TRANSACTIONS: A PRIMER

A major component in any distributed system/application is a (distributed) database system that stores, indexes, and manages shared state on behalf of other components of the DS. There are many challenges involved in building a database system, including a non-distributed database system, but two major systems challenges that are of relevance to this course are:

1. Handling failures, which are inevitable:
    - Failures of software, hardware, and in general, storage media. (For non-distributed databases, the failures assume no stable storage media failures or corruptions.)
    - Because of the possibility failure at any time, we have to worry about partial computations and the correctness of computations after restart **(the recovery problem)**.
2. Handling concurrency, which is needed for performance but it's hard to handle:
    - To get both high throughput and low latency in the face of a mixture of I/O devices and CPUs, must exploit concurrency
    - Parallelism – use multiple CPUs simultaneously. Overlapping I/O and computation of the same job, or from multiple jobs, to overcome latency of device/network.
    - But concurrency in the face of data sharing creates consistency problems. Need to use some form of synchronization or conflict detection to avoid races and resulting inconsistency **(the concurrency control problem)**.

This lecture gives a basic primer on *transactions*, a key abstraction offered by most popular databases to address these challenges, plus the core mechanisms used in single-node databases to implement this abstraction. Future lectures will build up a distributed version of a transactional database.

I. THE TRANSACTION ABSTRACTION

A Turing-award-winning idea; a transaction is an abstraction provided to programmers that **encapsulates a unit of work against a database**. Transactions provide a simple but powerful interface:

- txID = **begin**() // starts a transaction; returns a unique ID for the transaction
- outcome = **commit**(txID) // tries to commit a transaction; returns whether or not the commit
  // was successful. If successful, all operations included in the transaction have
  // been applied to the DB. If unsuccessful, none of the operations have been
  // applied.
- **abort**(txID) // cancels all operations of a transaction and erases their effects on the DB. Can
  // be called by the programmer or by the database engine itself.

By wrapping a set of accesses and updates in a transaction, the database can resolve the recovery and concurrency control problems and give a set of meaningful semantic guarantees to applications:

- **Atomicity:** Either all operations in the transaction will complete successfully (commit outcome), or none of them will (abort outcome).
    - Said differently, after a transaction commits or aborts, the database will not reflect a partial result of that transaction.
    - All transactions will either commit or abort.
    - Q: if one were to guarantee failure-freeness, does atomicity come "for free"?
        - A: yes, though it is a wide definition of "failure" for this to be true, e.g., no rollback of conflicting or deadlocking transactions.

- **Isolation**: A transaction's behavior is not impacted by the presence of other, concurrently executing transactions.
    - Said differently, a transaction will "see" only the state of the DB that would occur if the transaction were the only one running against the database, and it will produce only results that it could produce *if it were running alone*.
    - Q: if one executes only a single transaction at a time, does "isolation" come for free?
        - A: yes! this is tied to the very definition of isolation.
- **Durability**: The effects of committed transactions survive failures.
    - If there is non-volatile storage in the system: the effects of a committed transaction must be reflected in non-volatile storage at all times.
    - After a failure, the effects of committed transactions must be recoverable or already reflected in the DB.

These properties are often called **ACID** (yes, there's a C that stands for Consistency, but we don't discuss it in this course).

II. MOTIVATING EXAMPLES

The standard example is bank transactions, and that's what we'll use in this class. But the notion of transactions is much more vastly applicable, even to applications that don't seem to truly "need" transactions. The reason is that it's a lot simpler to program against a strong-semantic database system than it is to program against a weaker-semantic one. The introduction in Google's Spanner paper, which we'll review later, gives broader-context examples.

Here's a piece of code for two programs running on top of a non-distributed database that offers transactions as a programming abstraction.

```
TRANSFER(src, dst, x)
01   src_bal = Read(src)
02   if (src_bal > x):
03      src_bal -= x
04      Write(src_bal, src)
05      dst_bal = Read(dst)
06      dst_bal += x
07      Write(dst_bal, dst)

Invocation: TRANSFER(A, B, 50)
```

```
REPORT_SUM(acc1, acc2)
01   acc1_bal = Read(acc1)
02   acc2_bal = Read(acc2)
03   Print(acc1_bal + acc2_bal)

Invocation: PRINT_SUM(A, B)
```

Without transactions: What could go wrong? Think of crashes or inopportune interleavings between concurrent TRANSFER and REPORT_SUM processes .

TRANSFER:

- Assume statements operates on DB immediately, and DB is a single data structure
    - what happens if there is a crash after 04 but before 07?
    - money is lost
- Why? Because the transfer is **not atomic**

    - need some way of making sure entire transfer happens, or none.
    - That's what the **atomicity property** of ACID transactions gives.

