CS 61: Database Systems

Transactions/Concurrency

Adapted from Silberschatz, Korth, and Sundarshan unless otherwise noted

Practice: Normalization

Soccer player database

PlayerID	Name	Team	TeamPhone	Position1	Position2	Position3
1	Pessi	Argentina	54-11-1000-1000	Striker	Forward	
2	Ricardo	Portugal	351-2-7777-7777	Right Midfield	Defending Midfielder	
3	Neumann	Brazil	55-21-4040-2020	Forward	Left Fullback	Right Fullback
4	Baily	Wales	44-29-1876-1876	Defending Midfielder	Striker	
5	Marioso	Argentina	54-11-1000-1000	Sweeper	Defending Midfielder	Striker
6	Pare	Brazil	55-21-4040-2020	Goalkeeper		

Business rules

- Each player uniquely identified by PlayerID
- Each player plays for one team and can play zero or more positions
- Each team has many players and one phone number
- Assume players primary position listed first (e.g., Pessi primarily Striker)

Normalize this table

- Download soccer_unnormalized.mwb from course web page to start
- Create necessary tables and confirm at least 3NF

Based on Prof Charles Palmer lecture notes

Agenda

1. Database inconsistencies

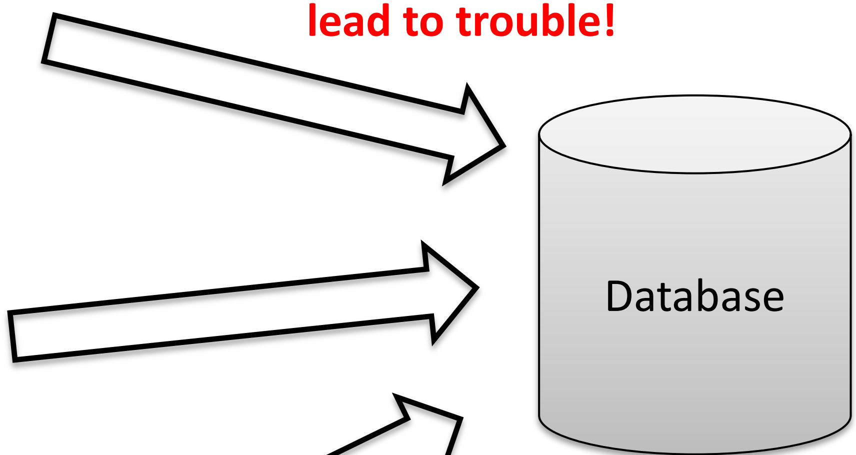
2. ACID transactions

3. Concurrency/Isolation

Goal: quickly serve many users at the same time, but data must stay consistent!

Avoid handling user requests sequentially – too slow!

Concurrent processing can lead to trouble!



Problem:
Must ensure data stays consistent with concurrent transactions

Assume database starts in consistent state

- All integrity constraints met
- All business rules followed

Multiple CPUs in database server could serve multiple requests at the same time

Result: increased throughput

Attribute-level inconsistencies can occur when transactions update the same data

Attribute-level inconsistency

T1

```
UPDATE CheckingAccount
SET Balance = Balance + 100
WHERE AccountNumber = 123456;
```

T2

```
UPDATE CheckingAccount
SET Balance = Balance + 150
WHERE AccountNumber = 123456;
```

Two clients initiate simultaneous update of checking account balance with transactions T1 and T2

- **Each transaction involves read, increment, and write of same data**
- **Assume Balance starts at $100**

T1	T2
Read Balance ($100)	
Increment Balance by $100 ($200)	
Write Balance ($200) Commit	
	Read Balance ($200)
	Increment Balance by $150 ($350)
	Write Balance ($350) Commit

If T1 and T2 complete as expected, afterward new Balance is $350

Based on Prof Palmer lecture notes

Attribute-level inconsistencies can occur when transactions update the same data

Attribute-level inconsistency

T1

```sql
UPDATE CheckingAccount
SET Balance = Balance + 100
WHERE AccountNumber = 123456;
```

T2

```sql
UPDATE CheckingAccount
SET Balance = Balance + 150
WHERE AccountNumber = 123456;
```

Two clients initiate simultaneous update of checking account balance with transactions T1 and T2

- **Each transaction involves read, increment, and write of same data**
- **Assume Balance starts at $100**

T1	T2
	Read Balance ($100)
	Increment Balance by $150 ($250)
	Write Balance ($250) Commit
Read Balance ($250)	
Increment Balance by $100 ($350)	
Write Balance ($350) Commit	

If T2 completes before T1, Balance afterward is still as expected, $350

Based on Prof Palmer lecture notes

Attribute-level inconsistencies can occur when transactions update the same data

Attribute-level inconsistency

T1

```
UPDATE CheckingAccount
SET Balance = Balance + 100
WHERE AccountNumber = 123456;
```

T2

```
UPDATE CheckingAccount
SET Balance = Balance + 150
WHERE AccountNumber = 123456;
```

Two clients initiate simultaneous update of checking account balance with transactions T1 and T2

- **Each transaction involves read, increment, and write of same data**
- **Assume Balance starts at $100**

T1	T2
Read Balance ($100)	
	Read Balance ($100)
Increment Balance by $100 ($200)	
Write Balance ($200) Commit	
	Increment Balance by $150 ($250)
	Write Balance ($250) Commit

If T1 is interrupted and T2 reads Balance before T1 finishes incrementing and writing, $100 is lost!

