Transactions

Universidad de Oviedo

This lecture is mostly taken from *CS145 Introduction to Databases* lecture given by **Christopher Re** at **Stanford University**

School of Computer Science

2021-2022

Introduction to transactions

Properties of transactions

Transactions, Scheduling and Serializability

Anomalies if we are careless

How to treat Conflicts carefully?

JDBC transactions management

Section 1

Introduction to transactions

Prequel

Performance of basic storage

Peter Norvig's blog norvig.com

Latency numbers every engineer should know

Ballpark timings on typical PC:

execute typical instruction	1/1,000,000,000 sec = 1 nanosec
fetch from L1 cache memory	0.5 nanosec
fetch from L2 cache memory	7 nanosec
Mutex lock/unlock	25 nanosec
fetch from main memory	100 nanosec
send 2K bytes over 1Gbps network	20,000 nanosec
read 1MB sequentially from memory	250,000 nanosec
fetch from new disk location (seek)	8,000,000 nanosec
read 1MB sequentially from disk	20,000,000 nanosec
send packet US to Europe and back	150 milliseconds = 150,000,000 nanosec

Srigi
@srigi

"Latency Numbers Every Programmer Should Know"

It is hard for humans to get the picture until you translate it to "human numbers":

1 CPU cycle	0.3 ns	1 s
Level 1 cache access	0.9 ns	3 s
Level 2 cache access	2.8 ns	9 s
Level 3 cache access	12.9 ns	43 s
Main memory access	120 ns	6 min
Solid-state disk I/O	50-150 μs	2-6 days
Rotational disk I/O	1-10 ms	1-12 months
Internet: SF to NYC	40 ms	4 years
Internet: SF to UK	81 ms	8 years
Internet: SF to Australia	183 ms	19 years
OS virtualization reboot	4 s	423 years
SCSI command time-out	30 s	3000 years
Hardware virtualization reboot	40 s	4000 years
Physical system reboot	5 m	32 millenia

Prequel

Performance of basic storage

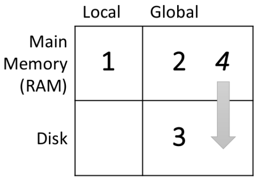
Keep in mind the tradeoffs here as motivation for the mechanisms we introduce

- **Main memory**: fast but limited capacity, volatile
- **Disk**: slow but large capacity, durable

How do we effectively utilize both ensuring certain critical guarantees?

Memory organization associated with a DBMS

Three Types of Regions of Memory

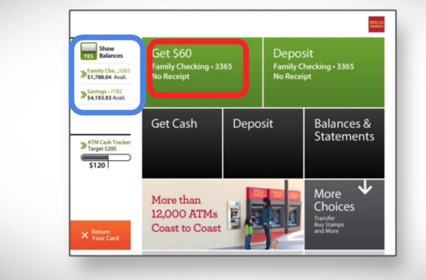


1. **Local**: Each process has its own local memory, where it stores values that only it *sees*
2. **Global** or **Shared**: Each process can read from / write to shared data in main memory (Database Buffer Cache)
3. **Disk**: Global memory can read from / flush to disk
4. **Log**: Assume on stable disk storage- spans both main memory and disk ...

Log is a sequence from main memory $\rightarrow$ disk

Flushing to disk = writing to disk from main memory

Get cash from an ATM



Read Balance
Give money
Update Balance

vs

Read Balance
Update Balance
Give money

Identify some requirements:

- Resilience to system failures, such as hardware failures or system crashes.
- Durability so if the operation success, new info will last in time.
- Concurrent database access to achieve better performance.

