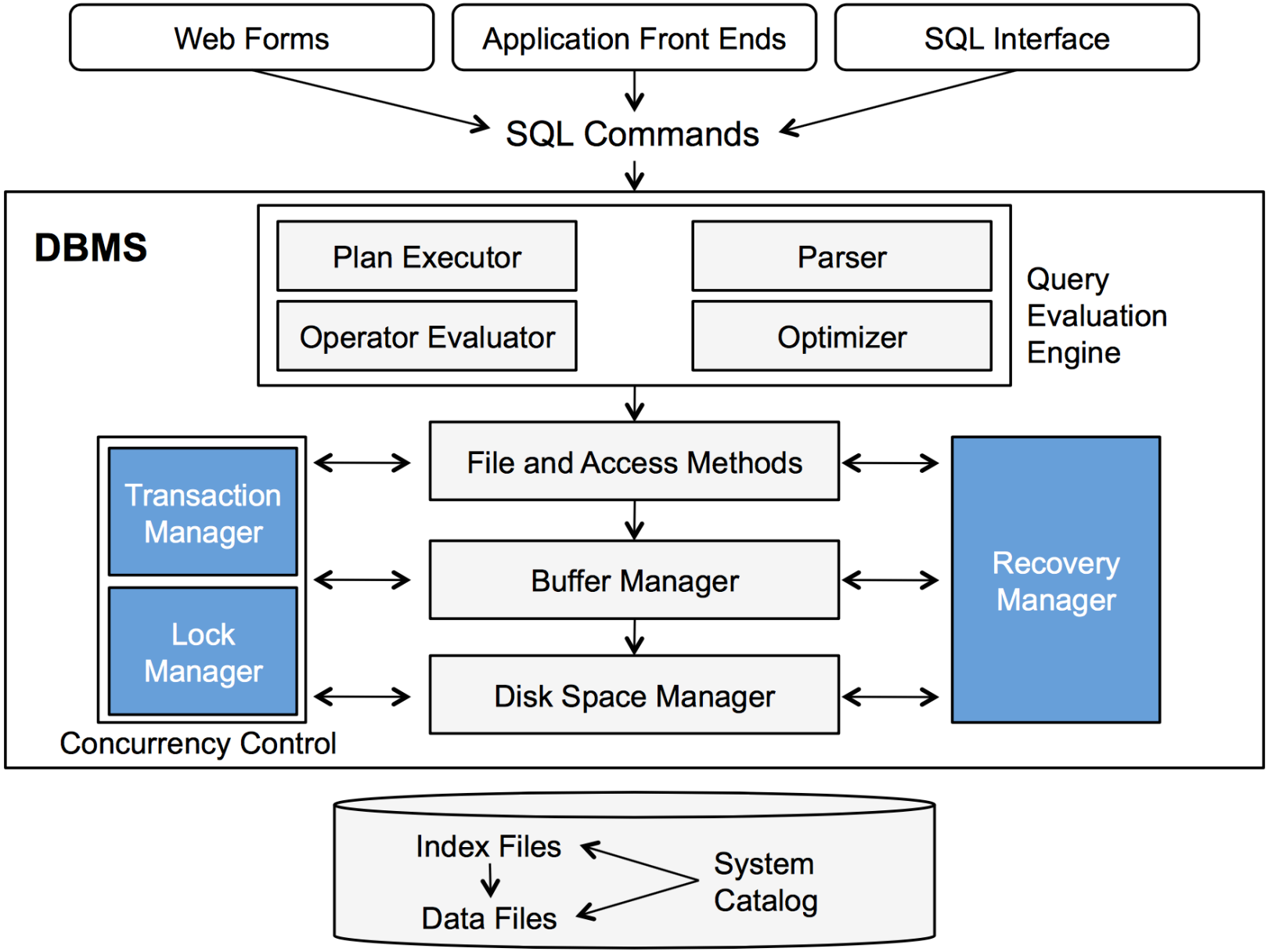
Database Management System

Lecture 9

Transaction, Concurrency Control

* Some materials adapted from R. Ramakrishnan, J. Gehrke and Shawn Bowers

Basic Database Architecture



Concurrency Control and Recovery

- Transactions
  - A way to define a single “all or nothing” set of SQL actions
  - Based on a set of properties (the “ACID” properties)
  - Enable concurrency
  - The basis for crash recovery

