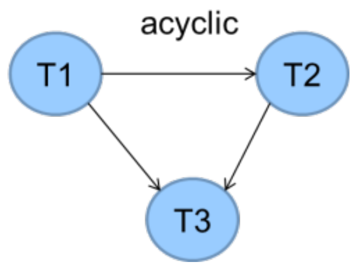
Serialisable Schedule

Transaction A	Transaction B
Get V1	
Add 5	
Put V1	
	Get V1
Get V2	
Add 5	
Put V2	
	Get V2

Serialization Graph

- **Serializabiltiy Theorem:**
  A schedule S is serializable if and only if its corresponding serializability graph SG(S) is acyclic





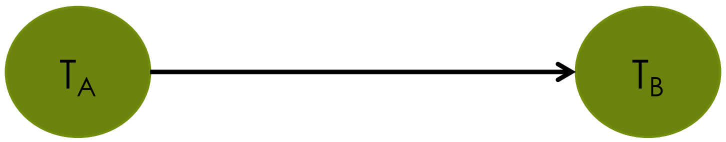
Algorithm to construct a serializability graph for a set of transactions $\mathcal{T}$

for $T \in \mathcal{T}$ **do**
  create node
end for
for each operation **do**
  if $(R \in T'$ follows $W \in T\ )$ **or**
  $(W \in T'$ follows $\{R|W\} \in T\ )$ **then**
    create edge from T to T'
  end if
end for

Drawing Serialization Graphs

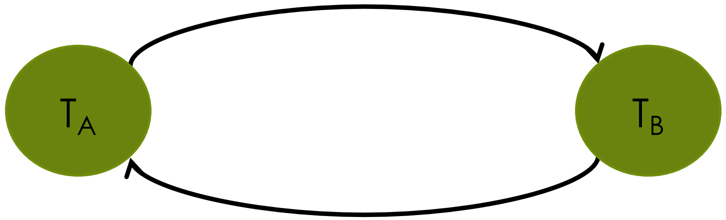
Serializable Schedule

Transaction A	Transaction B
Read V	
Write V	
	Read V
	Write V



Non-serializable Schedule

Transaction A	Transaction B
Read V	
	Read V
Write V	
	Write V

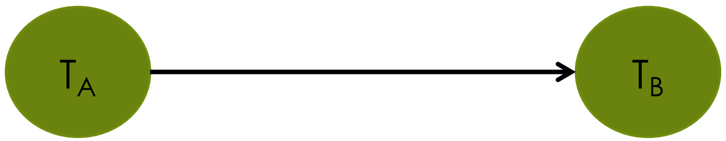


Drawi

Read X Follows Write Y then Y->X
Write X follows Read Y then Y->X
Write X Follows Write Y then Y->X

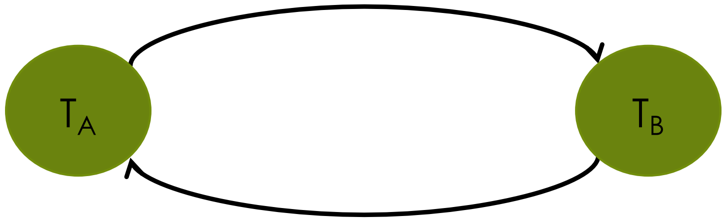
Serializable Schedule

Transaction A	Transaction B
Read V	
Write V	
	Read V
	Write V



Non-serializable Schedule

Transaction A	Transaction B
Read V	
	Read V
Write V	
	Write V



Homework

- Draw serialisation graph for problem 3

Transaction A	Transaction B	Value V1	Value V2
Get V1		5	15
Add 5		5	15
Put V1		10	15
	Get V1	10	15
	Get V2	10	15
Get V2		10	15
Add 5		10	15
Put V2		10	20

SQL Transaction

- Transaction to move money between accounts
- Don't want to "lose" money
  - Atomic: All should happen or none – money lost in ether
  - Consistency: IC to ensure account balance > 0
  - Isolation: Should not interfere with other transactions – money lost to ether
  - Durable: Once committed change not be lost – system crash

1.	START TRANSACTION;
2.	UPDATE Account SET balance = balance - 100 WHERE accountNo = 86036243;
3.	UPDATE Account SET balance = balance + 100 WHERE accountNo = 78612361;
4.	[COMMIT | ROLLBACK];

SQL Transaction with Variable

- Transaction all of Gareth Scarth's money to Zoe Kender
- Use variables to capture account number and balance

