Australian National University

Week 10 Workshop - Database Transactions

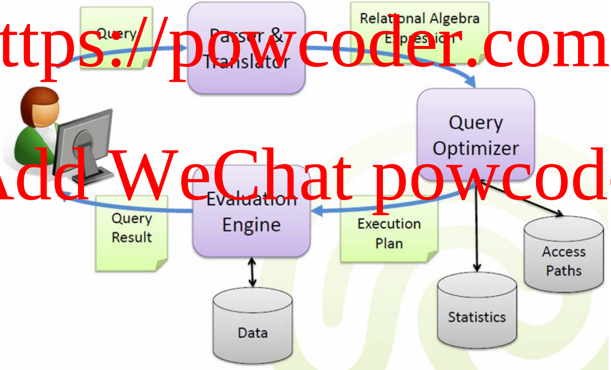
Transactions

- A **transaction** is a sequence of database operations grouped together for execution as a logic unit in a DBMS.

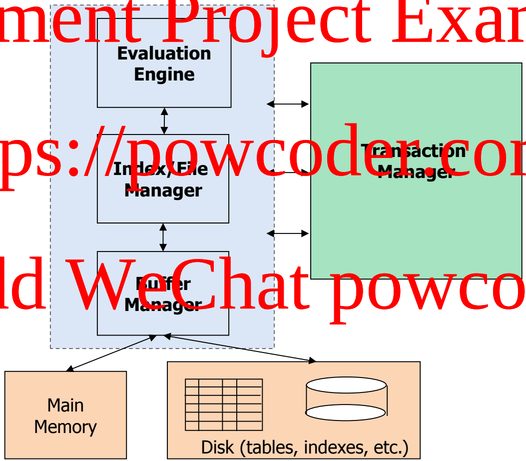
Steps	Transaction
	BEGIN TRANSACTION
1	SELECT balance FROM ACCOUNT WHERE name = 'Steve';
2	UPDATE ACCOUNT SET balance = balance-500 WHERE name='Steve';
3	SELECT balance FROM ACCOUNT WHERE name = 'Bob';
4	UPDATE ACCOUNT SET balance = balance+500 WHERE name = 'Bob';
5	COMMIT;

Transactions

- What's the difference between database transactions and programs written by a programming language like C, Java and Python?
- How are transactions handled in the query processing?



Transaction Manager - A Simplified View



Transactions - ACID Properties

Transactions

T_1 : BEGIN TRANSACTION
SELECT ...
UPDATE ...
COMMIT

T_2 : SELECT ...

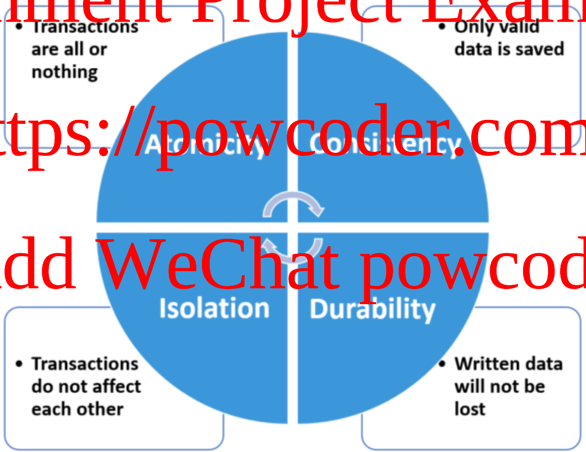
T_3 : INSERT ...

T_4 : BEGIN TRANSACTION
SELECT ...
DELETE ...
ABORT

ACID properties

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

Transactions - ACID Properties



Transactions - ACID Properties

ACID properties

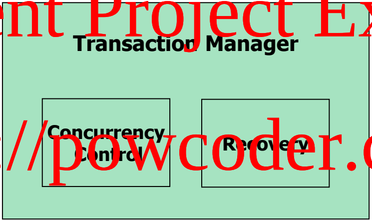
- Atomicity
- Consistency
- Isolation
- Durability

Transaction Manager

- Recovery
- Concurrency control

Consistency is the responsibility of an application developer.

