Introduction to Data Management

Lecture #18
(Transactions: The Final Frontier...!)

Instructor: Mike Carey
mjcarey@ics.uci.edu

Announcements

- HW and Endterm info:
  - HW#8 now in flight! (Due tomorrow; -10pts if late)
  - Endterm Exam: **Mon, June 12, 1:30-3:30 PM**
    - Cheat sheet allowed
    - Non-cumulative (see wiki for official scope!)
    - Sample exams available (interpret appropriately!)
    - Will include NoSQL, JSON, and even transactions!

- This week’s material:
  - JSON mini-course (last time) and quiz
  - Today: Transactions (and maybe Endterm Q&A)
Transactions

- Concurrent execution of user programs is essential for good DBMS performance.
  - Because disk accesses are frequent, and relatively slow, it’s important to keep the CPU cores humming by working on several user programs concurrently.
- A user’s program may carry out many operations on the data pulled 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 (record) reads and writes.

The ACID Properties

- **Atomicity**: Each transaction is all or nothing.
  - No worries about partial effects (if failures) and cleanup.
- **Consistency**: Each transaction moves the database from one consistent state to another one.
  - This is largely the application builder’s responsibility.
- **Isolation**: Each transaction can be written as if it’s the only transaction in existence.
  - No concurrency worries while building applications.
- **Durability**: Once a transaction has committed, its effects will not be lost.
  - Application code doesn’t have to worry about data loss.
Concurrency in a DBMS

- Users submit transactions, and they can think of each one as executing all 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 may enforce some ICs, depending on the ICs declared in CREATE TABLE statements. (CHECK, PK/FK, ...)
    - Beyond this, the DBMS does not understand the semantics of the data. (E.g., it doesn’t know 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.
  - Could violate some constraint, encounter some other error, be caught in a crash, or be picked to resolve a deadlock.
- 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.
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 bank account A to bank account B. The second is crediting both accounts with a 6% interest payment.
- 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 transactions running serially in some (but either!) order.

A Quick Aside on “A” & “B”

- What are these two transactions, really?

```
T1: BEGIN
   UPDATE Acct SET bal = bal + 100 WHERE acct_no = 101;
   UPDATE Acct SET bal = bal - 100 WHERE acct_no = 201;
END

T2: BEGIN
   UPDATE Acct SET bal = bal * 1.06 WHERE acct_type = ‘SV’;
   END
```

- Again, the first transaction is transferring $100 from account B (201) to account A (101). The second one is giving all savings accounts their 6% interest payment.
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 happens if:

  T1: A=A+100, B=B-100
  T2: A=1.06*A, B=1.06*B ← Too much interest!

- 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 (Defn’s.)

- **Serial schedule**: Any 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 (any!) serial execution of the transactions.

  *(Note: If each transaction preserves consistency, then every serializable schedule preserves consistency!)*
Anomalies with Interleaved Execution

- Reading Uncommitted Data (WR Conflicts, a.k.a. “dirty reads”):

  T3: \(R(A), W(A), R(B), W(B), \text{Abort}\)
  T4: \(R(A), W(A), C\)

- Unrepeatable Reads (RW Conflicts):

  T5: \(R(A), R(A), W(A), C\)
  T6: \(R(A), W(A), C\)

Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

  T7: \(W(A), W(B), C\)
  T8: \(W(A), W(B), C\)

(Note how results are a “must have been concurrent!” intermingling of transactions’ T1 & T2 writes…)
Lock-Based Concurrency Control

- **Strict Two-phase Locking (Strict 2PL) Protocol**:
  - Each Xact must get an *S* (shared) lock on an object before reading, and an *X* (exclusive) lock on it before writing.
  - All locks held by a transaction are released only when the transaction completes.
    - (Non-strict) 2PL Variant: Release locks anytime, but do not acquire any new locks after releasing any lock.
  - *Note*: If a Xact holds an X lock on an object, no other Xact can get a lock (S or X) on that object – they must wait.

- **Strict 2PL allows only serializable schedules**.
  - And additionally, it simplifies transaction aborts!
  - (Non-strict) 2PL also allows only serializable schedules, but needs more complex abort processing (as you’ll see).

2PL Prevents the Anomalies

- **Reading Uncommitted Data (WR Conflicts, a.k.a. “dirty reads”):**

  - **T3:** R(A), W(A), R(B), W(B), Abort
  - **T4:** R(A), W(A), C

- **Unrepeatable Reads (RW Conflicts):**

  - **T5:** R(A), R(A), R(A), W(A), C
  - **T6:** R(A), W(A), C
2PL & Anomalies (Continued)

- Overwriting Uncommitted Data (WW Conflicts):

```
T7: W(A), W(B), C
T8: W(A), W(B), C
```

(Now results will no longer be a “must have been concurrent!” intermingling of T1’s & T2’s writes…)

Aborting a Transaction

- If transaction Ti aborts, all its actions must be undone.
  - And, if some Tj already read a value last written by Ti, Tj must also be aborted! (“If I tell you, I’ll have to kill you…” 😒)
- Most systems avoid such cascading aborts by releasing a transaction’s locks only at commit time.
  - If Ti writes an object, Tj can read it only after Ti commits.
- In order to undo the actions of an aborted transaction, the DBMS keeps a log where every write is recorded.
  - Also used to recover from system crashes: active Xacts at crash time are aborted when the DBMS comes back up.
The Transaction Log

- The following actions are recorded in the log:
  - **Ti writes an object:** record the object’s old and new values.
    - Log record must go to disk before the changed page does – hence the name write-ahead logging (or WAL).
  - **Ti commits/aborts:** write a log record noting this outcome.
- Log records are back-chained together by Xact id, so it’s easy to undo a specific Xact if need be.
- Log is often **duplexed and archived** on stable storage.
- All log related activities (and in fact, all CC related activities such as lock/unlock, handling deadlocks etc.) are **transparently** taken care of by the DBMS (!!).