Further reading [1]

Transactions

A transaction (*txn*) *is a sequence of one or more operations (reads or writes) which reflects a single real-world transition, treated as a unit.*

If the system fails, each transaction's changes are reflected either entirely or not at all.

```
START TRANSACTION
    UPDATE Product
    SET Price = Price – 1.99
    WHERE pname = 'Gizmo'
COMMIT
```

- *Transfer money between accounts*
- *Purchase a group of products*
- *Register for a class (either waitlist or allocated)*

- By default, individual SQL Statements, are automatically *transactional* even if they insert, update, or delete millions of rows.
- **AutoCommit** can turn on/off each statement into transaction grouping multiple statements together in the same TX

```
UPDATE Bank SET amount = amount - 100 WHERE name = 'Bob'
```

```
UPDATE Bank SET amount = amount + 100 WHERE name = 'Alice'
```

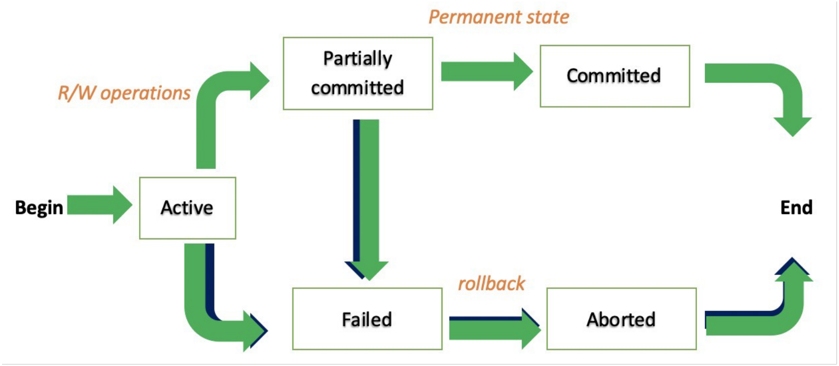
```
SET AUTOCOMMIT OFF
UPDATE Bank SET amount = amount - 100 WHERE name = 'Bob'
UPDATE Bank SET amount = amount + 100 WHERE name = 'Alice'
```

- In SQL, a transaction begins implicitly.
- A transaction ends (and a new one begins) by:
  - at the end of the current connection or with
  - **Commit** ends current transaction (flush to disk) and begins a new one, or with
  - **Rollback** causes current transaction to abort.

```
SET AUTOCOMMIT OFF
UPDATE Bank SET amount = amount - 100 WHERE name = 'Bob'
UPDATE Bank SET amount = amount + 100 WHERE name = 'Alice'
COMMIT
```

```
SET AUTOCOMMIT OFF
UPDATE Bank SET amount = amount - 100 WHERE name = 'Bob'
UPDATE Bank SET amount = amount + 100 WHERE name = 'Alice'
ROLLBACK
```

And from birth to death . . .

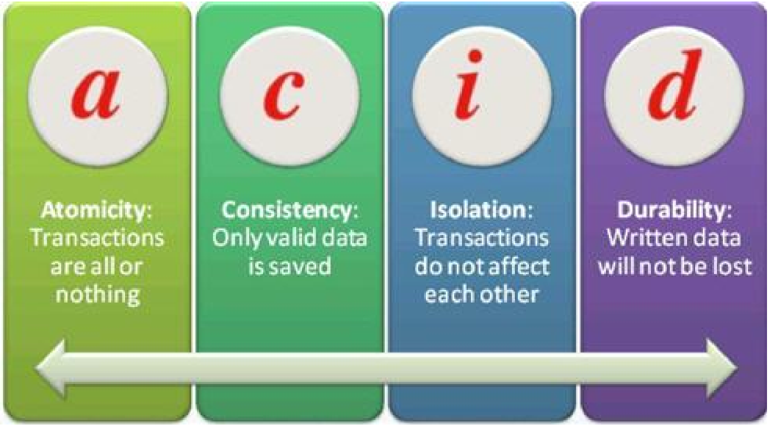


Section 2

Properties of transactions

ACID

Quickstart guide



Watch this video

`https://www.youtube.com/watch?v=yzzGOCuNqIw`

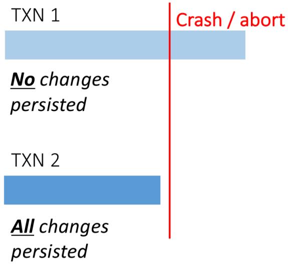
ACID: **A**tomicity

TXN's activities are atomic: **all or nothing**

- Intuitively: in the real world, a transaction is something that would either occur *completely or not at all*