Transaction Manager - Common Techniques



- **Logging** for recovery – assuring **atomicity/durability** of transactions
  e.g., Write-Ahead Log (WAL) Protocol

Logging - Introduction

- A **transaction log** is an append-only file that records changes to objects made by transactions.
- When multiple transactions run concurrently, log records are interleaved.

- A transaction log can be implemented as a separate file or set of files in the database.
- Recovery amounts to **either undoing or redoing changes from the log**:

  - **Undo** the operations that have not been committed;
  - **Redo** the operations that have been committed but not yet been written to disk.
- **Checkpoints** tell the points from which to begin applying transaction logs during database recovery.

Write-Ahead Log (WAL) Protocol

- **Write-Ahead Log** (WAL) requires that a record of every change to a database is available while attempting to recover from a crash.

  - Any change to an object is first recorded in the log, i.e., a record containing both the old and new values for the object.
  - A record in the log must be written to persistent storage before committing the transaction.

- Accordingly, the definition of a **committed transaction** is:

  “ **A transaction, all of whose log records, including a commit record, have been written to persistent storage**”.

Write-Ahead Log (WAL) Protocol

- Typical fields in a log record:

LSN
prevLSN
transID
type
pageID
length
offset
before-image
after-image

Additional fields for update log records

- Each log record has a unique id called LSN (Log Sequence Number).
- prevLSN is the LSN of the previous log record written by the same transaction.
- Possible types include: update, commit, abort, end, etc.

- Does WAL bring in some benefits for performance?
  - Often results in a significantly reduced number of disk writes
  - Supports one sync against the log file instead of potentially many against the data files
  - Enables online backup and point-in-time recovery

Transaction Manager - Recovery

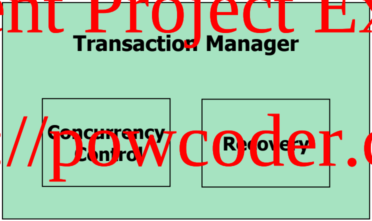
- Key concepts to aid in recovery:
  - **Transaction log**: records of database operations

    Write-Ahead Log (WAL)
    - **Undo** ...
    - **Redo** ...

  - **Checkpoint**: snapshot of the state of a database

    (Widely used in practice, but not covered in this course)

Transaction Manager - Common Techniques



- **Logging** for recovery – assuring **atomicity/durability** of transactions

  e.g., Write-Ahead Log (WAL) Protocol

- **Locking** for concurrency control – assuring **isolation** of transactions

  e.g., Two-Phase Locking (2PL) Protocol

Locking - Introduction

- A **lock** is associated with an object, e.g., file, table, record, page, etc.

- Two main types of locks:

  - **Shared lock** (read-lock): for reading an object by a transaction
  - **Exclusive lock** (write-lock): for writing an object by a transaction

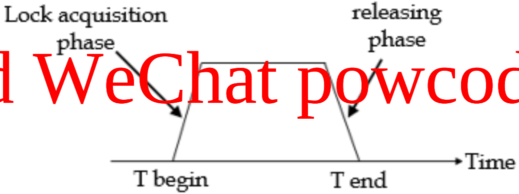
  (Note: there are other types of locks defined by different DBMSs)

- Lock compatibility:

Lock type	read-lock	write-lock
read-lock	Yes	No
write-lock	No	No

Two-Phase Locking (2PL) Protocol

- Locks are handled in two phases:
  - **Expanding**: locks are acquired and no locks are released.
  - **Shrinking**: locks are released and no locks are acquired.