Based on Prof Palmer lecture notes

Attribute-level inconsistencies can occur when transactions update the same data

Attribute-level inconsistency

T1

```
UPDATE CheckingAccount
SET Balance = Balance + 100
WHERE AccountNumber = 123456;
```

T2

```
UPDATE CheckingAccount
SET Balance = Balance + 150
WHERE AccountNumber = 123456;
```

Two clients initiate simultaneous update of checking account balance with transactions T1 and T2

- **Each transaction involves read, increment, and write of same data**
- **Assume Balance starts at $100**

T1	T2
Read Balance ($100)	
	Read Balance ($100)
Increment Balance by $100 ($200)	
	Increment Balance by $150 ($250)
	Write Balance ($250) Commit
Write Balance ($200) Commit	

OR $150 is lost!

This condition is called the *lost update problem*

Based on Prof Palmer lecture notes

Attribute-level inconsistencies can occur when transactions update the same data

Attribute-level inconsistency

T1

```sql
UPDATE CheckingAccount
SET Balance = Balance + 100
WHERE AccountNumber = 123456;
```

T2

```sql
UPDATE CheckingAccount
SET Balance = Balance + 150
WHERE AccountNumber = 123456;
```

Two clients initiate simultaneous update of checking account balance with transactions T1 and T2

- **Each transaction involves read, increment, and write of same data**
- **Assume Balance starts at $100**

T1	T2
Read Balance ($100)	
Increment Balance by $100 ($200)	
Write Balance ($200)	
	Read Balance ($200)
	Increment Balance by $150 ($350)
	Write Balance ($350)
	Commit
Rollback	

T1 could rollback, leading T2 with an erroneous value

Another variant is the *uncommitted data problem*

Based on Prof Palmer lecture notes

Attribute-level inconsistencies can occur when transactions update the same data

Attribute-level inconsistency

T1

```
UPDATE CheckingAccount
SET Balance = Balance + 100
WHERE AccountNumber = 123456;
```

T2

```
UPDATE CheckingAccount
SET Balance = Balance + 150
WHERE AccountNumber = 123456;
```

- **Database will often be temporarily in an inconsistent state**
- **Transactions can make the operations atomic so that they can't be interrupted (or are rolled back if they are interrupted)**

Two clients initiate simultaneous update of checking account balance with transactions T1 and T2

- **Each transaction involves read, increment, and write of same data**
- **Assume Balance starts at $100**

T1	T2
Read Balance ($100)	
Increment Balance by $100 ($200)	
Write Balance ($200)	
	Read Balance ($200)
	Increment Balance by $150 ($350)
	Write Balance ($350)
	Commit
Rollback	

Based on Prof Palmer lecture notes

Relation-level inconsistencies can occur when results depend on transaction order

Relation-level inconsistency

Apply holds student applications for college

- **Simple admission criteria based only on grade**
- **But maybe large school students get a GPA bump**

T1

```
UPDATE Apply
SET Decision = 'Y'
Where StudentID IN (SELECT StudentID from Student WHERE GPA > 3.9);
```

T2

```
UPDATE Student
SET GPA = 1.1*GPA
WHERE HighSchoolSize > 2500;
```

Some rows in the Apply table are affected by order in which these transactions are run

- **If T1 runs before T2, some students won't be accepted that would have been accepted if T2 ran first**
- **Here updates are applied to different relations, but could give different results**
- **T1 operates on two tables, T2 operates on one of those two**

Based on https://lagunita.stanford.edu/assets/courseware/v1/b91aa86921e55e62d426677a4a36e85e/c4x/DB/Indexes/asset/TransactionsProperties.pdf

Multi-statement inconsistencies can occur when results depend on transaction order

Multi-statement inconsistency

Results from SELECT statements depend on whether they run before, after, or between INSERT/DELETE statements

```
INSERT INTO Archive
    SELECT * FROM Apply WHERE Decision = 'N';
DELETE FROM Apply WHERE Decision = 'N';

SELECT COUNT(*) from Apply;
SELECT COUNT(*) from Archive;
```

If SELECT runs here

- DELETE has not yet run
- Total count will be incorrect because ‘N’ decision not yet deleted from Apply

Based on https://lagunita.stanford.edu/assets/courseware/v1/b91aa86921e55e62d426677a4a36e85e/c4x/DB/Indexes/asset/TransactionsProperties.pdf

Multi-statement inconsistencies can occur when results depend on transaction order

Multi-statement inconsistency

```
INSERT INTO Archive
     SELECT * FROM Apply WHERE Decision = 'N';
DELETE FROM Apply WHERE Decision = 'N';

SELECT COUNT(*) from Apply;
SELECT COUNT(*) from Archive;
```

So must we force all transactions to run serially (one after the other)?

- **Defeats the purpose of large databases serving many simultaneous users**
- **Want concurrency so we have highest possible performance**

What about system failures?

- **Power goes out during transaction**
- **Disgruntled employee types:** `rm –rf /`

Transactions to the rescue!