Transactions

- A “transaction” is a set of SQL statements (that modify a database) chosen by a user

Transfer $100 from one account to another (*MySQL*):

```
START TRANSACTION;
   SELECT @A1 := balance
      FROM Account
      WHERE acctno=500;
   UPDATE Account
      SET balance := @A1+100
      WHERE acctno=500;
   SELECT @A2 := balance
      FROM Account
     WHERE acctno=501;
   UPDATE Account
      SET balance := @A2-100
      WHERE acctno=501;
COMMIT;
```

BEGIN TRANSACTION
READ balance from account 500
ADD $100 to balance of account 500
WRITE new balance to account 500
READ balance in account 501
VERIFY balance to see if it contains at least $100
ABORT if balance is less than $100
SUBTRACT $100 from balance of account 501
WRITE new balance to account 501
COMMIT TRANSACTION

Transactions

- User (application developer) must
  - Begin transaction
  - Read, write, and modify statements intermixed with other programming language statements (e.g., verify)
- Plus either
  - Commit to indicate successful completion or
  - Abort to indicate that the the transaction should be “rolled back” ... i.e., erase previous steps of transaction
- *To ensure database stays in a consistent/correct state*
  - The DBMS and programmer must guarantee 4 properties of transactions
  - These are called the “*ACID*” properties

ACID Properties

- Atomicity
- Consistency
- Isolation
- Durability

ACID Properties

Atomicity

A transaction happens in its entirety or not at all

- What if the OS crashed half-way through the transaction ... after $100 was removed from the first account?
- The *recovery manager* of the DBMS must assure that the $100 is deposited back to the first account
- Often called “roll back”

ACID Properties

Consistency

If the DB starts in a consistent state, the transaction will transform it into a consistent state

- The notion of "consistency" is specific to the application constraints (defined by the user)
- Thus the *programmer* must ensure transactions are consistent
  - The DBMS ensures the transaction is atomic
- E.g., what if the transaction only deposited $100?
  - Probably not consistent according to our example application

ACID Properties

Isolation

Each transaction is "isolated" from other transactions …
DB state is as if each transaction executed by itself

- What if an another transaction computed the total bank balance after $100 was removed from the first account?
- The *concurrency control* subsystem must
  - ensure that all transactions run in isolation (i.e., don't mess up other transactions)
  - unless the programmer chooses a less strict level of isolation – similar to concurrency control in operating systems

ACID Properties

Durability

If a transaction commits, its changes to the DB state persist (changes are permanent)

- What if after the commit the OS crashed before the deposit was written to disk?
- The *recovery manager* must assure that the deposit was at least logged (e.g., to make the DB consistent)

Concurrency

- Why is concurrency important?
  - Better *utilization of resources*
  - E.g., while one user/transaction is reading the disk, another can be using the CPU or reading another disk
  - Results in better throughput and response time
  - Many applications require it (for performance)

Concurrency

- We'll look at concurrency in terms of isolation of transactions
  - Isolation is a problem when multiple transactions are running, using the same data, and operations are interleaved
  - Isolation ensured by the concurrency control subsystem
  - This should be familiar if you've taken the OS class ...

Serial Schedules

- Consider these transactions:
- *Deposit to A and withdraw from B*

  T1: BEGIN A = A + 100; B = B – 100; END
- *Compute the balance of A and B*

  T2: BEGIN C = A + B; END
- *Apply interest to A and B*

  T3: BEGIN A = 1.06 * A; B = 1.06 * A; END
  - A schedule is an interleaving of the actions of the transactions so that each transaction's *order is preserved*
  - A schedule of transactions is serial if its transactions occur consecutively, one after another

Serial Schedules

- Which of these is a schedule? Which is serial?

S1 Serial Schedule!

T1	T3
A=A+100 B=B-100	 A=1.06*A B=1.06*B

S2 Non-serial Schedule!

T1	T3
A=A+100 B=B-100	 A=1.06*A B=1.06*B

S3 Non-serial Schedule!

T1	T2
A=A+100 B=B-100	 C=A+B

S4 Not a Schedule!

