DB Transactions

TRANSACTION: A sequence of SQL statements that are executed "together" as one unit:

T1. **A money transfer transaction: <eg, Transfer $1M from Susan to Jane>**

S1: UPDATE Account SET balance = balance - 1000000
WHERE owner = 'Susan'

S2: Update Account SET balance = balance + 1000000
WHERE owner = 'Jane'

S1 and S2 are the actions composing the transaction T1.

The ACID Properties for Transactions

- **A**tomicity: "ALL-OR-NOTHING"
  - - Either ALL OR NONE of the operations in a transaction is executed.
  - - If the system crashes in the middle of a transaction, all changes
  - by the transaction are "undone" during recovery.
- **C**onsistency: If the database is in a consistent state (all integrity constraints are satisfied) before a transaction, the database is in a consistent state after the transaction
- **I**solation: Even if multiple transactions are executed concurrently, The result is the same as executing them in some sequential order.
  - Each transaction is unaware of (is isolated from) other transaction running concurrently in the system. But of course, overload might result in some slow-down.
- **D**urability If a transaction committed, all its changes remain permanently even after system crash

Transaction Properties

- Atomicity: all actions or none
- Concurrency: multiple transactions should be allowed to run in parallel while producing the same results as they run serially.
- Recovery from crash (or any situation where some actions cannot be completed):

In a transaction performing actions S1 and S2:
- Say that the system crashes after S1 but before S2. What now?
- Basically, we can keep a log of the actions, so we can **undo** S1.
  So, we are back in the situation we were before T1 was executed.
  Now the user can request that T1 and all uncompleted transactions should be re-executed.

Concurrent Executions

- Concurrent access from multiple client: We do not want to "lock out" the DBMS until one client finishes
- In general, advantages of letting multiple transactions allowed run concurrently in the system are:
  - **increased processor and disk utilization**,
  - **reduced average response time** for transactions
- But with concurrent executions transactions can interfere with each other and produce results that could not have produced if they were executed in any serial order.
- An execution order that produces the same results as a serial one is called a *serializable schedule*
- Examples:

Concurrency Examples

Example1: Concurrent with T1 (previous slide) we execute T2 that simply reports the current balances.

Example 2: <eg, Increase salary by $100 and then by $200>
T3: UPDATE Employee SET salary = salary + 1000
T4: UPDATE Employee SET salary = salary + 2000

Example 3: <eg, Increase salary by $100 and then by 20%>
T5: UPDATE Employee SET salary = salary + 1000
T6: UPDATE Employee SET salary = salary * 0.2

Example 3 has a concurrency problem. Example 2 does not---because addition commutes.

But, we do not want to bother with the semantics of the operations.

We want serializability criteria based only on read/write statements.

Serializable Schedules

We want to find schedule that are equivalent to serial schedules
- Independent of their actual computations
- According to their disk read/write action
  - ***Conflict Serializability***
- *This is achieved via various concurrency control schemes and protocols.*

Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed

Example Serial Schedules

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Example Concurrent Schedule

- Let T_1 and T_2 be the transactions defined previously.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Example Schedules (Cont.)

- The following concurrent schedule (Schedule 4 in the text) does not preserve the value of the the sum $A + B$.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Conflict Serializability

1: **read**(Q), *and* **read**(Q) *do not conflict*.
2: **read**(Q), *and* **write**(Q) from different transactions *conflict.*
3: **write**(Q), *and* **write**(Q) from different transactions *conflict.*

- **Conflict Graph for a set of transaction**: A node for each transaction in the schedule. Then draw directed arcs between transactions whose action conflicts
- **Serializable Schedule:** When its conflict graph has no directed cycles.

Conflict Serializability (Cont.)

- by series of swaps of non-conflicting instructions, the schedule on the left can be transformed into the one on the right, which is a serial schedule where T_2 follows T_1.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

✍ Thus the Schedule is conflict-serializable.

Sufficient vs. Necessary conditions

- Schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

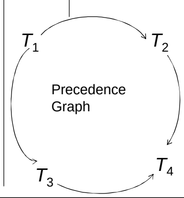
- Determining such equivalence requires analysis of operations other than read and write.

Example Schedule

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

Example Schedule:

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Concurrency Control Protocols vs. Serializability Tests

- **Testing a schedule for serializability *after* it has executed is a little too late!**
- Goal – to develop concurrency control protocols that will assure serializability.
  - We will study two: a locking protocol and a timestamp-based protocol
  - Our serializability test understand why a concurrency control protocol is correct.