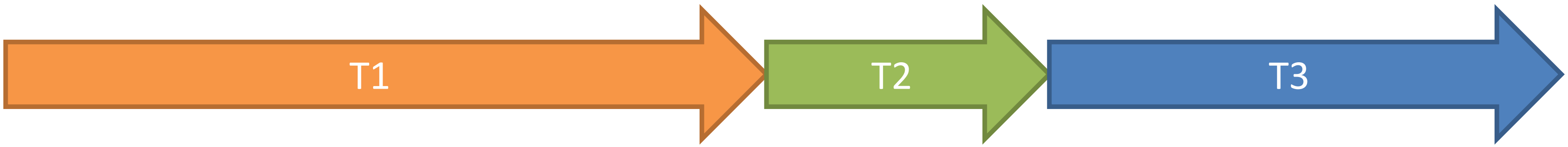
Based on Prof Palmer lecture notes

Agenda

1. Database inconsistencies
2. ACID transactions
3. Concurrency/Isolation

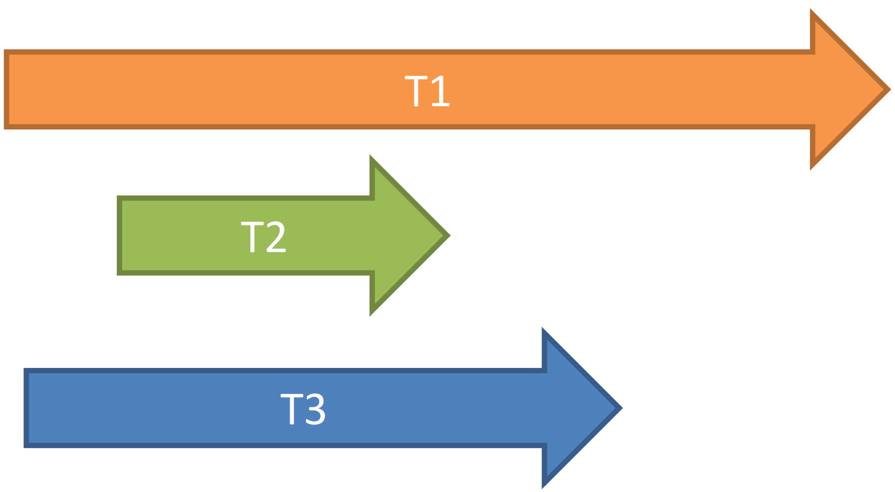
Goal: want transaction to run fast but not allow inconsistencies

Serial schedule (run consecutively; first come, first served)



Serialized schedule **interleaves execution and gives same result as if transaction ran serially**

Interleaved schedule (serialized)



Schedule is clearly serializable if:

- **Transactions operate on different data**
- **Only read operations**

- Consistency assumptions
  1. Database starts in consistent state
  2. Each transaction leaves database in consistent state when complete
  3. Serial execution of transactions preserves consistency
- As we have seen, problems can arise if we allow simultaneous (concurrent) transaction execution
- But performance is low if transactions must run serially
- Some transactions do not interfere with each other (they can be serialized)

To allow concurrent transactions we want ACID properties

ACID: Atomic, Consistent, Isolated, Durable

Atomic

- Transaction treated as indivisible unit of work
- All commands in transaction complete successful or transaction is aborted
- Locks commonly used to ensure only one transaction accesses data at a time
- Transaction log allows rollback if transaction aborts

Consistent

- All data integrity constraints satisfied
- Transaction must take database from one consistent state to another
- If any integrity constraint is violated, transaction is aborted

Isolated

- Data used during a transaction cannot be access by another transaction until the first transaction completes
- As if each transaction runs by itself, gives same result as serial execution

Durable

- Once changes are committed, they cannot be undone

A transaction is a logical unit of work that must be entirely completed or aborted

Transactions make multiple commands Atomic, Consistent and Durable

```sql
-- extended example from MySQL Docs section 15.7.2.2
DROP TABLE IF EXISTS Customers;
CREATE TABLE Customers (a INT, b CHAR (20), INDEX (a));

-- Do two transactions and commit
START TRANSACTION;
INSERT INTO Customers VALUES (10, 'Heikki');
INSERT INTO Customers VALUES (11, 'Elvis');
COMMIT; -- transactions saved
SELECT * FROM Customers;


START TRANSACTION;
INSERT INTO Customers VALUES (15, 'John');
INSERT INTO Customers VALUES (20, 'Paul');
DELETE FROM Customers WHERE b = 'Heikki';

-- Now we undo those last 2 inserts and the delete.
ROLLBACK;
SELECT * FROM Customers;
```

Transaction starts and data inserted

Power failure here would rollback changes at restart (not committed)