Two possible outcomes for a TXN

- It **commits**: all the changes are made
- It **aborts**: no effects should be seen; **rollback**

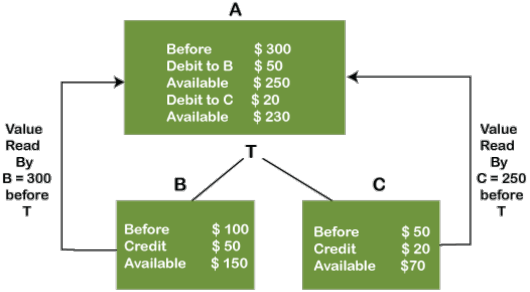


ACID: Consistency

There are certain statements about data that must always be true

- Passport number (not PK) is unique
- In a transfer operation, money is the same than before

If a database is consistent before a transaction is executed, then the database must also be consistent after the transaction is executed.



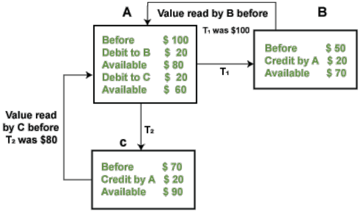
Data Consistent

ACID: Isolation

txn run **concurrently** to improve performance

Isolation: it seems as if **each transaction executes one after another**

- DBMS handles the details of *interleaving* various TXNs
- A txn should not be able to observe changes from other transactions during the run



Isolation - Independent execution of T_1 & T_2 by A

ACI**D**: **D**urability

- Once a transaction has been **committed**, its effects will remain, even after a system failure.
- While a transaction is under way (**partially committed**), the effects are not persistent. If the database crashes, backups will always restore it to a consistent state prior to the transaction starts.

Section 3

Transactions, Scheduling and Serializability

Transactions

Formally, transactions are defined as:

A transaction is a list of actions.
The **actions** are

- reads (written **R(O)**) and
- writes (written **W(O)**)

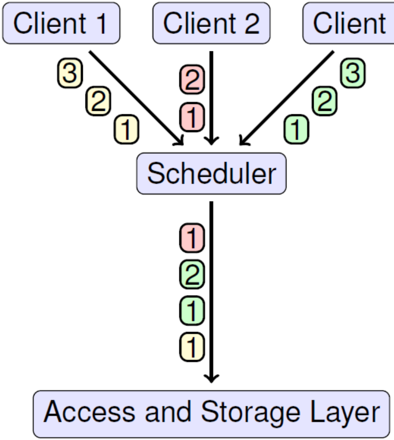
of database objects O.
Transactions end with **Commit** or **Abort**[a].

[a] These are sometimes omitted if not relevant

Example Transaction
T_1 : `R(V), R(Y), W(V), W(C), Commit`

Schedules

The scheduler decides the **execution order of concurrent database access**.



A **schedule** is a list of actions from a set of transactions.

Intuitively, this is a plan on how to execute transactions.

The order in which 2 actions of a transaction T appear in a schedule must be the same order as they appear in T.

T_1 : R(V) W(V)
T_2 : R(Y) W(Y)

Which of the following is a schedule of these transactions?

S_1 :

T_1	R(V)			W(V)
T_2		R(Y)	W(Y)	

S_2 :

T_1	W(V)			R(V)
T_2		R(Y)	W(Y)	

Serializable Schedules

A schedule is **serial** if the actions of the different transactions are not interleaved; they are executed one after another.

S_1 :

T_1			R(V)	W(V)
T_2	R(Y)	W(Y)		

A schedule is **serializable** if its effect on the database is the same as that of some serial schedule.

Quiz Serializable Schedules

We usually only want to allow serializable schedules. Why?

