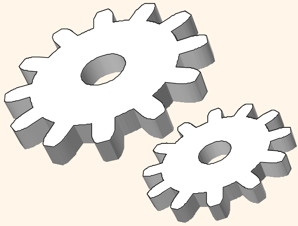
Transaction Management Overview

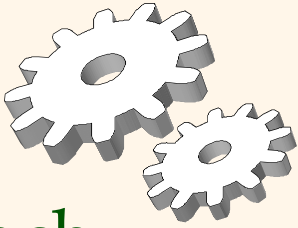
Chapter 16

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*.

ACID Properties

- *Atomicity*. A transaction is executed all or none. Users should not worry about incomplete transactions.
- *Consistency*. Each transaction should leave the database in a consistent state.
- *Isolation*. The execution of a transaction is isolated (protected) from the effects of other concurrent transactions
- *Durability*. Once the DBMS informs the user that the transactions is committed, its effect should persist even if the system 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.

Consistency of Transactions

- A DBMS is responsible for ensuring the *consistency* according to the predetermined constraints defined in the CREATE TABLE and CREATE ASSERTION statements

- However, the user is responsible for ensuring the semantic of the transaction

- If a user is doing something inconsistent, there is no way that the DBMS catches it

Isolation of Transactions

- Even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other

- For example, if two transactions T_1 and T_2 are executed concurrently, the net effect is guaranteed to be equivalent to executing (all of) T_1 followed by (all of T_2) OR equivalent to executing T_2 followed by T_1

Durability of Transactions

- A committed transaction guarantees that its effect will survive permanently and will not be undone later (without the user acknowledgment)

- Should take care of crashing effects before or after transaction effects go through the disk

Transaction Representation

- A transaction is represented as
  - A list of actions (*Reads* and *Writes*)
  - A final state (*Commit* or *Abort*)

- The following transaction is R(A), W(A), Commit

```
UPDATE Students S
SET S.age = S.age +1, S.gpa = S.gpa -1
WHERE S.sid = 54832
```

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.

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

- Write-Read Conflict (WR)
  - T2 reads a data written by T1
  - Reading uncommitted data
- Read-Write Conflict (RW)
  - T2 writes a data read by T1
  - Unrepeatable read
- Write-Write Conflict (WW)
  - T2 overwrites a data written by T1
  - Overwriting uncommitted data

WR Conflict: Reading Uncommitted Data

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

- A transaction may read a certain data that are not committed yet, yielding erroneous results
- *Dirty Read*

RW Conflict: Reading Uncommitted Data

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

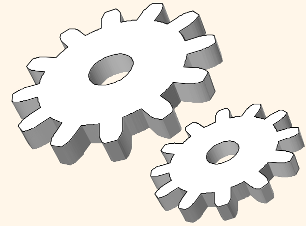
- *Unrepeatable Reads*

WW Conflict: Overwriting Uncommitted Data

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

- *Lost updates*

Lock-Based Concurrency Control



- *Strict Two-phase Locking (Strict 2PL) Protocol*:
  - *Rule 1:* Each Xact must obtain a S (*shared*) lock on object before reading, and an X (*exclusive*) lock on object before writing.
    - If an Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object.
  - *Rule 2:*All locks held by a transaction are released when the transaction completes
- Strict 2PL allows only serializable schedules.
  - Additionally, it simplifies transaction aborts
- *Deadlocks?* How to detect and resolve

Deadlocks

- Scenario: Transaction T_1 sets an exclusive lock on object A, T_2 sets an exclusive lock on B, T_1 requests an exclusive lock on B and is queued, and T_2 requests an exclusive lock on A and is queued.
  - T_1 is waiting for T_2 to release its lock ad vice versa.
- Transactions that involve in a deadlock cycle:
  - Make no further progress
  - They hold locks that may be required by other transactions.
- Some techniques are available for:
  - Deadlock avoidance
  - Deadlock detection and resolving (most common)

Aborting a Transaction

- If a transaction T_i is aborted, all its actions have to be undone. Not only that, if T_j reads an object last written by T_i, T_j must be aborted as well!
- Most systems avoid such *cascading aborts* by releasing a transaction's locks only at commit time.
  - If T_i writes an object, T_j 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.

Recovery Manager

- The recovery manager of a DBMS is responsible for ensuring transaction:
  - *Atomicity:* By undoing the actions of aborted transactions
  - *Durability:* By ensuring that all actions of committed transactions made their way to permanent storage.

- When a DBMS is restarted after crashes, the recovery manager is given control as it must bring the databse to a consistent state

- For now, we assume "atomic writes"

Stealing Frames and Forcing Pages

- *Steal:* Changes made by transaction T may be written to disk even before T commits. This could happen if another transaction T1 wants to bring a page into memory and the buffer manager chooses to replace (steal) the frame modified by T.
- *Force:* When a transaction commits, all modified pages are forced to disk.
- If *no-steal* approach is used:
  - We do not have to undo the changes of an aborted transaction
- If a *force* approach is used:
  - We do not have to redo the changes of a committed transaction
- State-of-the-art recovery managers use a *steal no-force* approach

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.

Recovering From a Crash

- There are 3 phases in a 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!)

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 commited Xacts seen.