Transaction Management Overview

Ramakrishnan & Gehrke, Chapter 14+

Transactions

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

Concurrency in a DBMS

JACOBS UNIVERSITY

- Users submit TAs, and can think of each transaction as executing by itself
  - Concurrency achieved by DBMS,
    which interleaves actions (reads/writes of DB objects) of various TAs
  - Each TA must leave the database in a consistent state
    if the DB is consistent when TA 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*
    - *Ex: does not understand how the interest on a bank account is computed*
- Issues:
  - Effect of interleaving TAs
  - crashes

Atomicity of Transactions

- Two possible TA endings:
  - commit after completing all its actions – data must be safe in DB
  - abort (by application or DBMS) – must restore original state
- Important property guaranteed by the DBMS: TAs atomic
  - Perception: TA executes all its actions in one step, or none
- Technically: DBMS logs all actions
  - can undo actions of aborted TAs
  - Write-ahead logging (WAL): save record of action before every update

Transaction Syntax in SQL

- START TRANSACTION start TA
- COMMIT end TA successfully
- ROLLBACK abort TA (undo any changes)
- If none of these TA management commands is present, each statement starts and ends its own TA
  - including all triggers, constraints,…

Anatomy of Conflicts

- Consider two TAs:

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

  - Intuitively, first TA transfers $100 from B's account to A's account
  - second TA credits both accounts with a 6% interest payment

- no guarantee that T1 will execute before T2 or vice-versa, if both are submitted together
- However, net effect must be equivalent to these two TAs running serially in some order

Scheduling Transactions: Definitions

- **Serial schedule**:
  Schedule that does not interleave the actions of different TAs
- **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 equivalent to some serial execution of the TAs
- each TA preserves consistency
  $\Rightarrow$ every serializable schedule preserves consistency

Lock-Based Concurrency Control

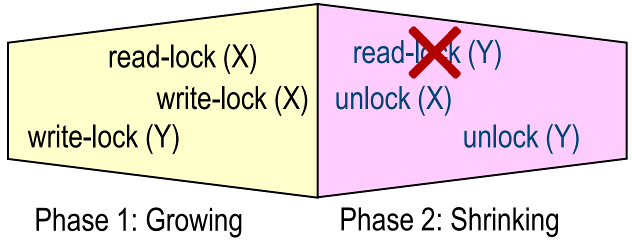
- Core issues: What lock modes? What lock conflict handling policy?
- Common lock modes: SX
  - Each TA must obtain an S (shared) lock before reading, and an X (exclusive) lock before writing

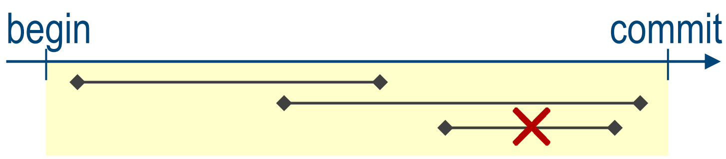
```
  | S X
--+------
S | + -
X | - -
```

- Lock conflict handling
  - Abort conflicting TA / let it wait / work on previous version
- Locking protocols
  - two-phase locking (strict, non-strict, conservative, …) – *next!*
  - Timestamp based
  - Multi-version based
  - Optimistic concurrency control

Two-Phase Locking Protocol

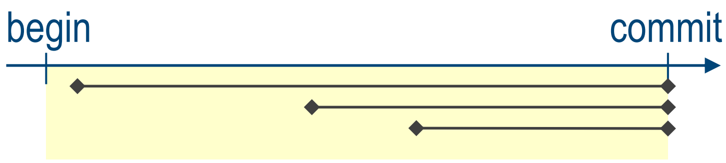
- 2PL
  - All locks acquired before first release (=all locks released after last acquiring)
  - cannot acquire locks after releasing first lock





- allows only serializable schedules ☺
  - but complex abort processing

- Strict 2PL
  - All locks released when TA completes



- Strict 2PL simplifies TA aborts ☺☺

Isolation Levels

- Isolation level directives: summary about TA's intentions, placed before TA
  - SET TRANSACTION READ ONLY
    TA will not write → can be interleaved with other read-only TAs
  - SET TRANSACTION READ WRITE
    (default)
- assists DBMS optimizer
- Example: Choosing seats in airplane
  - *Find available seat, reserve by setting occ to TRUE; if there is none, abort*
  - *Ask customer for approval. If so, commit, otherwise release seat by setting occ to FALSE, goto 1*
  - two "TA"s concurrently: can have dirty reads for occ – uncritical! (why?)

Effects of New Isolation Levels

- Consider seat choosing algorithm:
- If run at level READ COMMITTED
  - seat choice function will not see seats as booked
    if reserved but not committed (roll back if over-booked)
  - Repeated queries may yield different seats (other TAs booking in parallel)
- If run at REPEATABLE READ
  - any seat found in step 1 will remain available in subsequent queries
  - new tuples entering relation (e.g. switching flight to larger plane) seen by new queries

Summary

- Concurrency control & recovery: core DBMS functions
- Users need not worry about concurrency
  - System automatically inserts lock/unlocking,
    schedules TAs, ensures serializability (or what's requested)
- ACID properties!
- Mechanisms:
  - TA scheduling; Strict 2PL !
  - Locks
  - Write-ahead logging (WAL)

Outlook: ACID vs BASE

- BASE (Basically Available Soft-state Eventual Consistency)
  - Prefers availability over consistency
  - Relaxing ACID
- CAP Theorem [proposed: Eric Brewer; proven: Gilbert & Lynch]:
  In a distributed system you can satisfy at most 2 out of the 3 guarantees
    - *Consistency: all nodes have same data at any time*
    - *Availability: system allows operations all the time*
    - *Partition-tolerance: system continues to work in spite of network partitions*
- Comparison:
  - Traditional RDBMSs: Strong consistency over availability under a partition
  - Cassandra: Eventual (weak) consistency, availability, partition-tolerance