Consider two TXNs

Each action reads a value from global memory, does some operation and then writes one back to it

```
T1: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt + 100
        WHERE Name = 'A'

        UPDATE Accounts
        SET Amt = Amt - 100
        WHERE Name = 'B'
COMMIT
```

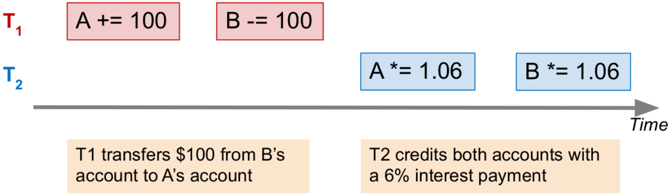
```
T2: START TRANSACTION
        UPDATE Accounts
        SET Amt = Amt * 1.06
COMMIT
```

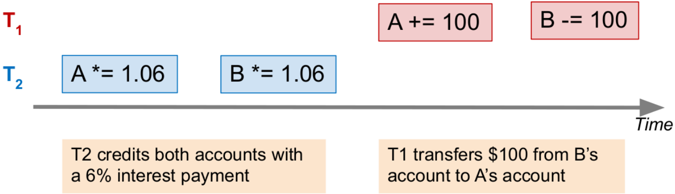
T1 transfers $100 from B's account to A's account

T2 credits both accounts with a 6% interest payment

Let's look at the TXNs in a timeline view

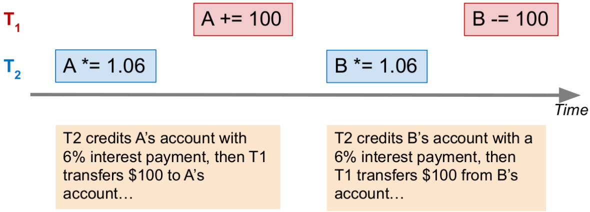
Serial execution



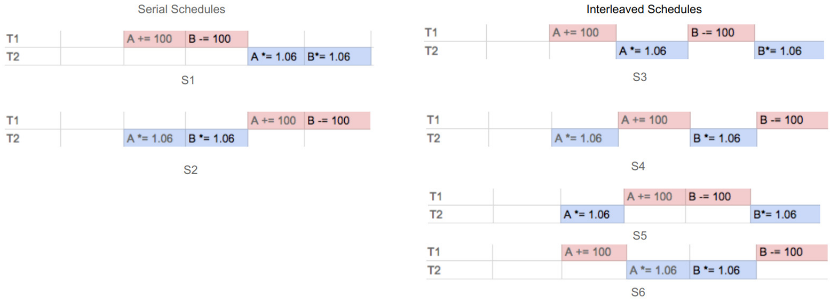


Interleaving transactions to boost performance

Other (non serial schedules) have TXNs occur concurrently by interleaving their component actions (R/W)



Scheduling examples



Serial schedules	S_1, S_2
Serializable schedules	S_3, S_4
Equivalent schedules	$< S_1, S_3 > < S_2, S_4 >$
Non-serializable (Bad) schedules	$< S_5, S_6 >$

To improve the throughput, concurrent transactions are allowed; but this can lead to wrong data

DBMS must guarantee that with that **interleaved schedule** isolation and consistency are maintained

- Must be *as if* the TXNs had executed serially!

With great power comes great responsibility

A<u>CI</u>D

Checking conflict-serializability

Given a schedule we can create a **precedence graph**:

- The graph has a node for each transaction.
- There is an edge from T_1 to T_2 if there is a conflicting action between T_1 and T_2 in which T_1 occurs first.

T_1	W(V)		W(V)
T_2		R(V)	

$$T_1 \rightleftarrows T_2$$

T_1	R(V)		W(V)
T_2		R(V)	

$$T_1 \longleftarrow T_2$$

T_1		W(Y)			
T_2	R(V)			R(Y)	W(Z)
T_3			W(V)		

$$\begin{tikzcd} T_1 \arrow[rr] & & T_2 \arrow[dl] \\ & T_3 & \end{tikzcd}$$

Checking conflict-serializability

Checking Conflict-Serializability

A schedule is **conflict-serializable** if and only if there is **no cycle** in the precedence graph!

