Transaction Processing

May 19, 2023

Transactions: basic definition

A transaction (“TXN”) is a collection of database operations that servers as a single, indivisible logical unit of work.

Example. A transaction that transfers 100 from account Alice to account Bob.

```
BEGIN;
  UPDATE account
  SET balance = balance - 100
  WHERE name = 'Alice';
  UPDATE account
  SET balance = balance + 100
  WHERE name = 'Bob';
COMMIT;
```

name	balance
Alice	200
Bob	200

Table: account(name, balance)

Transactions: basic definition

A transaction ("TXN") is a collection of database operations that servers as a single, indivisible logical unit of work.

Example. A transaction that transfers 100 from account Alice to account Bob.

```
BEGIN;
  UPDATE account
  SET balance = balance - 100
  WHERE name = 'Alice';
  UPDATE account
  SET balance = balance + 100
  WHERE name = 'Bob';
COMMIT;
```

- A TXN starts with the "BEGIN [TRANSACTION]" command.
- Followed by SQL operations that access/update the database.
- It stops with either "COMMIT" or "ABORT/ROLLBACK".

Transactions: basic definition

A transaction ("TXN") is a collection of database operations that servers as a single, indivisible logical unit of work.

Example. A transaction that transfers 100 from account Alice to account Bob.

```
BEGIN;
  UPDATE account
  SET balance = balance - 100
  WHERE name = 'Alice';
  UPDATE account
  SET balance = balance + 100
  WHERE name = 'Bob';
COMMIT;
```

- A TXN starts with the "BEGIN [TRANSACTION]" command.
- Followed by SQL operations that access/update the database.
- It stops with either "COMMIT" or "ABORT/ROLLBACK".

- COMMIT:make all the changes permanent and visible to other TXNs.
- ABORT/ROLLBACK:revert all the effects by the current TXN.

A simplified transaction model

- Database: A fixed set of named data objects, A, B, C.
- Transaction: A sequence of read and write operations, e.g., R(A), W(B).

```
BEGIN;
  UPDATE account
  SET balance = balance - 100
  WHERE name = 'Alice';
  UPDATE
  SET balance = balance + 100
  WHERE name = 'Bob';
COMMIT;
```

T_1
1. R(A)
2. A := A-100
3. W(A)
4. R(B)
5. B := B+100
6. W(B)

Transaction properties: ACID

- **A**tomicity: Each TXN is all-or-nothing, i.e., no partial TXN is allowed.
- **C**onsistency: Each TXN should leave the database in a consistent state.
- **I**solation: Each TXN is executed as if it were executed in isolation.
- **D**urability: Effects of a committed TXN are resilient against failures.

Atomicity

Each TXN is all-or-nothing, i.e., no partial TXN is allowed.

T_1
1. R(A)
2. A := A-100
3. W(A)
4. R(B)
5. B := B+100
6. W(B)

Q_1: What if after W(A) T_1 is aborted?

Q_2: What if after R(B), there is a power failure?

Consistency

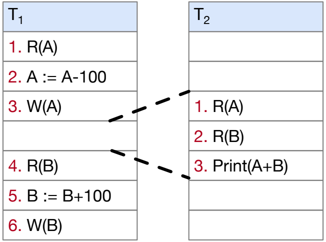
Each TXN should leave the database in a consistent state.

T_1
1. R(A)
2. A := A-100
3. W(A)
4. R(B)
5. B := B+100
6. W(B)

- If $A + B = 200$ before the execution of T_1, then $A + B = 200$ should still holds after T_1.
- Consistency is the programmer's burden.

Isolation

Each TXN is executed as if it were executed in isolation.



T_1	T_2
1. R(A)	
2. A := A-100	
3. W(A)	1. R(A)
	2. R(B)
4. R(B)	3. Print(A+B)
5. B := B+100	
6. W(B)	

- T_2 sees an inconsistent database, e.g., the printed value is smaller than 200.
- Isolation can be easily achieved by running transactions serially. Why not?
- The concurrency control manager allows interleaving executions of TXNs.

Isolation (cont'd)

Concurrent execution of transactions is essential for good DBMS performance.

- Improve throughput and resource utilization.
- Reduce average response time.