Data committed

Second transaction inserts two rows and deletes one

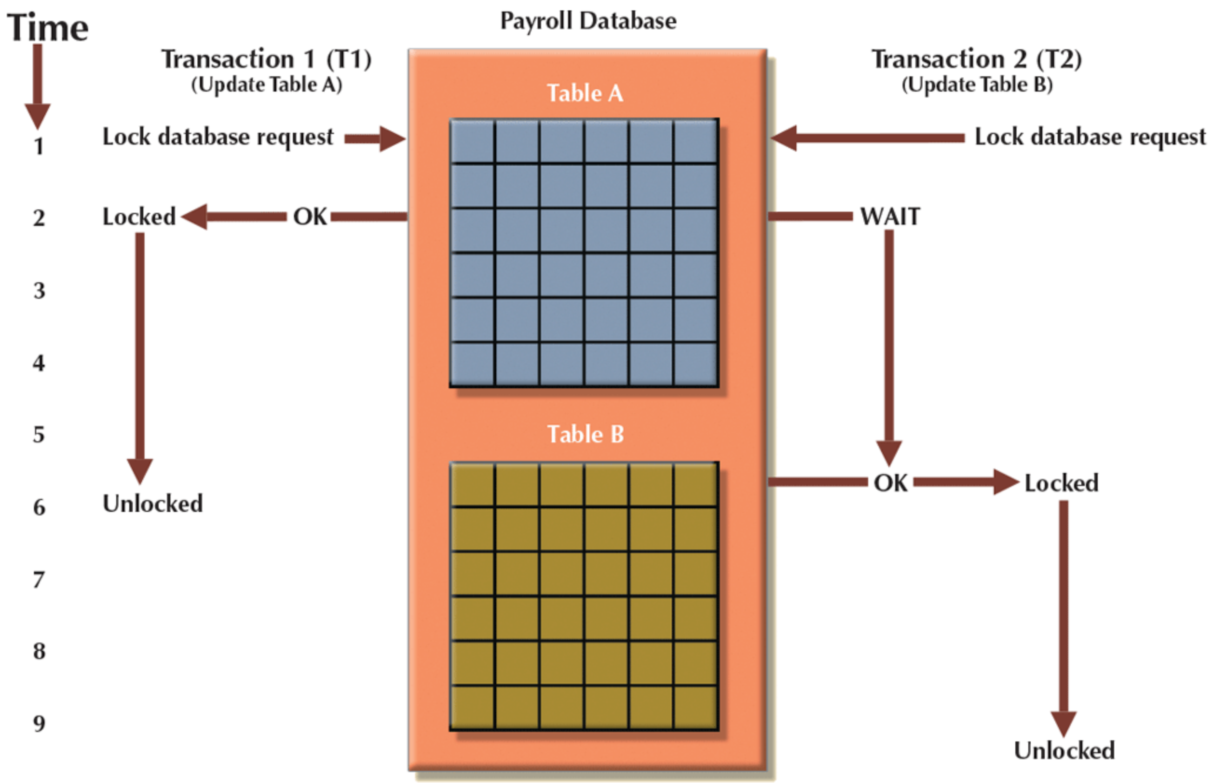
Inserts and delete rolled back, no change to Customers table

Result Grid Filter

a	b
10	Heikki
11	Elvis

Database locks can implement Atomic property, allow one transaction data access

Database-level lock



Database-level locks tie up the entire database while a transaction executes

- Good for batch processes
- Normally not used otherwise (defeats serialization!)

Based on Coronel and Morris

Locks implemented at the table-level allow unrelated transactions to run concurrently