T1	T3
B=B-100 A=A+100	 C=A+B

Allowable Concurrency

- What is wrong with S3?
  - It does not give the same result as any *serial schedule*
- The DBMS should
  - Allow serial schedules like S1
  - Forbid interleaved schedules like S3
- But what about S2?
  - Note that it gives the *same result* as S1!
  - The DBMS should also allow this schedule!
- If the DBMS only allows serial schedules, then it becomes a batch system (where is the concurrency?)

Serializable Schedules

- A schedule is "*serializable*" if its effect on the DB is the same as the effect on some serial schedule
  - Serial schedules are always serializable
  - S2 is serializable, but S3 is not
- Serializability is the same as the isolation condition
- The goal of the concurrency control subsystem is to ensure serializability

Schedules as Reads and Writes

- An expression A = 1.06 * A means – Read A from disk
  - Set A equal to 1.06 *A
  - Write A to disk
- Only the read and write to disk matter to the DBMS!
  - We'll use the notation ...
  - R(A) for Read A
  - W(A) for Write A

Schedules as Reads and Writes

- These are equal (from DBMS/concurrency perspective)

S2

T1	T3
A=A+100	
	A=1.06*A
B=B-100	
	B=1.06*B

S2

T1	T3
R(A), W(A)	
	R(A), W(A)
R(B), W(B)	
	R(B), W(B)

Conflict Serializability

- S2 has a special structure that makes it possible to show that it is serializable ...
- Two actions are "*nonconflicting*" if they are in different transactions and either they
  - Access different data times (resources)
  - Or both are reads
- If nonconflicting actions are commuted then the new schedule gives the same result

Conflict Serializability

- wo schedules are "*conflict equivalent*" if
  - One can be transformed into the other by commuting (swapping) nonconflicting actions
- A schedule is "*conflict serializable*" if it is conflict equivalent to at least one serial schedule

Thus, every conflict serializable schedule is serializable!
(... but not necessarily the other way around)

Serializability

- Nonconflicting shown in Red

S2

T1	T3
R(A), W(A)	
	R(A), W(A)
R(B), W(B)	
	R(B), W(B)

S2

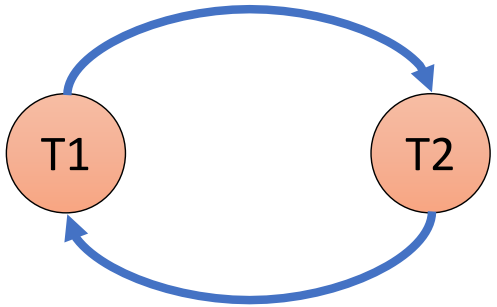
T1	T3
R(A), W(A)	
R(B), W(B)	
	R(A), W(A)
	R(B), W(B)

Ais used in T3, B in T1

This is a serial schedule!
This means S2 is "conflict serializable"
(... and thus is serializable)

Precedence Graphs

- Verifying conflict serializability is tedious
- There is an easier way!



- Precedence graphs
  - One node per transaction
  - Edge from T_i to T_j if an action in T_i occurs before an action in T_j and the actions conflict
- *Theorem*
  - A schedule is *conflict serializable* if an only if its precedence graph is acyclic

Exercise

- With a partner, draw the precedence graphs for the previous schedules (S1-S3)
  - You'll first have to convert them to R's and W's

S1

T1	T3
A=A+100 B=B-100	 A=1.06*A B=1.06*B

S3

T1	T2
A=A+100 B=B-100	 C=A+B

S4

T1	T3
B=B-100 A=A+100	 C=A+B

Exercise

- With a partner, draw the precedence graphs for the previous schedules (S1-S3)