DBMS achieves concurrency by interleaving the operations of transactions.

The concurrency control manager of DBMS ensures that

- Operations of different transactions can be interleaved, and
- The interleaving execution of TXNs is equivalent to some serial execution.

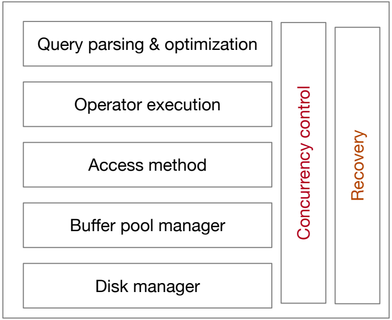
Durability

Effects of a committed TXN are resilient against failures.

T_1
1. R(A)
2. A := A-100
3. W(A)
4. R(B)
5. B := B+100
6. W(B)

- If DBMS crashes after T_1 committed successfully, e.g., the transfer has taken place, then all changes should be persistent and recoverable.
- DBMS handles durability (and atomicity) by its recovery manager.

Overview



- Concurrency control: ensure isolation in concurrent database access (this lecture).
- Recovery: ensure atomicity and durability via logging (next lecture).

Concurrency Control

A motivating example

- Assume that both accounts A and B have balance 200.
- TXN T_1: transfer 100 from account A to account B

$$T_1: \ \mathrm{R}(A), A := A - 100, W(A), \mathrm{R}(\mathrm{B}), \mathrm{B} := \mathrm{B} + 100, W(\mathrm{B})$$

- TXN T_2: Credits both A and B with 5% interest.

$$T_2: \ \mathrm{R}(A), A := A * 1.05, W(A), \mathrm{R}(\mathrm{B}), \mathrm{B} := \mathrm{B} * 1.05, W(\mathrm{B})$$

- Question: what are the possible outcomes of running T_1 and T_2?

Schedules

Time

T_1	T_2
R(A)	
A := A-100	
W(A)	
R(B)	
B := B+100	
W(B)	
	R(A)
	A := A*1.05
	W(A)
	R(B)
	B := B*1.05
	W(B)

- A serial schedule
- A = 105, B= 315

T_1	T_2
	R(A)
	A := A *1.05
	W(A)
	R(B)
	B := B *1.05
	W(B)
R(A)	
A := A - 100	
W(A)	
R(B)	
B := B + 100	
W(B)	

- Another serial schedule
- A = 110, B= 310

T_1	T_2
R(A)	
A := A-100	
W(A)	
	R(A)
	A := A *1.05
	W(A)
R(B)	
B := B + 100	
W(B)	
	R(B)
	B := B *1.05
	W(B)

- An interleaving schedule
- A = 105, B= 315

- A schedule specifies the chronological execution order for instructions of concurrent TXNs.
- A serial schedule executes transactions in order, with on interleaving of operations.

Good vs. bad schedule

T_1	T_2
R(A)	
A := A-100	
W(A)	
	R(A)
	A := A *1.05
	W(A)
R(B)	
B := B + 100	
W(B)	
	R(B)
	B := B *1.05
	W(B)

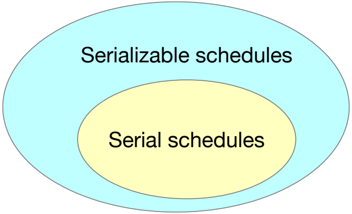
- A good interleaving schedule
- A = 105, B= 315

T_1	T_2
R(A)	
A := A-100	
W(A)	
	R(A)
	A := A *1.05
	W(A)
	R(B)
	B := B *1.05
	W(B)
R(B)	
B := B + 100	
W(B)	

- A bad interleaving schedule
- A = 105, B= 310

- If both T_1 and T_2 are submitted together, there is no guarantee that T_1 will be executed before T_2 or vice-verse.
- A good schedule requires that the effect must be equivalent to a serial one.
- A bad schedule has no equivalent serial counterpart.

Serializable schedules



- A schedule is serializable if it is equivalent to some serial schedule.
- Every serializable schedule preserves consistency if every TXN preservers consistency.
  - Serializability makes consistency reasoning easy.
- Serializable schedules allows more concurrency than serial schedules.