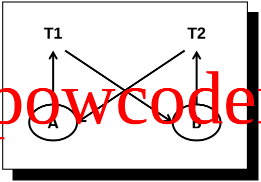
Australian National University

Two-Phase Locking (2PL) Protocol

Bad news:

- 2PL can radically limit interleaving among transactions in some cases ...
- 2PL may be subject to **deadlocks**, i.e., the mutual blocking of two or more transactions.

Step	T_1	T_2
1	lock-r(A)	
2	read(A)	
3		lock-r(B)
4		read(B)
5	lock-w(B)	
6	write(B)	
7		lock-w(A)
8		write(A)



- T_1 is waiting for T_2 to get a write-lock on B. T_2 is waiting for T_1 to get a write-lock on A.

Two-Phase Locking (2PL) Protocol

Good news:

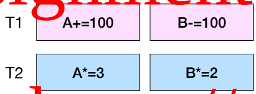
- 2PL makes interleaving safe, i.e., guarantee the serializability property for transactions.

  - **Serializability** means that a resulting database state is equal to a database state of running transactions serially.

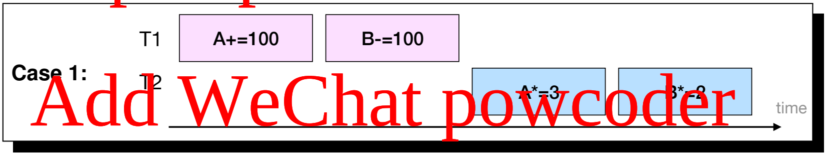
  - Serializability is the major **correctness criterion** for concurrent transactions.

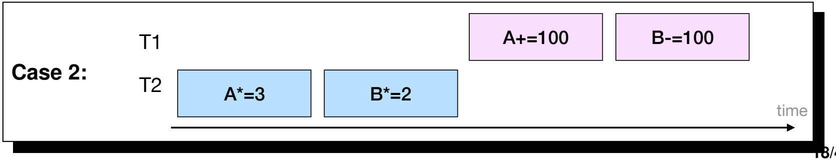
Serializability - Example

- Consider $A = 200$ and $B = 500$, and we have two concurrent transactions:



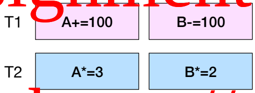
- Serializable transactions:





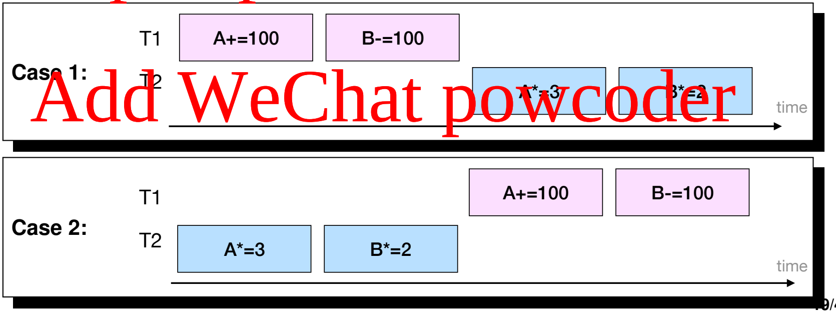
Serializability - Example

- Consider $A = 200$ and $B = 500$, and we have two concurrent transactions:



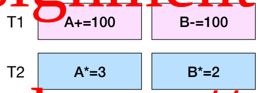
- **Case 1**: A=900 and B=800
- **Case 2**: A=700 and B=900

- Serializable transactions:



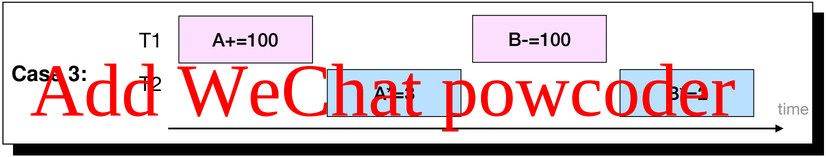
Serializability - Example

- Consider $A = 200$ and $B = 500$, and we have two concurrent transactions:



- **Case 1**: A=900 and B=800
- **Case 2**: A=700 and B=900

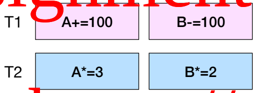
- Are the following transactions serializable?



