Transaction Management Overview

Chapter 16

Basic Architecture

- Application Layer - what most users see, talks SQL
- Parsing/Planning Layers
- Runtime or execution Layer
- Storage Layer - where data resides, may include simple access layer

Applications

Parsing

Planning

Processing

Data Access

Data in SSD/HDD

ACID

- Atomicity
    - All or nothing - transactions
- Consistency
    - Take DBMS from one consistent state to another
- Isolation
    - Each transaction is not impacted by others
- Durability
    - A committed transaction’s changes are “durable”

Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
  - Hides waits for disk I/O by keeping several transaction going at once
  - Exploit multi-core architecture
- DBMS is only concerned about what data is read/ written from/to the database during the execution of a user program
- 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.
  - Transactions are required to leave DBMS in a consistent state, i.e. each transaction must leave the database in a consistent state if the DB is consistent when the transaction begins.
- *Issues:* Effect of *interleaving* transactions, and *crashes*.

Atomicity of Transactions

- 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.

Example

- Consider two transactions (*Xacts*):

T1:	BEGIN	A=A+100,	B=B-100	END
T2:	BEGIN	A=1.06*A,	B=1.06*B	END

- Intuitively, the first transaction is transferring $100 from B's account to A's account. The second is crediting both accounts with a 6% interest payment.
- There is no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together. However, the net effect *must* be equivalent to these two transactions running serially in some order.

Two Transactions

T1

1. Read(A)
2. A = A + 100
3. Write(A)
4. Read(B)
5. B = B - 100
6. Write(B)

T2 A = 1000, B = 1000

1. Read(A)
2. A = A*1.05
3. Write(A)
4. Read(B)
5. B = B*1.05
6. Write(B)

- Normally we that either T1 or T2 goes first
- Then the other
- So either
- T1 then T2: A = 1155, B = 945
- T2 then T1: A = 1150, B = 950

1. T1 Read(A)
2. T2 Read(A)
3. T2 A = A*1.05
4. T1 A = A + 100
5. T1 Write(A)
6. T2 Write(A)
7. T2 Read(B)
8. T2 B = B*1.05
9. T2 Write(B)
10. T1 Read(B)
11. T1 B = B - 100
12. T1 Write(B)

A = 1100
B = 950

1. T1 Read(A)
2. T2 Read(A)
3. T2 A = A*1.05
4. T1 A = A + 100
5. T2 Write(A)
6. T1 Write(A)
7. T1 Read(B)
8. T1 B = B - 100
9. T1 Write(B)
10. T2 Read(B)
11. T2 B = B*1.05
12. T2 Write(B)

A = 1050
B = 945

Example (Contd.)

- Consider a possible interleaving (*schedule*):

T1:	A=A+100,		B=B-100	
T2:		A=1.06*A,		B=1.06*B

- This is OK. But what about:

T1:	A=A+100,		B=B-100
T2:		A=1.06*A, B=1.06*B	

- The DBMS's view of the second schedule:

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

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.

(Note: 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	

Two Transactions

T1

1. Lock(X,A)
2. Read(A)
3. A = A + 100
4. Write(A)
5. Lock(X, B)
6. Unlock(A)
7. Read(B)
8. B = B - 100
9. Write(B)
10. Unlock(B)

T2 A = 1000, B = 1000

1. Lock(X,A)
2. Read(A)
3. A = A*1.05
4. Write(A)
5. Lock(X,B)
6. Read(B)
7. B = B*1.05
8. Write(B)
9. Unlock(A)
10. Unlock(B)

- T2 then T1 strictly
- T1 (1-6), T2(1-4), T2(5,10)
- T1(7-10)

T1 starts details

1. T1 Lock(X,A)
2. T1 Read(A)
3. T1 A = A + 100
4. T1 Write(A)
5. T1 Lock(X,B)
6. T1 Unlock(A), T2 Lock(X,A)
7. T1 Read)B), T2 Read(A)
8. T1 B = B - 100, T2 A = A*1.05
9. T1 Write(B), T2 Write(A)
10. T1 Unlock(B), T2 Lock(X,B)
11. T2 Read(B)
12. T2 B = B * 1.05
13. T2 Write(B)
14. T2 Unlock(A)
15. T2 Unlock(B)

Lock-Based Concurrency Control

- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
  - All locks held by a transaction are released when the transaction completes (i.e. at commit/abort).
    - (Non-strict) 2PL Variant: Release locks anytime, but cannot acquire locks after releasing any lock.
  - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
  - (Non-strict) 2PL also allows only serializable schedules, but involves aborts that may have to be cascaded to transactions that read the data modified by this now-aborted Xact

Aborting a Transaction

- If a transaction *Ti* is aborted, all its actions have to be undone. Not only that, if *Tj* reads an object last written by *Ti*, *Tj* must be aborted as well (possible with non-strict 2PL)!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time (strict 2PL).
  - If *Ti* writes an object, *Tj* can read this only after *Ti* commits.
- In order to *undo* the actions of an aborted transaction, the DBMS maintains a *log* in which every write is recorded. This mechanism is also used to recover from system crashes: all active Xacts at the time of the crash are aborted when the system comes back up.

The Log

- 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.

Summary

- Concurrency control and recovery are among the most important functions provided by a DBMS.
- Users need not worry about concurrency.
  - System automatically inserts lock/unlock requests and schedules actions of different Xacts in such a way as to ensure that the resulting execution is equivalent to executing the Xacts one after the other in some order.
- Write-ahead logging (WAL) is used to undo the actions of aborted transactions and to restore the system to a consistent state after a crash.
  - *Consistent state*: Only the effects of committed Xacts seen.