Simplified view of schedules

T_1	T_2
R(A)	
A := A-100	
W(A)	
	R(A)
	A := A *1.05
	W(A)
R(B)	
B := B + 100	
W(B)	
	R(B)
	B := B *1.05
	W(B)

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

A simplified view of schedule

- DBMS's abstract view of TXN: a TXN consists of only only read and write operations.
- TXN may perform arbitrary computations on data in local buffers in between reads and writes. DMBS cannot not "see" the operations other tan read and write instructions.
- We define simplified views of schedules along the same lines.

Conflicting operations

Two operations from different TXNs in a schedule conflict if

- access the same data item,
- and at least one operation is a write.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Problems with conflicting operations

- Read-Write conflicts (R-W)

  Dirty read: A TXN reads a value written by another TXN that has not yet committed.

- Write-Read conflicts (W-R)

  Unrepeatable read: A TXN read the same data item twice, while another TXN modified the data value between the two reads.

- Write-Write conflicts (W-W)

  Lost update: Two TXNs update the same data item. The second write overwrites the first update.

Serializability

- A schedule is “correct/good” if it is equivalent to some serial schedule.
- Given these conflicts, we need a way to check the correctness of schedules.

Definition

Tow schedules S and S' are conflict equivalent if

- S and S' are schedules of the same set of TXNs.
- Every pair of conflicting operations is ordered in the same way.

A schedule S is conflict serializable if S is conflict equivalent to some serial schedule.

Serializability (cont'd)

T_1	T_2
R(A)	
A := A-100	
W(A)	
R(B)	
B := B+100	
W(B)	
	R(A)
	A := A*1.05
	W(A)
	R(B)
	B := B*1.05
	W(B)

- A serial schedule S
- A = 105, B= 315

T_1	T_2
R(A)	
A := A-100	
W(A)	
	R(A)
	A := A *1.05
	W(A)
R(B)	
B := B + 100	
W(B)	
	R(B)
	B := B *1.05
	W(B)

- An interleaving schedule S'
- A = 105, B= 315

Serializability (cont'd)

T_1	T_2
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

A serial schedule S

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

An interleaving schedule S’

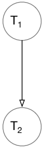
- S and S' are conflict equivalent.
- S' is a conflict serializable schedule.

Precedence graph

The precedence graph of a schedule S is a direct graph $G = (V, E)$, where

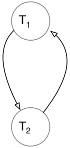
- Each node in V represents a TXN of S.
- Each edge in E represents a conflicts between two TXNs.
  - (T_i, T_j) in E indicates a pair of conflicting operation $O_i \in T_i$ and $O_j \in T_j$ such that O_i appears before O_j in S.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)



A serializable schedule

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	



A non-serializable schedule

Serializability test. A schedule is conflict serializable iff its precedence graph has no cycle.

Recap

- ACID probabilities of TXNs.
- Serializability: a desired property ensuring isolation.
- Reasoning about serializability via precedence graphs.

We next discuss how to generate schedules with the desired serializability properties.

Concurrency control approaches

Two-Phase Locking (2PL)

- A pessimistic approach: need to acquire a lock before every shared data access.
- The serializability order of conflicting operations is determined at runtime.

Timestamp ordering (T/O)

- An optimistic approach: (i) no locking, (ii) each TXN is assigned a unique timestamp before execution.
- Use the timestamps to determine the serializability order of TXNs.

Locking

	S	X
S	✓	✗
X	✗	✗

Table: Lock-compatibility matrix (✓: compatible, ✗:uncompatible)

A TXN T is allowed to access a data item A if and only if T holds a lock on A.

- Shared lock (S): (i) If T holds a shared lock on data A, then T can read but not write A. (ii) Multiple TXNs can hold the same shared lock.
- Exclusive lock (X): (i) If T holds an exclusive-mode lock on A, then T can both read and write A. (ii) At most one TXN can hold an exclusive lock on A.