- Yes. A=900 and B=800 $\hookrightarrow$ equivalent to Case 1!

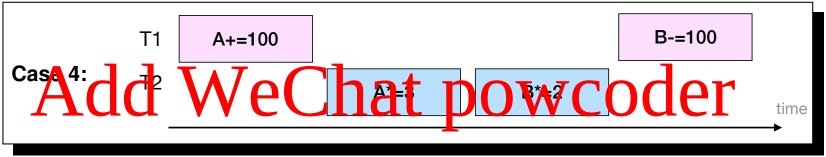
Serializability - Example

- Consider $A = 200$ and $B = 500$, and we have two concurrent transactions:



- **Case 1**: A=900 and B=800
- **Case 2**: A=700 and B=900

- Are the following transactions serializable?



- No. A=900 and B=900 $\hookrightarrow$ not equivalent to Case 1 or Case 2!

Problems in Concurrent Transactions

- If no concurrency control for transactions, some problems may occur:

Lost update	Dirty read	Unrepeatable read	Phantom read
write → write	write → read	read → write (read)	read → write (read)

Problems in Concurrent Transactions

- If no concurrency control for transactions, some problems may occur:

Lost update	Dirty read	Unrepeatable read	Phantom read
write ↷ write	write ↷ read	read ↷ write (read)	read ↷ write (read)
objects	objects	objects	a set of objects

The Lost Update Problem - Another Example

- Ben and Amy have the same salary. T_1 sets their salaries to \$80,000, and T_2 sets their salaries to \$90,000.

1. If executing T_1 and T_2 sequentially,
   - for T_1; T_2, both receive \$90,000.
   - for T_2; T_1, both receive \$80,000.

   $\hookrightarrow$ Either is acceptable from the transaction viewpoint.

2. If executing T_1 and T_2 concurrently, we may have:

	T_1	T_2
1	write(A) (A:=80000)	
2		write(A) (A:=90000)
3		write(B) (B:=90000)
4		commit
5	write(B) (B:=80000)	
6	commit	

$\hookrightarrow$ It is not acceptable!

The Dirty Read Problem - Another Example

- Both Ben and Amy are rewarded a bonus \$5,000 and a pay rise 5%. T_1 increases their salaries with \$5,000 and T_2 increments their salaries by 5%.

  1. If executing T_1 and T_2 sequentially, they would have .... Also, T_1 or T_2 could abort for some reasons. $\hookrightarrow$ all are acceptable from the transaction viewpoint.

  2. If executing T_1 and T_2 concurrently, we may have:

	T_1	T_2
1	read(A)	
2	write(A) (A:=A+5000)	
3		read(A)
4	read(B)	
5	write(B) (B:=B+5000)	
6	abort	
7		write(A) (A:=A+A×5%)
8		read(B)
9		write(B) (B:=B+B×5%)
10		commit

$\hookrightarrow$ It is not acceptable!

The Unrepeatable Read Problem - Another Example

- Amy and Ben are using a website to book flight tickets to Brisbane.
  - Amy signs on first to see that only one ticket is left, and finds it expensive.
  - Amy takes time to decide. Ben signs on later and also finds one ticket left, orders it instantly, and logs off.
  - Amy decides to buy a ticket, and finds no tickets left.

	T_1 (from Amy)	T_2 (from Ben)
1	read(X)	
2		read(X)
3		write(X) (X:=X-1)
4		commit
5	read(X)	

- This situation can never arise in a serial execution of T_1 and T_2.