REPORT_SUM:

- Fine if ReportSum() executes before or after TRANSFER()
- what happens if interleaved with TRANSFER()?
    - Depends on the interleaving, some are OK, others not. The following is not OK.
    - Suppose REPORT_SUM's steps are all interleaved between TRANSFER's steps 04 and 05. It will seem like some money was lost. Why is that not OK (for the example)? Suppose A and B are joint accounts, and one owner is transferring money and at the same time another owner checks for the total balance, then the latter will become very confused and will think they've lost some money.
    - Why is that not OK more generally, beyond this trivialized example?
        - Because REPORT_SUM and TRANSFER both depend on the same data
        - And TRANSFER is modifying that data
        - And REPORT_SUM sees both data that predates TRANSFER() (B) and post-dates TRANSFER() (A)
        - That violates this illusion of sequential execution! We lack **isolation**.
        - And the big problem may not be that users get confused, but even worse, applications may get confused, and if they aren't coded to take into account all of these corner-case situations that may happen, they may fail. And it's really, really hard to think about all corner cases. That's why we want strong semantics, so we (as programmers) don't have to worry about corner cases.
- That's what the **isolation** property of ACID transactions give.

With transactions: To fix these challenges, you just modify the TRANSFER and REPORT_SUM to wrap their operations into a transaction, i.e., add begin() and commit() at the beginning and end, respectively, of each method.

So, the idea is that if you build your applications on top of ACID transactions, you won't have to worry about the challenges we described. By and large, there are things to pay for using these strong semantic: transactions *are* expensive, so sometimes you may have to forego their strongest semantics and make do with something weaker. So it's good to understand a bit how these strong semantics are implemented so you can reason about their costs in your application, and potentially how you can weaken them in a way that is still meaningful to your application but more efficient. So, in the next section, we'll look at how single-node, non-distributed databases implement ACID transactions.

III. IMPLEMENTING TRANSACTIONS IN SINGLE-NODE DATABASES

Based on the preceding examples, we need to address two challenges to implement ACID transactions even in a *non-distributed database*.

Atomicity and durability challenges: How do we make sure that the operations included in a transaction either all succeed or none of them succeed despite temporary failures of the machine running the DB? (Remember no distribution in this section.) The key mechanism here is **write-ahead logging**. Assume that disks are reliable and cannot fail, but that machines can fail temporarily. Upon recovery,

they can access the data on disk but RAM data is vanished. The idea in write-ahead logging is to log to disk sufficient information about each operation *before you apply it to the database*, such that in the event of a failure in the middle of a transaction, you can undo the effects of its operations on the database. If you've managed to apply all the operations in a transaction without a failure, then you enter in your log that the particular transaction is completed. Upon a subsequent failure of the DB server, the server will read the logs and apply all transactions that are committed and undo any transactions that were still ongoing at the time of failure. These slides <https://columbia.github.io/ds1-class/lectures/05-local-transactions-wal.pdf>, courtesy of Dave Andersen, describe this mechanism with an example.

With write-ahead logging, you get the following semantic: (1) the operations in a transaction are completed as a unit, i.e., either all (commit outcome) or none (abort outcome) and (2) for any committed transaction, its effects will persist despite database failures, and become available after recovery. The two parts of the semantic correspond to atomicity and durability properties of ACID.

Isolation challenge: How do we make sure that the operations included in a transaction all witness the database in a coherent state, independent of other ongoing (a.k.a., concurrent) transactions? The key mechanism here is **locking**. In one instantiation of this mechanism, the DB acquires locks on all rows read or written and maintains them until the end of the transaction. Read but not written rows can be locked in a shared way, allowing other transactions to read (but not write) them. Rows that are written are locked exclusively (no other readers/writers allowed). There are some challenges that one needs to worry about with locks, including deadlocks; refer to these slides <https://columbia.github.io/ds1-class/lectures/05-local-transactions-2pl.pdf> (also courtesy of Dave Andersen) for a description of these challenges and the solution, which is called two-phase locking (2PL).

Databases use two-phase locking to achieve various levels of isolation between concurrent transactions: If the DB grabs locks for both read and written rows, and retain them till the end of the transaction, then you can get very strong isolation semantics (called *serializability*), but that can be very expensive. This is because other transactions that are trying to access some of the rows you've locked will be blocked waiting for your transaction to finish. Instead, if the DB grabs only locks for written rows, and holds each only while it performs each write operation, then you get weaker isolation semantics (called *read uncommitted*), but that mechanism allows for greater concurrency and hence it's more efficient. In between these two semantics, there are other semantics of intermediary strength and overhead.

In your applications, you will have to choose the strength of the semantic, so it's good to familiarize yourselves with each semantic before you make a choice. In general, the idea is that stronger semantic means you'll find it easier and more intuitive to build your application, but you'll sacrifice performance. Weak semantic means that you'll have to code your application around weird corner cases that the semantic allows, so it's harder to code but it can be made faster.

Key Papers

[Franklin-1992] Michael Franklin. *Concurrency Control and Recovery.*" In *Proceedings of ACM SIGMOD*, 1992.

Acknowledgements

The preceding notes contain portions adapted from: UW's distributed systems course, Steve Gribble's edition (https://courses.cs.washington.edu/courses/csep552/13sp/lectures/4/concurrency_recovery.pdf) and CMU's distributed systems course, Dave Andersen's edition (http://www.cs.cmu.edu/~dga/15-440/F10/lectures/).