Table-level lock

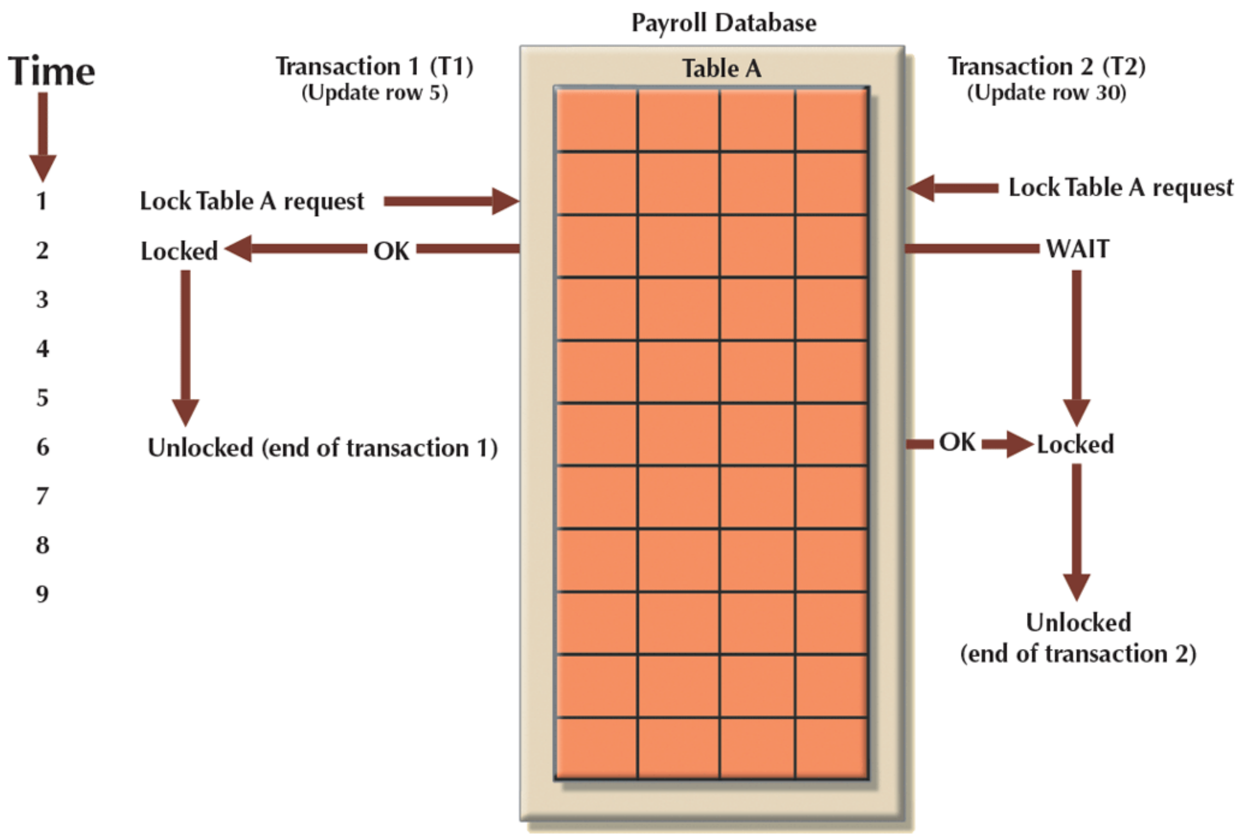


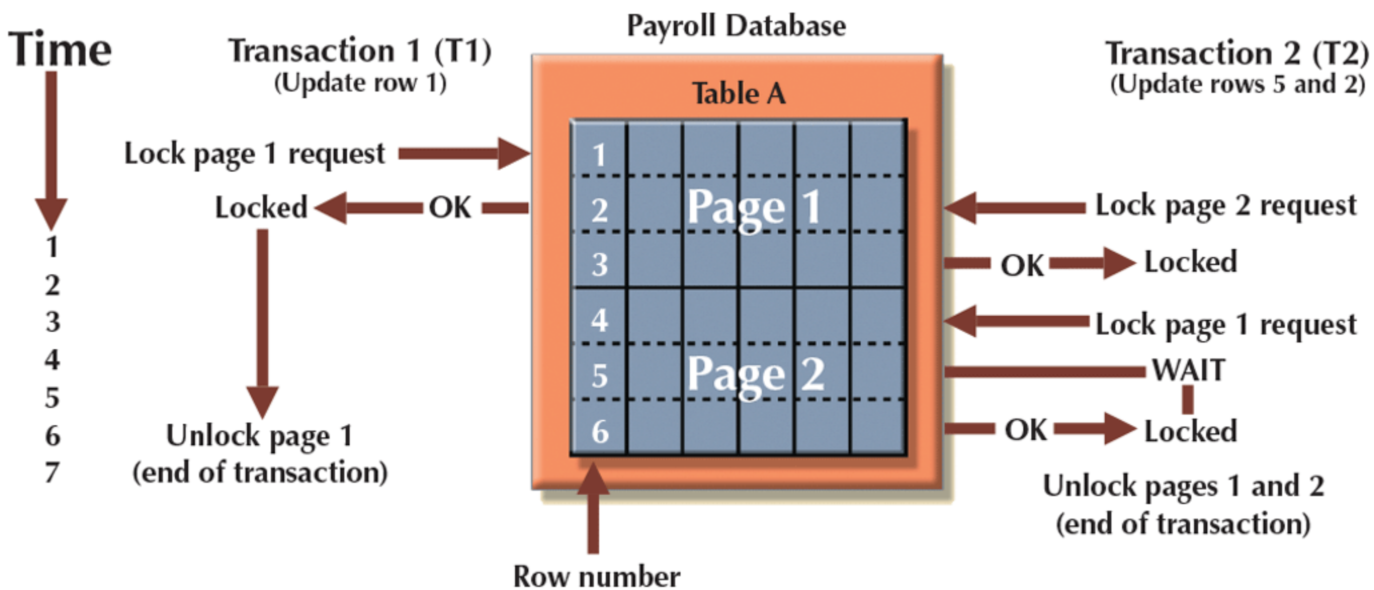
Table is locked during transaction

- Other tables can be accessed by different transactions
- Transactions attempting to access locked table must wait
- Lock manager notifies waiting a transaction it can proceed
- Still too coarse grained for many multi-users systems

Based on Coronel and Morris

Page-level locks allow concurrent access to different areas of one table

Page-level lock



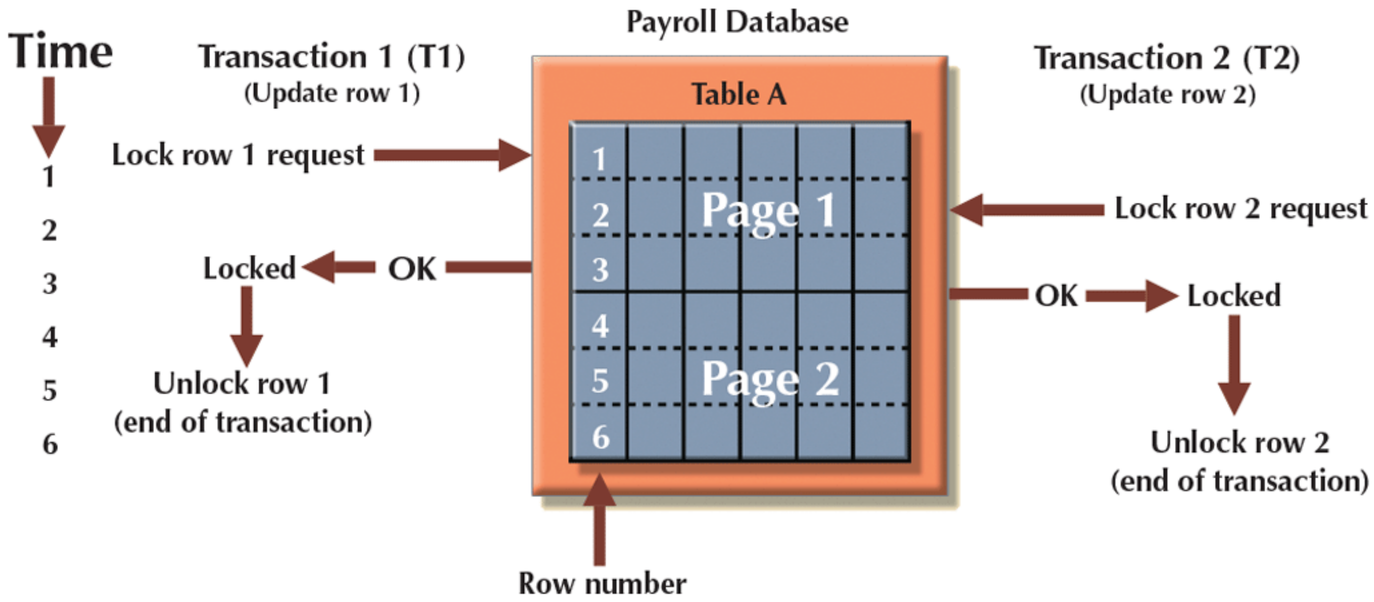
Database locks a disk page (disk block)