Basic locking is not enough

- T1: $R(A); A := A - 50; W(A); R(B); B := B + 50; W(B);$
- T2: $R(A); R(B); Print(A + B);$

A=100; B=100

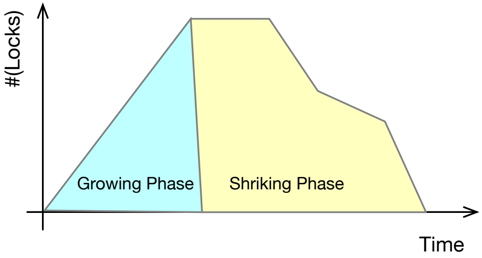
A+B = 150

T_1	T_2
Lock-X(A)	
R(A); A:= A-50	
W(A)	Lock-S(A)
Unlock(A)	
	R(A)
	Lock-S(B)
Lock-X(B)	R(B)
	print(A+B)
	Unlock(A)
	Unlock(B)
R(B); B:= B+50	
W(B)	
Unlock(B)	

TXN manager
Grant-X(T1, A)
Denied(T2,A)
Release-X(T1,A)
Grant-S(T2,A)
Grant-S(T2,B)
Denied(T1,B)
Release-S(T2,A)
Release-S(T2,B)
Grant-X(T1,B)
Release-X(T1,B)

This schedule generated via basic locking is not serilizable.

Two-Phase Locking (2PL)



In a transaction, all locks requests precede all unlock requests.

- Growing phase: (i) a TXN may obtain locks; (ii) a TXN may not release locks.
- Shrinking phase: (i) a TXN may release locks; (ii) a TXN may not obtain new locks.

We will show that 2PL guarantees conflict serializability.

Revised schedule with 2PL

T_1	T_2
Lock-X(A)	
R(A); A:= A-50	
W(A)	Lock-S(A)
Unlock(A)	
	R(A)
	Lock-S(B)
Lock-X(B)	R(B)
	print(A+B)
	Unlock(A)
	Unlock(B)
R(B); B:= B+50	T1
W(B)	T1
Unlock(B)	T1

Not 2PL: T1 released the lock on A before locking B.

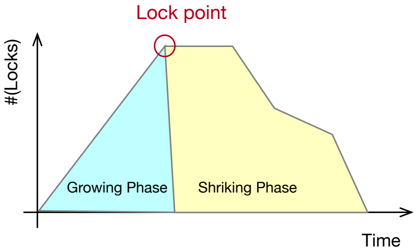
T_1	T_2
Lock-X(A)	
R(A); A:= A-50	
W(A)	Lock-S(A)
Lock-X(B)	
Unlock(A)	R(A)
	Lock-S(B)
R(B); B:= B+50	
W(B)	
Unlock (B)	
	R(B)
	print(A+B)
	Unlock(A)
	Unlock(B)

Lock point

Lock point

2PL: (T1, T2) is an equivalent serial schedule.

Why 2PL works



2PL warrants conflict serializability: a 2PL schedule is conflict equivalent to the serial scheduling obtained by ordering the TXNs according to their lock points.

Lemma 1

For every edge (T_i, T_j) in the precedence graph, it holds that $t_i < t_j$, where t_i and t_j are the lock points of T_i and T_j, respectively.

$\Longrightarrow$ No cycle in the precedence graph and the schedule is conflict serilizable.

Why 2PL works (cont'd)

Proof. Let O_i and O_j be a pair of conflict operations from T_i and T_j that induce the edge (T_i, T_j) in the precedence graph. We show that T_j cannot acquire the lock for O_j under 2PL until T_i release it. It follows that $t_i < t_j$. Let t_i' be point that T_i obtains the lock and t_i'' be the point that T_i releases it. Observe that following.

- T_j cannot obtain the lock for O_j before t_i'. Otherwise T_i cannot obtain the lock at t_i'.
- T_j cannot obtain the lock from time t_i' to time t_i'' since T_i holds the lock by 2PL.
- T_j can only obtain the lock after t_i''. By 2PL, $t_i < t_i'' < t_j$. □

Cascading aborts

If T_1 aborts here, then T_2 also needs to abort since T_2 read A written by T_1

T_1	T_2
Lock-X(A)	
R(A); A:= A-50	
W(A)	Lock-S(A)
Lock-X(B)	
Unlock(A)	R(A)
	Lock-S(B)
R(B); B:= B+50	
W(B)	
Unlock (B)	
	R(B)
	print(A+B)
	Unlock(A)
	Unlock(B)

- Cascading abort: A TXN T needs to abort if T read data written by an aborted TXN T′.
- Question. How to avoid cascading abort?