T_1	W(V)		W(V)
T_2		R(V)	

$$T_1 \rightleftarrows T_2$$

T_1	R(V)		W(V)
T_2		R(V)	

$$T_1 \longleftarrow T_2$$

T_1			W(Y)		
T_2	R(V)			R(Y)	W(Z)
T_3		W(V)			

$$\begin{tikzcd} T_1 \arrow[rr] & & T_2 \arrow[dl] \\ & T_3 & \end{tikzcd}$$

Schedules 2 and 3 have no cycles in their precedence graph. They are conflict serializable!

Exercise

Consider txn[1] T_1, T_2 and T_3 and two possible schedules

Schedule 1	Schedule 2
$R_1(X)$	$R_1(X)$
$R_3(X)$	$R_3(X)$
$W_1(X)$	$W_3(X)$
$R_2(X)$	$W_1(X)$
$W_3(X)$	$R_2(X)$

Determine which schedule, if any, is serializable.

[1] W operations write a value different from the original

Are these transactions conflict serializable?

Watch this video
`https://www.youtube.com/watch?v=U3SHusK80q0`

Section 4

Anomalies if we are careless

Ensuring serializability

So far, we have seen a sufficient condition that allows us to check whether a schedule is serializable.

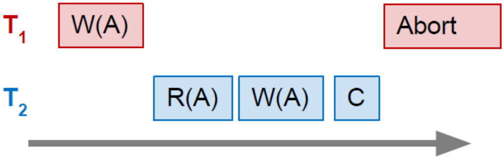
But how to ensure serializability during runtime?

Challenge: the system does not know in advance which transactions will run and which items they will access.

Some problems will arise, by sure!!

Classic Anomalies with Interleaved Execution

Dirty read / Reading uncommitted data

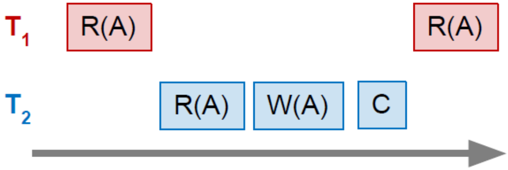


1. T_1 writes some data to A
2. T_2 reads from A, then writes back to A & commits
3. T_1 then aborts - now T_2's result is based on an obsolete / inconsistent value

Occurring with / because of a WR conflict

Classic Anomalies with Interleaved Execution

Unrepeatable read



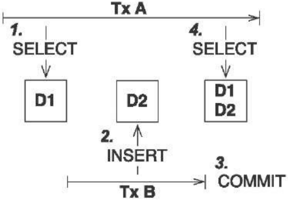
1. T_1 reads some data from A
2. T_2 writes to A
3. Then, T_1 reads from A again and now gets a different / inconsistent value

Occurring with / because of a **RW conflict**

Classic Anomalies with Interleaved Execution

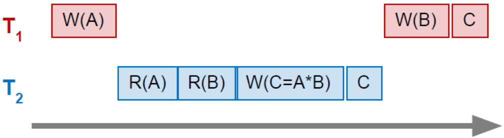
Phantom Read

Phantom Read occurs when two same queries are executed inside a transaction, but the rows retrieved by the two, are different.



Classic Anomalies with Interleaved Execution

Inconsistent read / Reading partial commits

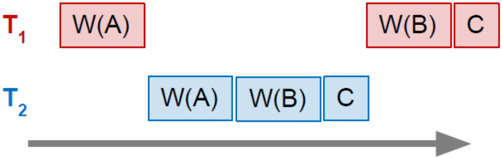


1. T_1 writes some data to A
2. T_2 reads from A and B, and then writes some value which depends on A & B
3. T_1 then writes to B - now T_2's result is based on an incomplete commit

Again, occurring because of a WR conflict

Classic Anomalies with Interleaved Execution

Partially-lost update



1. T_1 writes some data to A
2. T_2 writes to A and B
3. T_1 then writes to B; now we have T_1's value for B and T_2's value for A - not equivalent to any serial schedule!

Occurring because of a WW conflict

Exercises

Let us assume that initial value of data items A and B are both 1000. Consider transaction T_1 transfers 50 from A to B and transaction T_2 withdraws 10% of amount from A.