Reminder: Disks and Files

- DBMSs store all information on disk.
- This has major implications for DBMS design!
  - **READ:** transfer data from disk to main memory (RAM).
  - **WRITE:** transfer data from RAM to disk.
  - Both are high-cost operations, relative to in-memory operations, so must be considered carefully!
Recovering From a Crash

- A three-phase recovery algorithm (Aries):
  - **Analysis**: Scan log forward (from most recent checkpoint) to identify all Xacts that were active, and also all dirty pages in the buffer pool, as of the time of the crash.
  - **Redo**: Redo 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. (Establishes the state to recover from.)
  - **Undo**: Undo writes of all Xacts that were active at the crash (restore the before value of each update, which is in the log record for the update), working backwards through the log. *(Note: Care must be taken to account for the possibility of a subsequent crash during this recovery process...!)*

Support for Transactions in SQL-92

- A transaction is automatically started whenever a statement accesses or modifies the database
  - SELECT, UPDATE, CREATE TABLE, INSERT, ...
  - Multi-statement transactions also supported
- A transaction can be terminated by
  - A COMMIT statement
  - A ROLLBACK statement (SQL-speak for abort)
- Each transaction runs under a combination of
  - An access mode
  - An isolation level
Transactions in SQL-92 (Cont’d.)

- Access mode – controls what the transaction can potentially do to the database:
  - READ ONLY: not permitted to modify the DB
  - READ WRITE (default): allowed to modify the DB

- Isolation level – controls the transaction’s exposure to other (concurrent) transactions:
  - READ UNCOMMITTED
  - READ COMMITTED
  - REPEATABLE READ
  - SERIALIZABLE

Increasing isolation

Concurrency/Consistency Issues

- Hey, what could possibly go wrong...? (☺)
  - Dirty Read (WR conflict): a transaction could read an object written by an uncommitted transaction.
  - Unrepeatable Read (RW conflict): a transaction could overwrite an existing object that was read by an uncommitted transaction.
  - Phantom: a transaction reads a collection of objects twice, and sees some different (newly inserted) objects on the second time through.
Isolation Levels

<table>
<thead>
<tr>
<th>Isolation Level</th>
<th>Dirty Read</th>
<th>Unrepeatable Read</th>
<th>Phantom</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ UNCOMMITTED</td>
<td>Maybe</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>READ COMMITTED</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
</tr>
<tr>
<td>REPEATABLE READ</td>
<td>No</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>SERIALIZABLE</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

An application-“controllable” tradeoff:
- Consistency vs. performance (concurrency)
- Note that this will affect your programming model

Things to watch out for:
- Default consistency level is DBMS engine-specific
- Some engines may not support all levels
- Default consistency level often not SERIALIZABLE

You may also hear about “snapshot isolation”
- DBMS keeps multiple versions of data, transactions see data versions as of their start timestamp
Remember the **ACID** Properties!

- **Atomicity**: Each transaction is *all or nothing*.
  - No worries about partial effects (if failures) and cleanup.
- **Consistency**: Each transaction moves the database from one *consistent state* to another one.
  - This is largely the application builder’s responsibility.
- **Isolation**: Each transaction can be written as if it’s the *only transaction* in existence *(if so desired)*.
  - Minimize concurrency worries when building applications.
- **Durability**: Once a transaction has committed, its *effects will not be lost*.
  - Application code doesn’t have to worry about data loss.

For transactions, NoSQL systems tend to be limited to *record-level* transactions (in order to *scale* on a cluster). As a result, one might consider transactional desires when designing a schema (e.g., what to nest). In general, my “rules” for NoSQL schema design are:
- Start with an E-R model – you’re still DB professionals!
- Strong entities: keep as “top-level objects” in most cases
- Weak entities: probably* nested within their parent object
- Relationships: all relational options still available, *plus* you could have a set of keys if you wanted (e.g., dept.emps)
- 1NF: composite and set-valued attributes are fair game
- Size: aim to keep object sizes reasonable and also *“fixed”*
v CS122a has just given you an “outside” view of database management systems.
v CS122b is available to give you a “programmer’s” view – with an emphasis on data-centric web applications.
v CS122c (a.k.a. CS222 lite) is available to give you an “insider’s” (engine developer’s) view of database systems.
v CS223 is available for learning all about transactions.
v CS190 (if/when offered) and CS199 (independent project work) are also possible avenues for gaining further info.
  ▪ I’m planning a CS190 offering in Winter ’18 that’ll be called “Beyond SQL Data Management” (→ CS122d, eventually?)
  ▪ MongoDB, AsterixDB, Couchbase, Cassandra, Neo4j, Hadoop, Spark, ...