CS34800
Information Systems

Transactions

Walid Aref

Transaction

- Sequence of operations treated as a "single unit"
  - Either all happen, or none do
- Various syntaxes
  - SQL:1999 : **begin atomic** … **end**
  - Oracle: **set transaction** … **commit**
- Default in most DBMSs: each statement is a transaction

Oracle Syntax

- Starting a transaction:
  - commit; -- End previous transaction
  - set transaction; -- Start the new transaction
  - set constraint all deferred; -- Check at commit
  - <statements>
  - commit; -- End the transaction
- Can rollback instead of commit
  - As if the transaction never happened

Transactions

- Unit of work
- Atomic transaction
  - either fully executed or rolled back as if it never occurred
- Isolation from concurrent transactions
- Transactions begin implicitly
  - Ended by **commit work** or **rollback work**
- But default on most databases: each SQL statement commits automatically
  - Can turn off auto commit for a session (e.g. using API)
  - In SQL:1999, can use: **begin atomic** …. **end**
    - Not supported on most databases

Second goal of transactions: *Sequence* of Operations

- Update should complete entirely
  - update stipend set stipend = stipend*1.03;
  - What if it gets halfway and the machine crashes?
- What about multiple operations?
  - Withdraw x from Account1
  - ~~Deposit x into Account2~~
- Simultaneous operations?
  - Print paychecks while stipend being updated

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g. one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Example

- Consider two transactions:

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.01*A,	B=1.01*B	END

- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together.
- Assume A=100, B=100 at start. Result:
  - A. A = 202, B = 0
  - B. A = 201, B = 1
  - C. A = 202, B = 1
  - D. A = 201, B = 0

Example (Contd.)

- Consider a possible interleaving:

T1:	A=A+100, B=B-100	
T2:		A=1.01*A, B=1.01*B

- Assume A=100, B=100 at start. Result:
  - A. A = 202, B = 0
  - B. A = 201, B = 1
  - C. A = 202, B = 1
  - D. A = 201, B = 0

Example (Contd.)

- Consider a possible interleaving:

T1:		A=A+100, B=B-100
T2:	A=1.01*A, B=1.01*B	

- Assume A=100, B=100 at start. Result:
  A. A = 202, B = 0
  B. A = 201, B = 1
  C. A = 202, B = 1
  D. A = 201, B = 0

Example (Contd.)

- Consider a possible interleaving:

T1:	A=A+100,		B=B-100	
T2:		A=1.01*A,		B=1.01*B

- Assume A=100, B=100 at start. Result:
  A. A = 202, B = 0
  B. A = 201, B = 1
  C. A = 202, B = 1
  D. A = 201, B = 0

Example (Contd.)

- Consider a possible interleaving:

T1:	A=A+100,		B=B-100
T2:		A=1.01*A, B=1.01*B	

- Assume A=100, B=100 at start. Result:
  A. A = 202, B = 0
  B. A = 201, B = 1
  C. A = 202, B = 1
  D. A = 201, B = 0

Solution: *Transaction*

- Sequence of operations grouped into a transaction
  - Externally viewed as *Atomic*: All happens at once
  - DBMS manages so even the programmer gets this view
- Oracle: Requires additional argument
  - set transaction serializable

ACID properties

Transactions have:

- Atomicity
  - – All or nothing
- Consistency
  - – Changes to values maintain integrity
- Isolation
  - – Transaction occurs as if nothing else happening
- Durability
  - – Once completed, changes are permanent

Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
  – Because disk accesses are frequent, and relatively slow, it is important to keep the cpu humming by working on several user programs concurrently.
- A user's program may carry out many operations on the data retrieved from the database, but the DBMS is only concerned about what data is read/written from/to the database.
- A *transaction* is the DBMS's abstract view of a user program: a sequence of reads and writes.