Strict Two-Phase Locking

T_1	T_2
Lock-X(A)	
R(A); A:= A-50	
W(A)	Lock-S(A)
Lock-X(B)	
R(B); B:= B+50	Lock-S(B)
W(B)	
Unlock(A)	
Unlock (B)	
	R(A)
	R(B)
	print(A+B)
	Unlock(A)
	Unlock(B)

If T_1 aborts here →

T_2 no longer needs to abort

- Strict 2PL: 2PL + “Release exclusive locks only after TXNs committed”.
- TXNs under strict 2PL never read other TXNs’ uncommitted data.
- Strict 2PL guarantees serializability and avoids cascading aborts.

Concurrency control approaches

Two-Phase Locking (2PL)

- A pessimistic approach: need to acquire a lock before every shared data access.
- The serializability order of conflicting operations is determined at runtime.

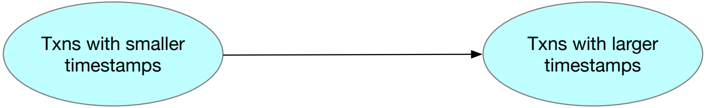
Timestamp ordering (T/O)

- An optimistic approach: (i) no locking, (ii) each TXN is assigned a unique timestamp before execution.
- Use the timestamps to determine the serializability order of TXNs.

Timestamps

- Each TXN T receives a unique timestamp $\mathrm{TS}(\mathrm{T})$.
- Each data item A is associated with two timestamps:
  - $W\text{—}\mathrm{TS}(A)$: the largest $\mathrm{TS}(\mathrm{T})$ of any txn T that wrote A successfully.
  - $\mathrm{R}\text{—}\mathrm{TS}(A)$: the large $\mathrm{TS}(\mathrm{T})$ of any txn T that read A successfully.
- The timestamps can be obtained by either the system's clock or a logical counter.
- $W\text{—}\mathrm{TS}(A)$ and $\mathrm{R}\text{—}\mathrm{TS}(A)$ are updated whenever a $W(A)$ or a $\mathrm{R}(A)$ executes successfully.

A timestamp-ordering (T/O) protocol



- The timestamp order induces a serial order of scheduled TXNs.
- The protocol ensures that conflicting operations are processed in the timestamp order.
- $\mathrm{TS}(\mathrm{T}) < \mathrm{TS}(\mathrm{T}')$ indicates that in an equivalent serial schedule T must appear before T'.

For each $\mathrm{R}(A)$ and $W(A)$ request issued by a TXN T, the scheduler checks operation conflicts.

- Let T proceed if conflicting operations follows the timestamp order.
- Otherwise, abort and restart T with a newer timestamp.

A timestamp-ordering (T/O) protocol

Read rule: T issues $\mathrm{R}(A)$

- If $\mathrm{TS}(\mathrm{T}) < W\text{—}\mathrm{TS}(A)$, this violates the timestamp ordering of T w.r.t. a txn that wrote A. Then abort and restart T.
- Otherwise, execute $\mathrm{R}(A)$ of T and update $\mathrm{R}\text{—}\mathrm{TS}(A)$ to $\max\{\mathrm{R}\text{—}\mathrm{TS}(A), \mathrm{TS}(\mathrm{T})\}$.

Write rule: T issues $W(A)$

- If $\mathrm{TS}(\mathrm{T}) < \mathrm{R}\text{—}\mathrm{TS}(A)$ or $\mathrm{TS}(\mathrm{T}) < W\text{—}\mathrm{TS}(A)$, then abort and restart T.
- Otherwise, allow T to write A and update $W\text{—}\mathrm{TS}(A)$ to $\max\{W\text{—}\mathrm{TS}(A), \mathrm{TS}(\mathrm{T})\}$.

Lemma. The T/O protocol guarantees conflict serializability.

An example schedule with T/O protocol

- T_1: R(A), W(A), R(B), W(B)
- T_2: R(A), W(A), R(B), W(B)
- $TS(T_1) = 1$, $TS(T_2) = 2$

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Table: A possible schedule with T/O