1.	START TRANSACTION;
2.	SELECT @a1:= accountNumber, @b1:=balance FROM Account WHERE firstnames = 'Gareth' AND lastname = 'Scarth'
3.	UPDATE Account SET balance = 0 WHERE accountNo = @a1;
4.	UPDATE Account SET balance = balance + @b1 WHERE firstnames = 'Zoe' AND lastname = 'Kender';
5.	[COMMIT | ROLLBACK];

SQL Isolation Levels for Transactions

- **Serializable** (highest): Appears that a transaction run entirely before or after others.
- **Read repeatable** (InnoDB default): Repeated read gets same data or same data with new inserts.
- **Read committed**: Repeated reads get new values from committed transactions.
- **Read uncommitted** ("dirty read"): Reads can see values from other concurrent transactions

Summary

- **Transactions:** group operations into unit of work
- Provide **ACID guarantees**
  - **Atomicity:** All or nothing
  - **Consistency:** Valid against schema and constraints
  - **Isolation:** Does not interfere with other transactions
  - **Durable:** Once committed actions are not lost
- Enable concurrent access
- Interleave operations into **Serialisable Schedule**

Concurrency Control

F28DM Database Management Systems

Matthew P Aylett
m.aylett@hw.ac.uk

Materials from: Alasdair G J Gray

Revision Quiz

- What is a transaction?
- What properties does the transaction manager guarantee?
- What is a serializable schedule?

Example: Money Transfer

Move £100 between two accounts

R/W 1.
```
UPDATE Account SET
balance = balance – 100
WHERE accountNo = 123;
```

R/W 2.
```
UPDATE Account SET
balance = balance + 100
WHERE accountNo = 124;
```

What would happen if only the first query executed?

Transaction Properties: ACID

- **Atomicity:** All actions complete or none
  - All or nothing
- **Consistency:** Database finishes in a consistent state, i.e. no integrity constraints are violated
  - Only valid data is saved
- **Isolation:** No interference between concurrent transactions
  - Transactions do not affect each other
- **Durability:** Changes are permanent
  - Written data will not be lost

Schedules

Serial Schedule

Transaction A	Transaction B
Get V1	
Add 5	
Put V1	
Get V2	
Add 5	
Put V2	
	Get V1
	Get V2

Serialisable Schedule

Transaction A	Transaction B
Get V1	
Add 5	
Put V1	
	Get V1
Get V2	
Add 5	
Put V2	
	Get V2

Concurrency Control

Running many transactions in parallel

- There are many strategies because how efficient it is depends on the way data is accessed and especially written
- That depends on what your database is for
- Different strategies have different advantages and disadvantages

Concurrency Control

Running many transactions in parallel

- **Single-Version:** maintains single copy of data
  - Locking (pessimistic):
    - Assumes conflicts
    - Locks prevent interactions
  - Timestamp (optimistic):
    - Deal with conflicts when they happen
    - Repeat work
- **Multi-Version:** maintains multiple copies of data

Locks

- Shared Lock (s-lock) aka read-lock:
  - Required for reading data
  - Many transactions can hold simultaneously
- Exclusive Lock (X-lock) aka write-lock:
  - Required for writing data
  - Only one transaction
  - No shared locks can exist for other transactions

Lock Granting Matrix

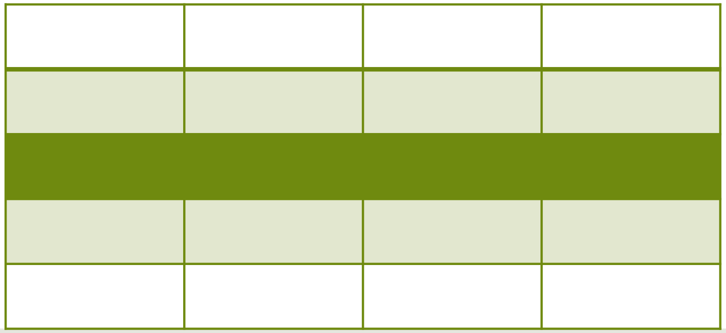
Lock Held	Shared requested	Exclusive requested
Shared	Granted	Rejected
Exclusive	Rejected	Rejected