Another Challenge: Recovery

- A transaction that has completed all its reads and writes will **commit**.
- A transaction that cannot complete all its reads and writes must abort (i.e. execute a rollback command)
- After commit a transaction cannot be rolled back.

Crash recovery--cont

But what does completion mean? Completion of computation in main memory means nothing, because that the system could crash soon after and ... everything will be lost.

Only secondary-store data are durable--we must focus on disk read/write.

T1. <Transfer $1M from Susan to Jane>

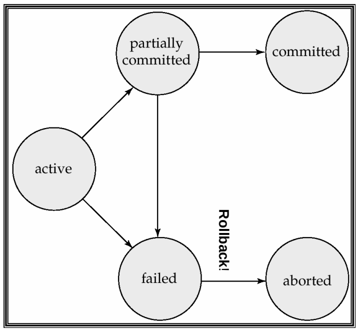
A1. read(balance) from Account WHERE owner = 'Susan'
A2. SET balance = balance - 1000000 %main memory
A3. write(balance) into Account WHERE owner = 'Susan'

A4. read(balance) from Account WHERE owner = 'Jane'
A5. SET balance = balance + 1000000 %main memory
A6. write(balance) into Account WHERE owner = 'Jane'

If we crash before A3, or after A6, nothing needs to be done.

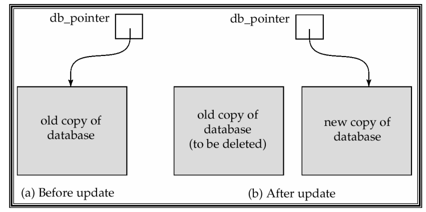
If we crash after A3 and before A6 we have to undo A3.

Transaction States (Cont.)



Implementation of Commit

The shadow-database scheme:



- Assumes disks to not fail
- Useful for text editors, but databases are too large.

Recoverability: Dirty Reads

- **Recoverable schedule** — if a transaction *Ti* reads a data items written by a transaction *Tj*, the commit operation of *Tj* must occur before the commit of *Ti.*
- The following schedule is not recoverable if T_9 commits immediately after the its dirty read (i.e., a read from a trans not yet committed)

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

- If T_8 should abort, T_9 could have shown to the user an inconsistent database state. Hence database must ensure that schedules are recoverable.

Recoverability (Cont.)

- **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

- If T_{10} fails, T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work

The ACID Properties

- **A**tomicity: "ALL-OR-NOTHING"
  - - Either ALL OR NONE of the operations in a transaction is executed.
  - - If the system crashes in the middle of a transaction, all changes
  - by the transaction are "undone" during recovery.
- **C**onsistency: If the database is in a consistent state (all integrity constraints are satisfied) before a transaction, the database is in a consistent state after the transaction
- **I**solation: Even if multiple transactions are executed concurrently, The result is the same as executing them in some sequential order.
  - Each transaction is unaware of (is isolated from) other transaction running concurrently in the system. But of course, overload might result in some slow-down.
- **D**urability If a transaction committed, all its changes remain permanently even after system crash

Transactions in SQL

- Explicit:
  - **begin transaction**
  - … update statements …
  - **end transaction**
- Implicit: each update statement is treated as a separate transaction
- Lower level of consistency
- User-controlled commit and rollbacks
- Vendor-dependent primitives and defaults

SQL: Levels of Consistency

- **Serializable** — default
- **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.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Lower degrees of consistency useful for gathering approximate information about the database, e.g., statistics for query optimizer.

e.g. SET TRANSACTION READ ONLY, REPEATABLE READ

AUTOCOMMIT mode OFF

- Transaction implicitly begins when any data in DB is read or written
- All subsequent read/write is considered to be part of the same transaction
- A transaction finishes when COMMIT or ROLLBACK statement is executed
  - COMMIT: All changes made by the transaction is stored permanently
  - ROLLBACK: Undo all changes made by the transaction

```
                             X                              X
------------------------------------------------------------------------
|         |        |         |          |        |         |       |
INSERT  DELETE  SELECT  COMMIT    DELETE  SELECT  ROLLBACK  INSERT
```

Setting Autocommit mode:

- In DB2: UPDATE COMMAND OPTIONS USING c ON/OFF (default is on)
- In Oracle: SET AUTOCOMMIT ON/OFF (default is off)
- In JDBC: connection.setAutoCommit(true/false) (default is on)

End