The Phantom Read Problem - Another Example

- Amy is 30 years old, but her age in the table players is mistakenly recorded as 40. Ben is 28 years old and his age is correctly recorded in players.
- Suppose that we have the following two current transactions:

T_1:
```
SELECT * FROM players
WHERE age<32;
...
SELECT * FROM players
WHERE age<32;
COMMIT;
```

T_2:
```
UPDATE players
SET age=30
WHERE rating=8 and name='Amy';
COMMIT;
```


	T_1	T_2
1	read(players)	
2		read(players)
3		write(players)
4		commit
5	read(players)	
6	commit	

- This situation also can never arise in a serial execution of T_1 and T_2.

Discussion

- What are the differences between "unrepeatable read" and "phantom read"?

- **Unrepeatable read**
  - Executing the same SELECT twice yields the same tuples, but attribute values might be different;
  - May occur when reading objects that are affected by UPDATE from another transaction;
  - Can be prevented using record-level locking.
- **Phantom read**
  - Executing the same SELECT twice yields two different sets of tuples;
  - May occur when querying a set of tuples that are affected by INSERT/DELETE/UPDATE from another transaction;
  - Can be prevented using table-level locking.

What Should We lock?

- Consider the following two concurrent transactions again:

T_1:
```
SELECT * FROM players
WHERE age<32;
...
SELECT * FROM players
WHERE age<32;
COMMIT;
```

T_2:
```
UPDATE players
SET age=30
WHERE rating=8 and name='Amy';
COMMIT;
```


- What objects should the DBMS lock in order to avoid the phantom read problem?

  - **Table-level locks**
    e.g., read-lock on players for T_1, write-lock on players for T_2
  - **Record-level locks**
    e.g., read-lock on every record with age<32 for T_1, write-lock on every record with rating=8 and name='Amy' for T_2
  - ...

Transaction Support in SQL

- An explicit transaction may have no `BEGIN TRANSACTION` statement, but must be ended with either `COMMIT` or `ABORT` (`ROLLBACK`) statement.
- When no explicit transaction statements are given, each single SQL statement is considered to be a transaction.

- To give programmers more control over transaction overhead, SQL allows them to specify **isolation level**, i.e., the degree of interference that a transaction is prepared to tolerate on concurrent transactions.
  - Key idea:

> To trade off **consistency** (i.e., increased risk of violating database integrity) with **performance** (i.e., greater concurrent access to data)

Isolation Levels

- SQL-92 defines four isolation levels:
  1. **Read Uncommitted**
  2. **Read Committed**
  3. **Repeatable Reads**
  4. **Serializable**

- To specify an isolation level, e.g.,

```
SET TRANSACTION ISOLATION LEVEL serializable;
```

- The SQL standard does not impose a specific locking scheme or mandate particular behaviors.

Isolation Levels

- The intention is to prohibit certain problems:

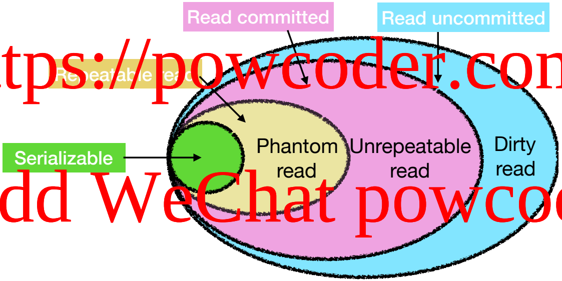
Isolation Level	Dirty Read	Unrepeatable Read	Phantom Read
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

- Different DBMSs implement isolation levels quite differently.
- The isolation level required for **Lost Update** is debatable (depending on a DBMS's implementations). But in general, it may require the highest level SERIALIZABLE to prevent it. [1]

[1] https://drtom.ch/posts/2011/11/12/The_Lost_Update_Problem_-_Part_1/

Isolation Levels - Concurrency Control

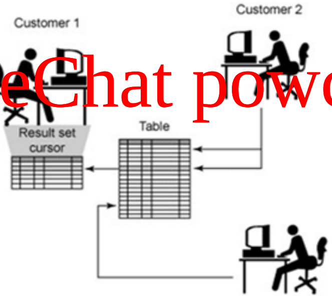
- A DBMS provides different levels of isolation → different degrees of concurrency control to prevent different problems.