	T_1	T_2	
i_{11}	R(A)	R(A)	i_{21}
i_{12}	A=A-50	aux=A*0.1	i_{22}
i_{13}	W(A)	A=A-aux	i_{23}
i_{14}	R(B)	W(A)	i_{24}
i_{15}	B=B+50	COMMIT	i_{25}
i_{16}	W(B)		
i_{17}	COMMIT		

Table 1 shows final values if txn are executed in serial order.

If T1 then T2	If T2 then T1
Final values are, A = 855 B = 1050	Final values are, A = 850 B = 1050

Table 1: Final values of A and B if T1 and T2 are executed in serial order

Now, consider this schedule i_{11}, i_{12}, i_{21}, i_{22}, i_{23}, i_{13}, i_{14}, i_{24}, i_{25}, i_{15}, i_{16}, i_{17}. Identify the anomaly and the conflict (RW, WR, WW)

Let us suppose the initial value of A and B are 1000; here, transaction T_1 transfers 50 from account A to B, and transaction T_2 calculates the 10% interest on the balance of A and credits A with the interest.

	T_1	T_2	
i_{11}	R(A)	R(A)	i_{21}
i_{12}	A=A-50	aux=A * 0.1	i_{22}
i_{13}	W(A)	A=A+aux	i_{23}
i_{14}	R(B)	W(A)	i_{24}
i_{15}	B=B+50	COMMIT	i_{25}
i_{16}	W(B)		
i_{17}	COMMIT		

Now, consider interleaving $i_{11}, i_{12}, i_{13}, i_{21}, i_{22}, i_{23}, i_{24}, i_{25}$. While T_2 was executing, T_1 decided to **rollback** for some reason. Identify the anomaly and the conflict (RW, WR, WW)

Section 5

How to treat Conflicts carefully?

Pesimistic approach: Lock-based concurrency control

Avoid conflicts by employing a **pessimistic** locking mechanism:
Lock-based Concurrency Control

- every read requires a shared lock acquisition, while a write operation requires taking an exclusive lock.
- a shared lock blocks Writers, but it allows other Readers to acquire the same shared lock
- an exclusive lock blocks both Readers and Writers concurring for the same lock

However, locking incurs contention, and contention affects scalability.

Schedule with explicit lock actions

T_1			X(B)	W(B)	U(B)					
T_2	S(A)	R(A)				X(B)	W(B)	U(A)	U(B)	

Here we use the following abbreviations:

- S(A) = shared lock on A
- X(A) = exclusive lock on A
- U(A) = unlock A, or if more precision is needed
  - US(A) = unlock shared lock on A
  - UX(A) = unlock exclusive lock on A

Multiversion Concurrency Control

Optimistic approach

- hope for the best
- let transactions freely proceed with read/write operations
- only at commit, check that no conflicts have happened

Transactions proceed in three phases[2]:

1. **Read phase**: Execute transaction, but do not write data back to disk. Collect updates in the transaction's private workspace.
2. **Validation phase**: When the transaction wants to commit, the DBMS test whether its execution was correct (only acceptable conflicts happened). If not, abort the transaction.
3. **Write phase**: Transfer data from private workspace into database.

[2] phases 2 and 3 are performed in a uninterruptible critical section (also called val-write phase)

Is this schedule serializable?

T_1	R(X)	W(X)			R(Y)	W(Z)
T_2			R(X)	W(Y)		

No

But what if we had a copy of the old values available?

Then we could do:

Multi-version

T_1	R(X)	W(X)			R(Y-old)	W(Z)
T_2			R(X)	W(Y)		

This is can be serialised to:

T_1	R(X)	W(X)	R(Y)	W(Z)		
T_2					R(X)	W(Y)

Isolation levels

The application does not typically set locks or take snapshots manually.

It can provide the DBMS with hints to help it improve concurrency: **Isolation levels**

An isolation level represents a particular locking strategy employed in the database system to improve data consistency.

Or, in MVCC, the way to implement reads using snapshots taken at an specifica moment.

Isolation levels

- **Per transaction**
- **Effect applies to read statements**: Set an isolation level forces the txn to check whether the **reads** conform the isolation level or not.
- **In the eye of the beholder**: The isolation level of one transaction won't affect other txn.