Concurrency in a DBMS

- Users submit transactions, and can think of each transaction as executing by itself.
  - Concurrency is achieved by the DBMS, which interleaves actions (reads/writes of DB objects) of various transactions.
  - Each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
    - DBMS will enforce some ICs, depending on the ICs declared in CREATE TABLE statements.
    - Beyond this, the DBMS does not really understand the semantics of the data. (e.g., it does not understand how the interest on a bank account is computed).
- *Issues:* Effect of *interleaving* transactions, and *crashes*.

Atomicity of Transactions

- A transaction might *commit* after completing all its actions, or it could *abort* (or be aborted by the DBMS) after executing some actions.
- A very important property guaranteed by the DBMS for all transactions is that they are *atomic.* That is, a user can think of a Xact as always executing all its actions in one step, or not executing any actions at all.
  - DBMS *logs* all actions so that it can *undo* the actions of aborted transactions.

Scheduling Transactions

- *Serial schedule:* Schedule that does not interleave the actions of different transactions.
- *Equivalent schedules*: For any database state, the effect (on the set of objects in the database) of executing the first schedule is identical to the effect of executing the second schedule.
- *Serializable schedule*: A schedule that is equivalent to some serial execution of the transactions.

(If each transaction preserves consistency, every serializable schedule preserves consistency.)

Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, "dirty reads"):

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A), C	

- Unrepeatable Reads (RW Conflicts):

T1:	R(A),		R(A), W(A), C
T2:		R(A), W(A), C	

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

T1:	W(A),		W(B), C
T2:		W(A), W(B), C	

Example:

T1: Read(A)
$A \leftarrow A+100$
Write(A)
Read(B)
$B \leftarrow B+100$
Write(B)

T2: Read(A)
$A \leftarrow A \times 2$
Write(A)
Read(B)
$B \leftarrow B \times 2$
Write(B)

Constraint: A=B

Schedule A

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
Read(B); B ← B+100;			
Write(B);			125
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		250
		250	250

Schedule B

T1	T2	A	B
		25	25
	Read(A);A ← A×2;		
	Write(A);	50	
	Read(B);B ← B×2;		
	Write(B);		50
Read(A); A ← A+100			
Write(A);		150	
Read(B); B ← B+100;			
Write(B);			150
		150	150

Schedule C

T1	T2	A	B
		25	25
Read(A); A $\leftarrow$ A+100			
Write(A);		125	
	Read(A);A $\leftarrow$ A×2;		
	Write(A);	250	
Read(B); B $\leftarrow$ B+100;			
Write(B);			125
	Read(B);B $\leftarrow$ B×2;		
	Write(B);		250
		250	250

Schedule D

T1	T2	A	B
		25	25
Read(A); A ← A+100			
Write(A);		125	
	Read(A);A ← A×2;		
	Write(A);	250	
	Read(B);B ← B×2;		
	Write(B);		50
Read(B); B ← B+100;			
Write(B);			150
		250	150

Schedule E

Same as Schedule D but with new T2′

T1	T2’	A	B
		25	25
Read(A); A $\leftarrow$ A+100			
Write(A);		125	
	Read(A);A $\leftarrow$ A×1;		
	Write(A);	125	
	Read(B);B $\leftarrow$ B×1;		
	Write(B);		25
Read(B); B $\leftarrow$ B+100;			
Write(B);			125
		125	125

Deadlocks

- Deadlock: Cycle of transactions waiting for locks to be released by each other.
- Two ways of dealing with deadlocks:
  - Deadlock prevention
  - Deadlock detection

Dynamic Databases

- If we relax the assumption that the DB is a fixed collection of objects, even Strict 2PL will not assure serializability:
  - T1 locks all pages containing sailor records with *rating* = 1, and finds oldest sailor (say, *age* = 71).
  - Next, T2 inserts a new sailor; *rating* = 1, *age* = 96.
  - T2 also deletes oldest sailor with rating = 2 (and, say, *age* = 80), and commits.
  - T1 now locks all pages containing sailor records with *rating* = 2, and finds oldest (say, *age* = 63).
- No consistent DB state where T1 is “correct”!