- Page normally fixed size (4K, 8K, or 16K)
- To write 73 bytes to a 4K page, must read all 4K bytes, make update, then write all 4K bytes back to disk
- Table may be several pages long
- This scheme is commonly used in practice
- Multiple processes can access same table simultaneously

Based on Coronel and Morris

Row-level locks allow concurrent access to different rows of a table

Row-level lock



Database locks a single row in a table

- **Improves availability of data**
- **Requires high overhead to track**
- **Not widely implemented (use page-level instead)**

Field-level locks are conceptually possible, but not often used (too much overhead)

Based on Coronel and Morris

Transaction log allows database to rollback if a transaction aborts

Transaction log Like our Audit table

If system failure or ROLLBACK, use log to return to prior consistent state

```
START TRANSACTION;
UPDATE Products SET Quantity = 23 WHERE ProductID = 1558;
UPDATE Customers SET Balance = 615.73 WHERE CustomerID = 1001;
COMMIT;
```

Log often kept on separate/ multiple disks (RAID)

Write changes to transaction log first , then update database (called a write-ahead-log protocol)

LogID	TransID	Prev	Next	Op	Table	RowID	Attribute	Before value	After value
341	101	Null	352	Start	** Start				
352	101	341	363	Update	Products	1558	Quantity	25	23
363	101	352	365	Update	Customer	1001	Balance	525.75	615.73
365	101	363	Null	Commit	** End				

Log and transaction IDs assigned by database

Prev and Next LogID

Operation, table, row, and attribute affected by change

Before and after values

Doesn't clean up variables that change or updates to other schemas

Based on Coronel and Morris

Transaction log allows database to rollback if a transaction aborts

Transaction log

```
START TRANSACTION;
UPDATE Products SET Quantity = 23 WHERE ProductID = 1558;
UPDATE Customers SET Balance = 615.73 WHERE CustomerID = 1001;
COMMIT;
```

LogID	TransID	Prev	Next	Op	Table	RowID	Attribute	Before value	After value
341	101	Null	352	Start	** Start				
352	101	341	363	Update	Products	1558	Quantity	25	23
363	101	352	365	Update	Customer	1001	Balance	525.75	615.73
365	101	363	Null	Commit	** End				

Two common approaches:

1. **Deferred-write – transaction log updated immediately, but database tables not updated until commit; if aborts, no changes made to tables; write "dirty buffers" at commit**
2. **Write-through – transaction log updated immediately, then database tables updated directly afterward; use transaction log to rollback if needed**

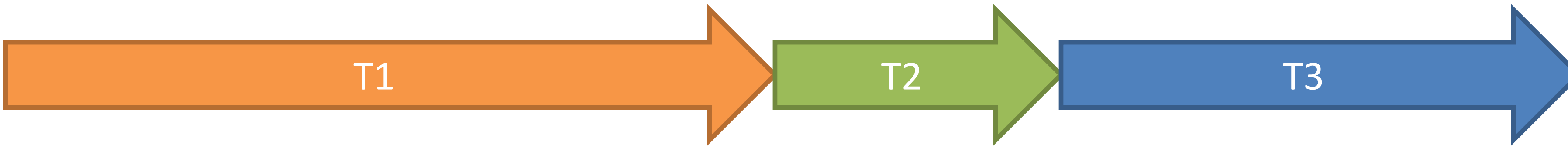
Based on Coronel and Morris

Agenda

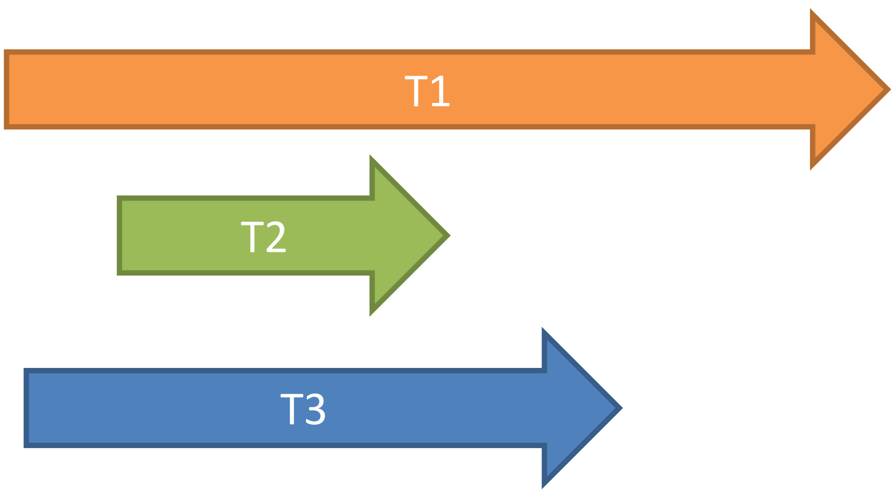
1. Database inconsistencies
2. ACID transactions
3. Concurrency/Isolation

