CS162 Discussion #08: Databases & Transactions

Announcements

- Project 2 initial design due TODAY, 10/18 @5:00pm (`submit proj2-initial-design`)
  - Sign up for design review slots 10/19 and 10/20 at http://goo.gl/KmRYu
  - Give Angela a hardcopy of your design doc!
- Midterms are graded! Good job ☺
  - Mean 77.5, StDev 13.7, Median 81
  - If you would like a regrade, submit your test along with a written summary of what you want regarded and why to Angela by one week from today (10/25) – remember your whole test will be regraded!

A Brief Introduction to Databases

So far we've talked about how to run processes and get them to communicate with each other over a network. However, to make these mechanisms more useful, we need to have some structure for storing and updating data in a meaningful way. This is where databases come in. Here are some useful definitions:

- *Database:* A collection of data.
- *Schema:* Describes the fields contained in the database as well as how the data is organized.
- *Database Management System (DBMS):* Stores, manages, and facilitates access to the database. The DBMS includes a Data Definition Language (DDL) that dictates how to create schemas and a Data Manipulation Language (DML) that provides a way to query the database and manipulate entries.

Atomic Transactions

Just as we saw in the Readers / Writers problem earlier in the semester, having some shared collection of data means that we have to control access to the data in some way. As before, we want to allow as much concurrent access to the data as possible without causing unexpected or incorrect results. Before thinking about how to structure concurrent access to the database, we first have to define what operations we care about. A *transaction* is a sequence of reads and writes – we would like all transactions to be atomic, which means that regardless of whatever other transactions are taking place, they take the database from one consistent state to another consistent state. All transactions should follow the ACID properties:

- *Atomicity:* Either all actions in the transaction happen, or none of them happen. This can be achieved by logging the operations for the transaction. If something goes wrong partway through, we can abort the transaction by rolling back all the operations for this transaction that have already happened.
- *Consistency:* A set of integrity constraints must hold in order for the transaction to take place; otherwise, the transaction is aborted. Integrity constraints are usually very simple (e.g., checking that all data types are correct, null checks, etc.).
- *Isolation:* Execution of one transaction is isolated from others. Even though the database may be accessed concurrently and operations from different transactions could be interleaved, we want all transactions to execute as if it were running by itself.
- *Durability:* Once a transaction is committed, it should be persistent. This means if the system crashes in any way, all effects of committed transactions (and only effects of committed transactions) should still be there.

We could certainly ensure that all transactions happen correctly if we only allow one transaction to take place at once…but this isn't an efficient solution. We should allow multiple transactions to happen as long as they won't adversely affect each other. But how do we know when conflicts will happen? There are three types of conflicts that can happen with concurrent access (notice that all of them must involve a write operation):

- *Read-write conflict:* Multiple reads within the same transaction don't yield the same result. This happens if another transaction writes to a particular entry between two reads in the first transaction.
- *Write-read conflict:* We end up reading uncommitted data as a result of reading an entry in the middle of a series of write operations, before another transaction was finished writing.
- *Write-write conflict:* One transaction overwrites uncommitted data from another transaction by writing the same entry before the other transaction is finished.

You can see that interleaving operations between transactions can be very dangerous! So what kind of interleaving should we allow? A *serial schedule* is one in which there is no interleaving between transactions. While we don't want to force all transactions to follow a serial schedule for efficiency reasons, we do want to have *serializable schedules*: schedules with interleaving, but are equivalent to some serial schedule. Two schedules are equivalent if they have the same effect on the data. (A given interleaving is called *conflict serializable* if you can swap the order of consecutive operations from different transactions until you have a serial schedule.)

One way we can guarantee conflict serializability is by using the *Two-Phase Locking* (2PL) protocol. In this scheme, we have two different kinds of locks for each piece of shared data:

- *Shared lock (S):* Allows for multiple read-only access to the data.
- *Exclusive lock (X):* Allows for exclusive read-and-write access to the data.

For every transaction, we acquire the locks we need in order to complete the transaction (S or X locks to read, and X locks to write). To ensure that this locking creates conflict serializable interleavings of transactions, we also impose the restriction that no locks are acquired after a lock has been released. This means that all the locks needed for any part of the transaction must be acquired before any locks are released, causing all lock acquires and releases to happen in two stages. The first stage in which locks are acquired is called the *growing phase*, while the stage in which locks are gradually released is called the *shrinking stage*. Forcing these lock acquires and releases to happen in stages guarantees that we will never allow another transaction to come in and modify data before we interact with it later in this transaction, so the two transactions can never become interdependent to produce a particular effect on the data.

Note: In *Strict 2PL*, all locks must be released at the very end of the transaction. This avoids cascading aborts, in which aborting one transaction forces us to abort another transaction as well due to the second transaction taking place in the middle of the first transaction, using data the first transaction modified.

As always with locking, however, deadlock is a possible issue we could run into with 2PL. This can happen if a particular schedule is not conflict serializable but we still try to use 2PL – the interdependency between the two transactions in a non-conflict-serializable schedule indicates that they have a cyclic dependency in waiting for each other to release locks on the data. The easiest way to deal with this is to allow the deadlock to happen, but detect it in some way and abort one of the transactions that is causing the deadlock, and restart that transaction again later. (What considerations would we have to take into account when doing this?)