Lock Granularity

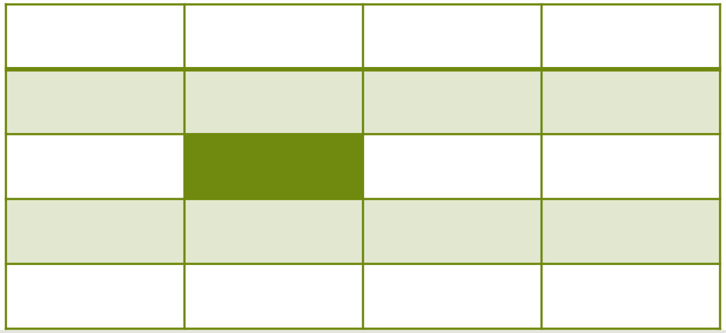
Table

Disk Block

Row



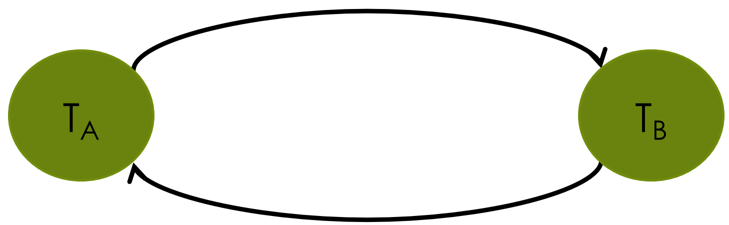
Cell



Blocking & Deadlock

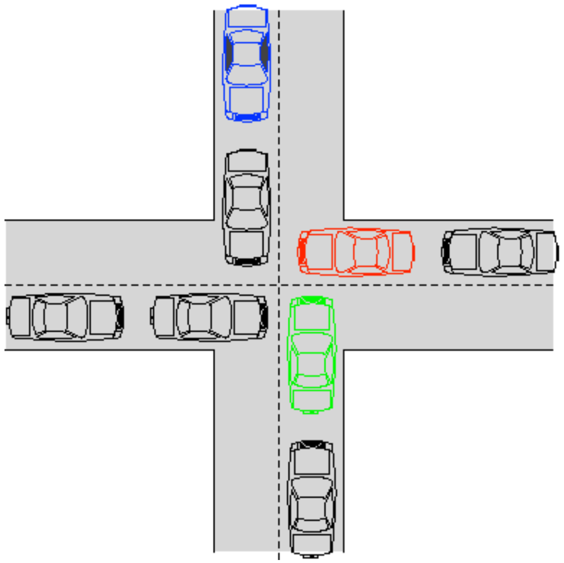
- **Blocking:** waiting to acquire lock held by another transaction
- **Deadlock:** Cycle of transactions waiting for locks

T_A	T_B
Get V1	
Put V1	
	Get V2
	Get V1
Get V2	
Put V2	



Handling Deadlock

- **Time-out:** Abort blocked transaction
  - Simple to implement, pick random transaction
  - Aggressive: can undo more than required
- **Detection:** Identify cycle
  - Draw wait-for graph (serialisation graph)
    - Requires processing
  - Abort selected transactions
    - Don't undo unnecessarily
- **Two phase locking** protocol