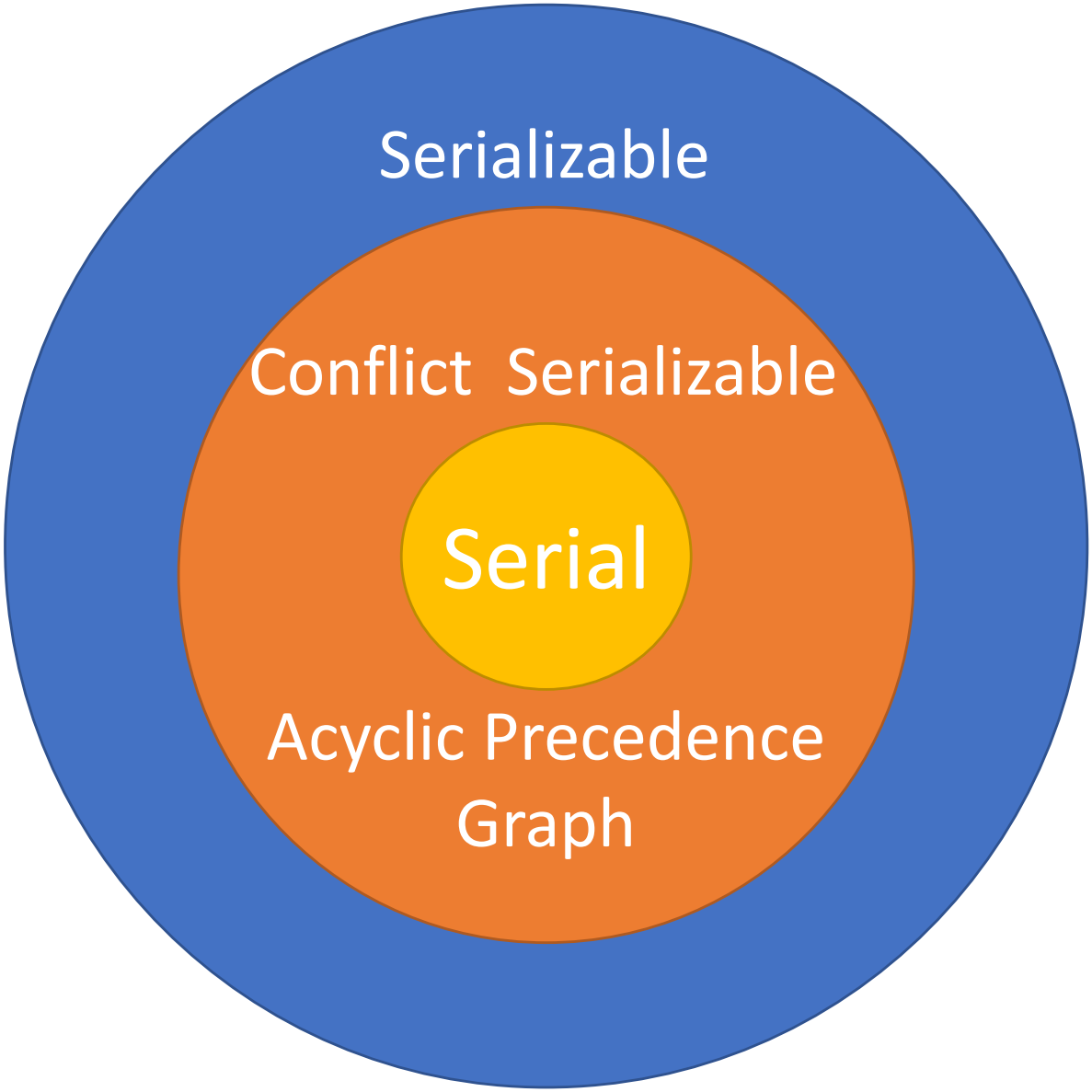
S5

T1	T2	T3
R(A)		
	W(A)	
R(B)		
		W(A)
W(B)		
		R(B)

S6

T1	T2	T3
R(A)		
	W(A)	
W(B)		
		W(A)

Relationships



Serializability in Practice

- Precedence graphs give us a simple way to prove that a schedule is (conflict) serializable
  - But they do not work for all schedules
  - In theory, this could be used by a DBMS to check for serializability ...
- *In practice*
  - A DBMS is not presented with schedules ... it only sees a stream of transactions
  - Instead, *locking* is used to achieve “isolation”

Locking

- Transactions *must obtain a lock* before reading or updating (writing) data
- Two kinds of locks:
  - Shared (S) locks
  - Exclusive (X) locks
- To read a record you MUST get an S lock
- To write (modify/delete) a record you MUST get an X lock
- Lock information is maintained by a "*lock manager*"