- Concurrency control is **NOT** binary in a database system.

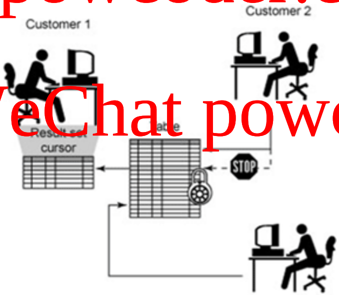
Isolation Levels - Read Uncommitted

- **Read Uncommitted** is the least restrictive isolation level.
- One transaction can see changes made by other transactions which are not yet committed. This can be quite dangerous.
- Use it when executing queries over read-only data or if it does not matter whether a query returns uncommitted data.



Isolation Levels - Read Committed

- **Read Committed**: One transaction only sees committed changes by other transactions
- It is the most commonly used isolation level in database applications.
- Use it when you want to maximize concurrency between applications but do not want queries to see uncommitted data.

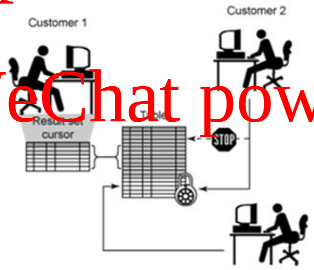


Isolation Levels - Repeatable Reads

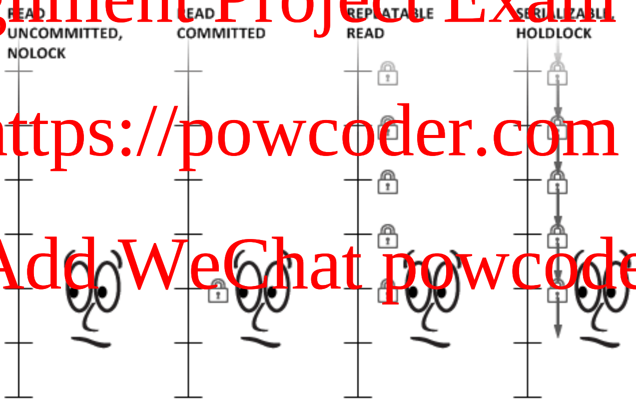
- **Repeatable Reads**: The objects touched by a transaction are locked and cannot be updated or deleted by a concurrent transaction.
- Use it when you want some level of concurrency between applications but do not expect individual objects to be changed during a transaction.

Isolation Levels - Serializable

- **Serializable** is the highest isolation level. All transactions are totally isolated from other transactions. It is safe but may cause significant performance hit.
- Use it when you want some level of concurrency between applications but do not expect that a query returns different sets of results when running at different times.



Locks Taken by SQL Server for Isolation Levels [2]



[2] http://michaeljswart.com/2012/06/visualizing-transaction-isolations-for-sql-server/

Wrap-up - Isolation Levels

- A lower isolation level increases **the ability of many users to access data at the same time**, but also increases the number of concurrency effects (such as dirty reads or lost updates) users might encounter.

- Conversely, a higher isolation level **reduces the types of concurrency effects that users may encounter**, but requires more system resources and increases the chances that one transaction will block another.

- Choosing the appropriate isolation level depends on **balancing**

  - **the data integrity requirements of the application**

    against

  - **the overhead of each isolation level**.

Research Topics

Research Topics

- This is an active research area covering many interesting research topics.
- Historically, much of the work has been done in the context of relational database systems.

- However, the ideas in general are independent of whether the underlying system is a relational database system or something else.
  - Distributed database systems
  - Graph database systems
  - Document-oriented database systems
  - ...

Research Topics

- Distributed transactions