Isolated property demands transactions do not interfere with each other

Serial schedule (run consecutively; first come, first served)



Interleaved schedule (serialized)



- Consecutive assumptions
  - Database starts in consistent state
  - Each transaction leaves database in consistent state when complete
  - Serial execution of transactions preserves consistency
- Serial schedule has poor performance; transactions must wait for preceding transactions to finish
- A schedule is ***serializable*** if it is interleaved, but equivalent to a serial schedule (not all schedules are serializable)
- Serialized schedule results in increased performance and ***Isolation***

Most combinations of reads and writes of related data can cause potential problems

Inconsistent retrieval and uncommitted data problems

T1 \ T2	Read	Write
Read	✓	✗
Write	✗	✗

If T1 and T2 operate on different data (e.g., T1 updates Employees, T2 updates Products)

- No problems running concurrently
- Each can run concurrently

If T1 and T2 operate on the *same* data

- Could have problems if one or both write data
- No problem to if both only read data

Most combinations of reads and writes of related data can cause potential problems

Inconsistent retrieval and uncommitted data problems

T1 \ T2	Read	Write
Read	✓	✗
Write	✗	✗

If T1 and T2 operate on different data (e.g., T1 updates Employees, T2 updates Products)

- No problems running concurrently
- Each can run concurrently

If T1 and T2 operate on the same data

- Could have problems if one or both write data
- No problem to if both only read data

Problems if one transaction reads and another writes

Inconsistent retrieval problem: read operation may read data that is no longer current

- Example: T1 calculates summary info over set of data while T2 updates portion of same data

Uncommitted data problem: if T1 reads after T2 writes, but T2 rolls back, T1's data is incorrect

Most combinations of reads and writes of related data can cause potential problems

Inconsistent retrieval and uncommitted data problems

	T2	
T1	**Read**	**Write**
Read	✓	✗
Write	✗	✗

If T1 and T2 operate on different data (e.g., T1 updates Employees, T2 updates Products)

- No problems running concurrently
- Each can run concurrently

If T1 and T2 operate on the same data

- Could have problems if one or both write data
- No problem to if both only read data

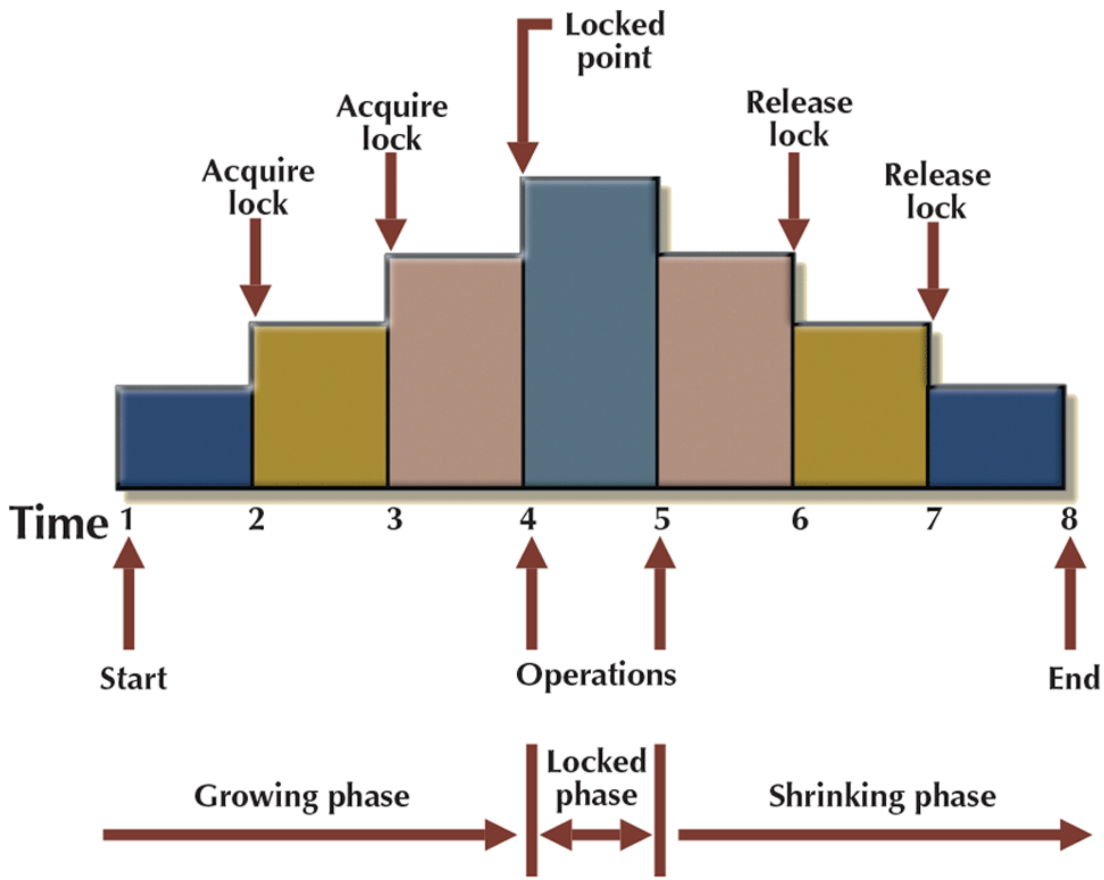
Problems if two transactions write the same data