The Problem

- T1 implicitly assumes that it has locked the set of all sailor records with *rating* = 1.
  - Assumption only holds if no sailor records are added while T1 is executing!
  - Need some mechanism to enforce this assumption. (Index locking and predicate locking.)
- Example shows that conflict serializability guarantees serializability only if the set of objects is fixed!

Logging and Recovery

- The following actions are recorded in the log:
  - *Ti writes an object*: the old value and the new value.
    - Log record must go to disk *before* the changed page!
  - *Ti commits/aborts*: a log record indicating this action.
- Log records are chained together by Xact id, so it's easy to undo a specific Xact.
- Log is often *duplexed* and *archived* on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, dealing with deadlocks etc.) are handled transparently by the DBMS.

Recovering From a Crash

There are 3 phases in the *Aries* recovery algorithm:

- *Analysis*: Scan the log forward (from the most recent *checkpoint*) to identify all Xacts that were active, and all dirty pages in the buffer pool at the time of the crash.
- *Redo*: Redoes all updates to dirty pages in the buffer pool, as needed, to ensure that all logged updates are in fact carried out and written to disk.
- *Undo*: The writes of all Xacts that were active at the crash are undone (by restoring the *before value* of the update, which is in the log record for the update), working backwards in the log. (Some care must be taken to handle the case of a crash occurring during the recovery process!)

Transaction Support in SQL-92

- Each transaction has an access mode, a diagnostics size, and an isolation level.

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No

Commit/Abort Decision

Each transaction ends with either:

1. *Commit* = the work of the transaction is installed in the database; previously its changes may be invisible to other transactions.
2. *Abort* = no changes by the transaction appear in the database; it is as if the transaction never occurred.
   - `ROLLBACK` is the term used in SQL and the Oracle system.

- In the ad-hoc query interface (*e.g.*, PostgreSQL psql interface), transactions are single queries or modification statements.
  - Oracle allows `SET TRANSACTION READ ONLY` to begin a multistatement transaction that doesn't change any data, but needs to see a consistent “snapshot” of the data.
- In program interfaces, transactions begin whenever the database is accessed, and end when either a `COMMIT` or `ROLLBACK` statement is executed.

SQL Isolation Levels

Isolation levels determine what a transaction is allowed to see. The declaration, valid for one transaction, is:

`SET TRANSACTION ISOLATION LEVEL` *X*`;`

where:

- *X* = `SERIALIZABLE`: this transaction must execute as if at a point in time, where all other transactions occurred either completely before or completely after.
  - Example: Suppose Sally's statements 1 and 2 are one transaction and Joe's statements 3 and 4 are another transaction. If Sally's transaction runs at isolation level SERIALIZABLE, she would see the `Sells` relation either before or after statements 3 and 4 ran, but not in the middle.

SQL Isolation Levels (Cont'd)

- *X* = `READ COMMITTED`: this transaction can read only committed data.
  - Example: if transactions are as above, Sally could see the original `Sells` for statement 1 and the completely changed `Sells` for statement 2.
- *X* = `REPEATABLE READ`: if a transaction reads data twice, then what it saw the first time, it will see the second time (it may see more the second time).
  - Moreover, all data read at any time must be committed; *i.e.*, `REPEATABLE READ` is a strictly stronger condition than `READ COMMITTED`.
  - Example: If 1 is executed before 3, then 2 must see the Bud and Miller tuples when it computes the min, even if it executes after 3. But if 1 executes between 3 and 4, then 2 may see the Heineken tuple.

SQL Isolation Levels (Cont'd)

- *X* = `READ UNCOMMITTED`: essentially no constraint, even on reading data written and then removed by a rollback.
  - Example: 1 and 2 could see Heineken, even if Joe rolled back his transaction.

Required Properties of a Transaction

- Transaction to transfer $50 from account A to account B:
  1. **read**(A)
  2. $A := A - 50$
  3. **write**(A)
  4. **read**(B)
  5. $B := B + 50$
  6. **write**(B)
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Required Properties of a Transaction (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction, when starting to execute, must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency

Required Properties of a Transaction (Cont.)