<u>Weaker "Isolation Levels"</u>

`Read Uncommitted`

`Read Committed`

`Repeatable Read`

<u>Strongest "Isolation Levels"</u>

`Serilizable order`

↓ Overhead ↑ Concurrency

↓ **Consistency Guarantees**

Isolation levels

1. **Read Uncommitted (Least Restrictive)**
   - Physically corrupt data is not read
   - Even uncommitted records may be read.
2. **Read Committed**
   - Will not allow reading of uncommitted data (**block** or old **snapshot**).
   - but successive reads of record may return different (but committed) values.
3. **Repeatable Read**
   - Only committed records to be read, repeated reads of same record must return same value.
   - However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
4. **Serializable (Most Restrictive)**
   - Prevents updates or appending of new rows until transaction is complete.

Anomalies vs Isolation levels

Level/Anomaly	Dirty Read	Unrepetable read	Phantom
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Read	No	No	Maybe
Serializable	No	No	No

Exercises

Consider table `W(name,pay)` and two concurrent txns. Remember individual sql statements S1, S2, S3, and S4 **always execute atomically**. Let Amy's pay be 50 before either txn begins execution.

T_1:
Begin Transaction
S1: update W set pay = 2*pay where name = 'Amy'
S2: update W set pay = 3*pay where name = 'Amy'
Commit

T_2:
Begin Transaction
S3: update W set pay = pay-20 where name = 'Amy'
S4: update W set pay = pay-10 where name = 'Amy'
Commit

Assume T_1 and T_2 execute to completion with different isolation levels. What are the possible values for Amy's final pay if

1. both execute with Serializable?
2. both execute with Read-Committed?.
3. T_1 is Read-Committed and T_2 executes with Read-Uncommitted?.
4. both execute with Read-Uncommitted?.
5. both execute with isolation level Serializable. T_1 executes to completion, but T_2 rolls back after statement S3 and does not re-execute.

Section 6

JDBC transactions management

JDBC and transactions

Transactions are **not explicitly opened**

- They start automatically right after a connection is created or
- inmediately after the end of another transaction.

The connection has a state called `AutoCommit` mode

- if true, then every statement is automatically committed
- if false, then every statement is added to an ongoing transaction and must be explicitly committed by using `connection.commit()` and `connection.rollback()`
- Default: true

Transactions are closed automatically, with an explicit commit or rollback, by any fail or when connection is closed.
DDL statements (e.g., creating/deleting tables) in a txn may be ignored or may cause a commit to occur

- The behavior is DBMS dependent

Connection interface for transaction management in JDBC

- **Sets isolation level for the current connection**

  `public int getTransactionIsolation()` and

  `void setTransactionIsolation(int level)`

- Specifies whether transactions in this connection are read-only

  `public boolean getReadOnly()` and

  `void setReadOnly(boolean b)`

- If autocommit is unset, then a transaction is committed using `connection.commit()` or aborted using `rollback()`.

Connection interface for transaction management in JDBC

- To check commit mode, use

```
public boolean getAutoCommit()
```

  and to change it, use

```
void setAutoCommit(boolean b)
```

- Checks whether connection is still open

```
public boolean isClosed()
```

Further Reading (highly recommended) (1)

[1] https://www.codemag.com/Article/0607081/Database-Concurrency-Conflicts-in-the-Real-World

[2] https://www.youtube.com/watch?v=FA85kHsJss4 and https://www.youtube.com/watch?v=aNCpEC0-VVs

[3] https://www.youtube.com/playlist?list=PLroEs25KGvwzmvIxYHRhoGTz9w8LeXek0 12-01, 12-02, 12-03

[4] https://cs.stanford.edu/ chrismre/

[5] https://cs.stanford.edu/people/widom/

[6] https://es.slideshare.net/brshristov/database-transactions-and-sql-server-concurrency

Further Reading (highly recommended) (2)

[7] https://ucbrise.github.io/cs262a-spring2018/notes/07-concurrency.pdf

[8] https://15445.courses.cs.cmu.edu/fall2018/