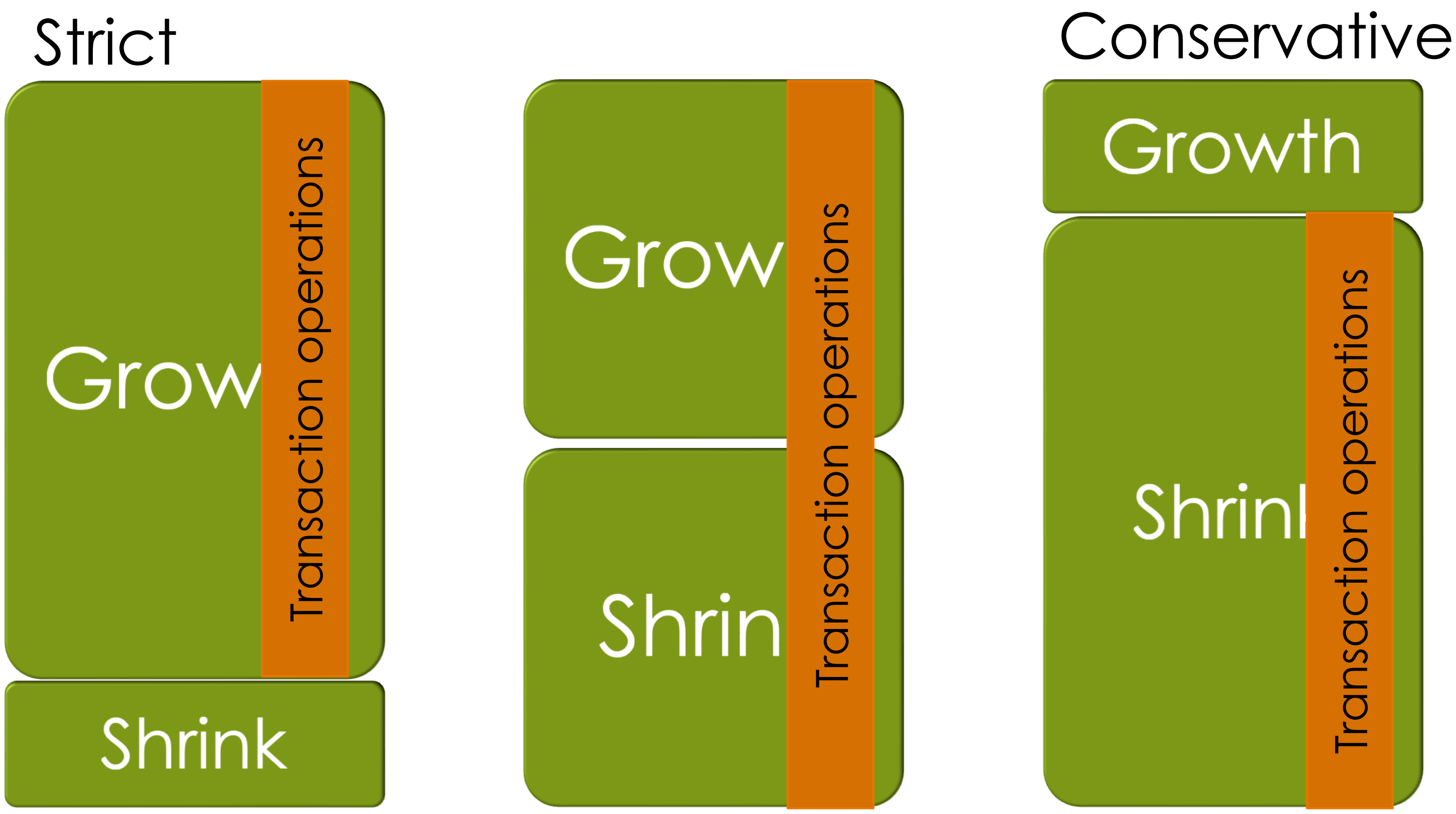


http://pages.cs.wisc.edu/~bart/537/lecturenotes/figures/s12.crash.gif

Two-Phase Locking (2PL)

- **Growth phase:** transaction acquires locks as needed
- **Shrink phase:** transaction releases locks
  - Can no longer acquire locks
- **2PL:** Locks acquired as needed, locks released when no longer required and no more locks will be needed
  - Problems occur if a lock released and transaction aborted

Two-Phase Locking Summary



2PL: Example

T_A	T_B	***2PL***
BEGIN		
Get V1		T_A: Acquire S-lock on V1
Put V1		T_A: Acquire X-lock on V1 Lock not released as require more locks
	BEGIN	
	~~Get V1~~	T_B: blocked acquiring S-lock on V1, transaction paused
Get V2		T_A: Acquire S-lock on V2
Put V2		T_A: Acquire X-lock onV2.
Get V1		T_A: Lock on V2 released, already have lock on V1
COMMIT		T_A: Release lock on V1
	GET V1	T_B: Unpaused. Acquire S-lock on V1
	…	

2PL: Read Uncommitted (dirty read)

Grow

Shrin

Transaction operations

T_A	T_B	*2PL*
BEGIN		
Get V1		T_A: Acquire S-lock on V1
Put V1		T_A: Acquire X-lock on V1 Lock not released as require more locks
	BEGIN	
	~~Get V1~~	T_B: blocked acquiring S-lock on V1, transaction paused
Get V2		T_A: Acquire S-lock on V2
Put V2		T_A: Acquire X-lock onV2, release lock on V1
	GET V1	T_B: Unpaused. Acquire S-lock on V1
ABORT		T_A: Work undone, release lock on V2
	…	

T_B has read the updated value of T_A

Strict Two-Phase Locking

- Locks acquired as needed
- All locks kept until the end of the transaction
  - Released in one go
  - Eliminates ‘dirty read’ problem of 2PL
  - Reduces throughput