- **Isolation requirement** — if between steps 3 and 6 (of the fund transfer transaction) , another transaction **T2** is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	**T2**
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties

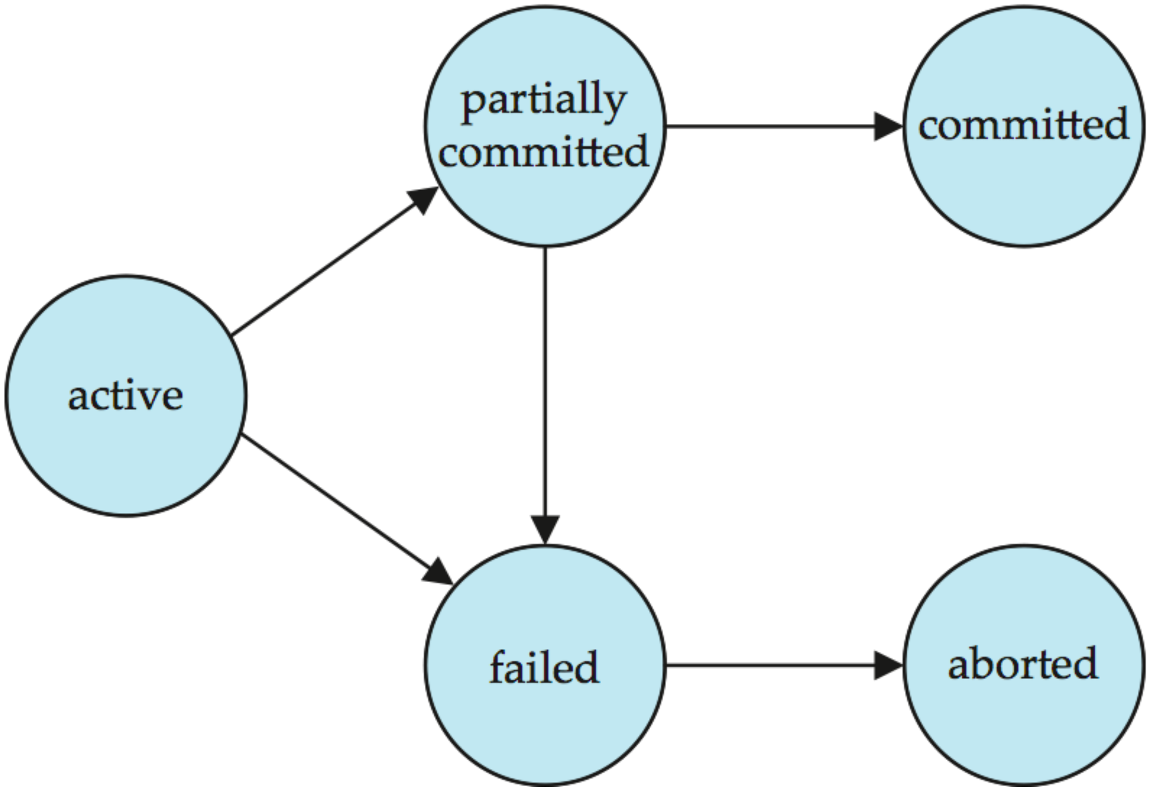
A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed --** after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - Restart the transaction
    - can be done only if no internal logical error
  - Kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



Additional Slides

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a **commit** instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an **abort** instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- An example of a **serial** schedule in which T_1 is followed by T_2 :

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

Schedule 2

- A **serial** schedule in which T_2 is followed by T_1 :

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

Schedule 3

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is **equivalent** to Schedule 1.

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) commit	
	read (B) $B := B + temp$ write (B) commit

Note -- In schedules 1, 2 and 3, the sum "A + B" is preserved.

Schedule 4

- The following concurrent schedule does not preserve the sum of "*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)	
commit	
	$B := B + temp$
	write (B)
	commit

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **conflict serializability**
  2. **view serializability**

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

- **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
- Warning: some database systems do not ensure serializable schedules by default
  - E.g., Oracle and PostgreSQL by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g. in JDBC, connection.setAutoCommit(false);