How Locks Work

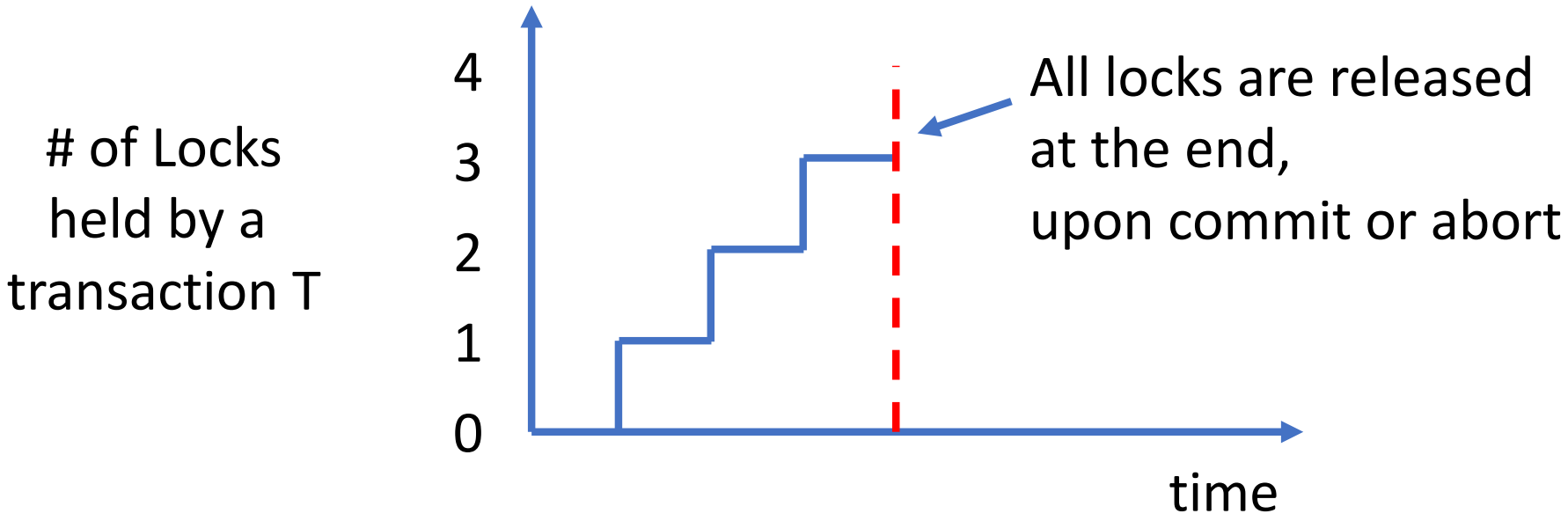
- If an object has an S (shared) lock
  - new transactions can obtain S (shared) locks
  - but *not* X (exclusive) locks
- If an object has an X lock
  - no other transaction can obtain *any* lock on that object
- If a transaction cannot obtain a lock
  - It is *blocked* (i.e., waits in a queue)

Strict Two Phase Locking Protocol (Strict 2PL)

- In Strict 2PL
  - Transaction T obtains (S and X) locks gradually, as needed
  - T holds all locks until end of transaction (commit/abort)

This guarantees serializability!

- Still permits interleaved schedules
- But can lead to deadlock



Strict Two Phase Locking Protocol (Strict 2PL)

- Examples

S6

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

S6 (S2PL)

T1	T2	T3
X(A)		
R(A)		
W(A)		
Commit		
	X(A)	
	W(A)	
	Commit	
		X(A)
		W(A)
		Commit

S7 (S2PL)

T1	T2
	X(A)
	W(A)
X(B)	
X(B)	
	X(C)
	W(C)
	Commit
X(C)	
W(C)	
Commit	

Strict Two Phase Locking Protocol (Strict 2PL)

- Yet another example
  - T1:W(A),W(B)
  - T2:W(B),W(A)

S7 (S2PL)

T1	T2
X(A)	
W(A)	
	X(B)
	W(B)
Waiting for X(B)	Waiting for X(A)
…	…

DEADLOCK!!

Deadlocks in DBMS

- What is a Deadlock?
  - A cycle of transactions, e.g.,T1 ... Tn where each Ti is waiting for Ti-1 to release a lock!
  - Causes these transactions to sleep/wait forever
- Deadlocks can occur in strict 2PL
- Solutions
  - Always access resources in the same order (impractical)
  - The DBMS will typically detect deadlocks
  - ... and then abort the transaction (it thinks) has used the least resources