Strict Two-Phase Locking

T_A	T_B	***2PL***
BEGIN		
Get V1		T_A: Acquire S-lock on V1
Put V1		T_A: Acquire X-lock on V1
	BEGIN	
	~~Get V1~~	T_B: blocked acquiring S-lock on V1, transaction paused
Get V2		T_A: Acquire S-lock on V2
Put V2		T_A: Acquire X-lock onV2.
Get V1		T_A: Already have lock on V1
COMMIT		T_A: Release locks on V1 & V2
	GET V1	T_B: Unpaused. Acquire S-lock on V1
	…	

Strict 2PL: No dirty read

T_A	T_B	***2PL***
BEGIN		
Get V1		T_A: Acquire S-lock on V1
Put V1		T_A: Acquire X-lock on V1 Lock not released as require more locks
	BEGIN	
	~~Get V1~~	T_B: blocked acquiring S-lock on V1, transaction paused
Get V2		T_A: Acquire S-lock on V2
Put V2		T_A: Acquire X-lock onV2, ***lock on V1 is not released***
ABORT		T_A: Work undone, release locks on V1 and V2
	GET V1	T_B: Unpaused. Acquire S-lock on V1
	…	

T_B cannot proceed until T_A completes

Conservative Two-Phase Locking

- All locks acquired at start
- Locks released once no longer needed
  - Locks released before end of transaction
  - Susceptible to dirty read problem
  - Prevents deadlocks
  - Better throughput than strict

Example: Conservative 2PL

T_A	T_B	***2PL***
BEGIN		T_A: Acquire X-locks on V1 & V2
Get V1		
Put V1		
	BEGIN	T_B: blocked acquiring X-lock on V1, transaction paused
Get V2		
Put V2		
Get V1		T_A: Release lock on V2
	GET V1	T_B: Unpaused. Having acquired S-lock on V1
COMMIT		T_A: Release lock V2
	…	

Conservative 2PL: Dirty read

T_A	**T_B**	***2PL***
BEGIN		T_A: Acquire X-locks on V1 & V2
Get V1		
Put V1		
	BEGIN	T_B: blocked acquiring X-lock on V1, transaction paused
Get V2		
Put V2		Release lock on V1
	GET V1	T_B: Unpaused. Acquires requited locks *(T_B does not use V2)*
ABORT		T_A: Work undone, release lock on V2
	…	

T_B has read the updated value of T_A

Homework: MySQL Locks

Perform operations in two different terminal windows both connected to MySQL. Terminal 2 cannot proceed until Terminal 1 commits.

Operation	Terminal 1	Terminal 2
1	START TRANSACTION;	
2	UPDATE account SET balance = 100 WHERE accountNo = 23525;	
3		UPDATE account SET balance = balance + 10 WHERE accountNo = 23525;
4	UPDATE account SET balance = 100 WHERE accountNo = 23526;	
5	COMMIT;	

Timestamp Protocol

- Each transaction assigned timestamp t_T
- Each data item has
  - Last read timestamp t_r
  - Last write timestamp t_w
- Operation permitted iff

  $$t_{r|w} < t_T$$

- Otherwise
  - Abort transaction t_T
  - Restart transaction

Exercise: Draw sequence of events for the three problems

http://upload.wikimedia.org/wikipedia/commons/8/89/Zeitstempel_01.jpg

Timestamp Protocol

Transaction 28	Transaction 32
Read V	
	Read V
Write V	
	Write V

Vs timestamp

Read	Write
26	26

Multi-version Concurrency (MVCC)

- Multiple simultaneous transactions
- Each transaction sees isolated snapshot
- Changes only seen across transactions on commit

MVCC in Action

xmin	xmax	name	notes
100	0	Alice	Great at programming
101	0	Bob	Always talking to Alice
102	0	Eve	Listens to everyone's conversations