Lost update problem:

- Each transaction reads the same data, changes it, then writes it back
- Last update wins

Two-phase locking protocol guarantees serializability, but may deadlock

Two-phase locking to ensure serializability



Phase 1: growing phase

- Acquire all needed locks before conducting data operations
- Two transaction cannot both hold conflicting lock (two reads are not a conflict)
- No data is affected until all locks are obtained (atomic)

Phase 2: shrinking phase

- Release all locks and cannot obtain a new lock until all locks released
- No unlock operation can precede a lock operation in same transaction

Ensures serializability but might deadlock!

Transactions can deadlock, either prevent them or detect and recover from them

Deadlocks

T1		T2
Exclusive lock (A)		
Read (A)		
A=A-1		
Write (A)		Shared lock (B)
		Read (B)
		Exclusive lock (A)
Exclusive lock (B)		

Shared lock – read only, many transactions can hold
Exclusive lock – for writes, only one transaction holds
T1: exclusive locked A and tries to exclusive lock B
T2: shared locked B and tries to exclusive lock A
Result is deadlock (exclusive lock request does not override existing shared lock)
System must roll back (and unlock) one transaction

To deadlock, four conditions must each be met

1. Mutual exclusion – only one transaction can access data at a time
2. Hold and wait – one process holding a resource while waiting for another
3. No preemption – no transaction can be forced to give up a lock
4. Circular wait – must be a circular chain of locks waiting for access

Break any of these condition and you can overcome deadlock

Transactions can deadlock, either prevent them or detect and recover from them

Deadlocks

T1		T2
Exclusive lock (A)		
Read (A)		
A=A-1		
Write (A)		Shared lock (B)
		Read (B)
		Exclusive lock (A)
Exclusive lock (B)		

Shared lock – read only, multiple transaction hold
Exclusive lock – write, only one transaction holds
T1: exclusive locked A and tries to exclusive lock B
T2: shared locked B and tries to exclusive lock A
Result is deadlock (exclusive lock request does not override existing shared lock)
System must roll back (and unlock) one transaction

Book covers graph-based methods that do not deadlock, but have high overhead

Deadlock options

- **Prevention** – never allow deadlock to occur
  - Make acquisition of all locks atomic operation (break hold and wait)
  - Use if probability of deadlocks is high
- **Recovery** – detect deadlock and roll back a victim transaction
  - Force one transaction to release locks and roll back (break no preemption)
  - Use if probability of deadlocks is low

SQL allows different levels of transaction isolation for improved performance

Isolation levels

Dirty read: a transaction can read data not yet committed by another transaction

SQL allows different levels of transaction isolation for improved performance

Isolation levels

Dirty read: a transaction can read data not yet committed by another transaction

Nonrepeatable read: a transaction reads a given row, then later reads the same row and may get different result if row updated or deleted by another process

```
-- Transaction log
START TRANSACTION;
SELECT ... ;
-- Begin some complex calculation that uses the following result
SELECT GPA FROM Student WHERE StudentID = 1234;
-- do some other stuff, then get that same GPA again to finish the calculation, and this
-- GPA should be the same as before or else had nonrepeatable read!
SELECT GPA FROM Student WHERE StudentID = 1234;
-- more stuff
COMMIT; -- This ends the transaction
```

SQL allows different levels of transaction isolation for improved performance

Isolation levels

Dirty read: a transaction can read data not yet committed by another transaction

Nonrepeatable read: a transaction reads a given row, then later reads the same row and may get different result if row updated or deleted by another process

Phantom read: a transaction execute a query, then later runs the same query and gets additional rows inserted by another process

```
-- Transaction log
START TRANSACTION;
SELECT ... ;
-- Begin some complex calculation that uses the following result
SELECT COUNT (*) FROM ENROLLMENT WHERE ClassDept = "CompSci";
-- do some other stuff, then get that same result again to finish the calculation, and this
-- count should be the same as before or else had phantom read!
SELECT COUNT (*) FROM ENROLLMENT WHERE ClassDept = "CompSci";
-- more stuff
COMMIT; -- This ends the transaction
```

SQL allows different levels of transaction isolation for improved performance

Isolation levels

Dirty read: a transaction can read data not yet committed by another transaction

Nonrepeatable read: a transaction reads a given row, then later reads the same row and may get different result if row updated or deleted by another process

Phantom read: a transaction execute a query, then later runs the same query and gets additional rows inserted by another process

Can set Isolation level per transaction to allow dirty, nonrepeatable, or phantom reads

Isolation level	Dirty Read	Nonrepeatable Read	Phantom Read	Comment
Read Uncommitted	OK	OK	OK	Reads uncommitted data; most serializable (best performance)
Read Committed	No	OK	OK	Does not allow dirty reads
Repeatable Read	No	No	OK	Allows phantom reads (MySQL default)
Serializable	No	No	No	Most restrictive (least serializable)