- TXID 103: update Bob

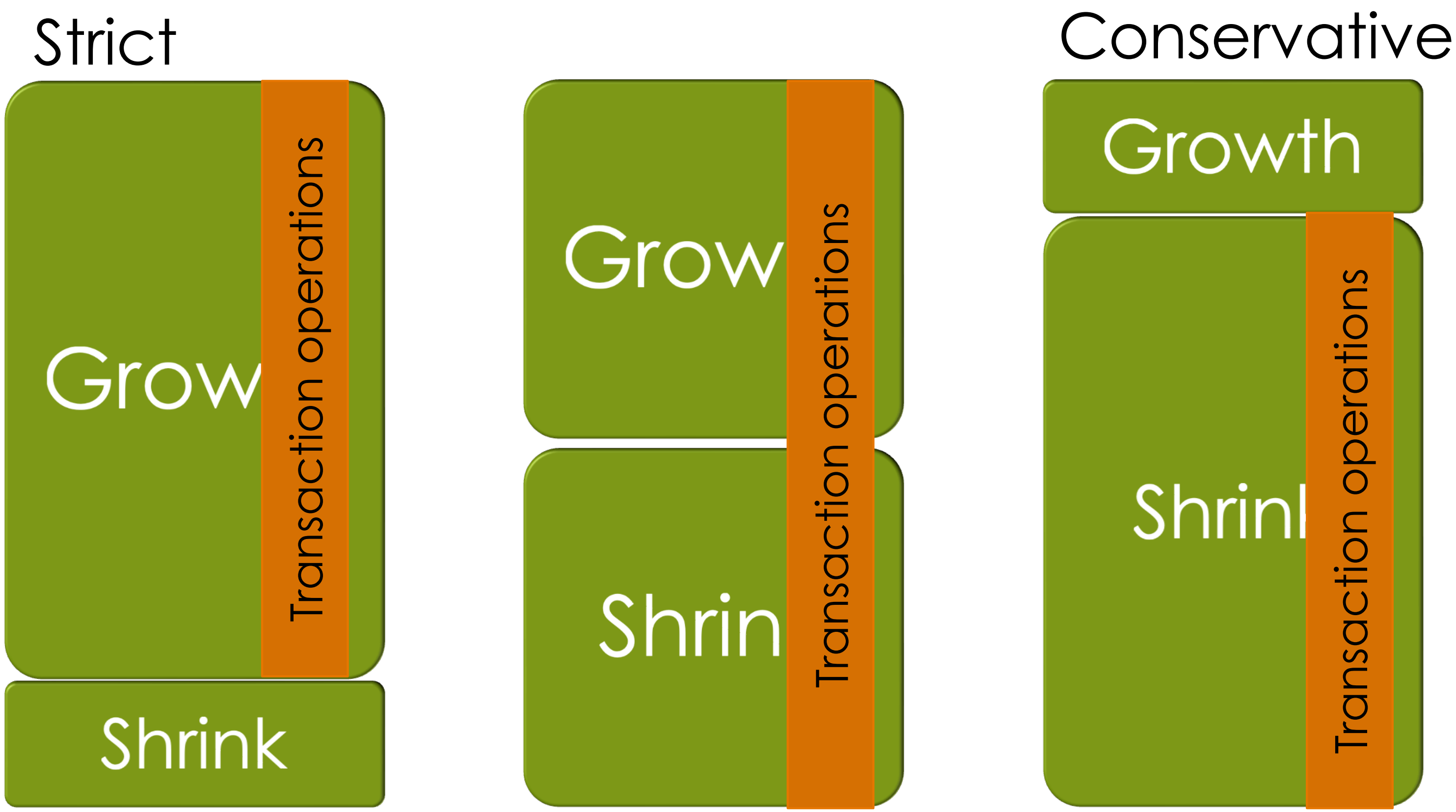
xmin	xmax	name	notes
100	0	Alice	Great at programming
101	**103**	Bob	Always talking to Alice
102	0	Eve	Listens to everyone's conversations
103	0	Bob	Working very hard

- TXID102: long running read over entire table
  - Reads value 'Always talking to Alice'
  - Won't see value about Eve (only see values strictly less than TID)

Summary

- **Concurrency control:** enables multiple users to interact with the database
- **2 phase commit**: uses **locks** (s-lock/x-lock)
  - Classical: dirty read problem
  - Strict: reduces throughput
  - Conservative: dirty read problem
- **Timestamp concurrency control:** can only operate on old values
- **Multi-version concurrency control:** retains multiple versions of the data and when they are valid

Two-Phase Locking Summary



References

Connolly, T., & Begg, C. (2005). *Database Systems: A Practical Approach to Design, Implementation, and Management* (4th ed.). Addison Wesley. Chapter 20

Ward, P. (2008). *Database Management Systems* (2nd ed.). Middlesex University Press. Chapter 9