ECE356: Database Systems Winter 2018

Lecture 32 — Distributed Databases

Jeff Zarnett

Distributed Databases

A distributed database is basically a scaled up shared nothing system. The database is stored on multiple computers and almost always at multiple physical locations. The systems that make up the distributed database need not be the same (e.g., they may have different CPU types, different amounts of memory, different operating systems, etc). This is probably out of necessity: you can't order a computer with 4000 CPUs and and 64 000 GB of RAM and 2000 hard drives, right? But you can order a thousand computers that each have 4 CPUS, that have 64 GB of RAM and 2 hard drives.

Our discussion of distributed databases is not in any sense a replacement for a distributed systems course but it might be a bit of an introduction to the topic that might increase your interest in it.

Transparency

When designing a distributed system one of the primary concerns is *transparency*, but not the sort we expect of governments where we should know what is going on. It's actually the opposite: what we want is for it to be imperceptible to the users that the database is distributed. There are a few types of transparency that are possible (and generally they are desirable nut not necessarily) [EN11]:

- **Location Transparency**: The command used to perform a task is the same no matter the node on which it executes.
- **Naming Transparency**: When an item has a name, it can be accessed via the name regardless of location.
- **Replication Transparency**: Copies of the data can be stored in multiple locations for reasons such as performance, availability, or reliability. The user should not be aware that the data is a copy (which implies the copies need to agree... or at least not contradict each other).
- **Fragmentation Transparency**: The database can be broken up into multiple parts and it means a query may need to be broken up into multiple subqueries and recombined without noticing.

Fragmentation

Fragmentation is splitting up data. A relation r can be divided up into arbitrarily many fragments $r_1, r_2, ..., r_n$; combining all the fragments again will allow reconstruction of the original relation r.

Horizontal Fragmentation. In horizontal fragmentation, each part of the relation r_i contains some number of complete tuples. Each tuple is assigned to one or more fragments (but it has to be in at least one of them). If we were looking at a bank that had data centres in, say, New York and Hong Kong, then the tuples of the account data for the New York customers would be located in the New York data centre and the HK ones in HK. This keeps tuples close to where they are used the most, to minimize the data transfer requirements; it does not mean that the tuples are inaccessible at other locations; they are just going to take longer to retrieve [SKS11].

If it helps with understanding, the definition of the table is the same at both locations, but some rows are in one database and some in the other.

In relational algebra representation, a fragment r_i is created by a selection with a predicate. So a fragment $r_i = \sigma_{P_i}(r)$ and the entire relation can be reconstructed as a union of the sets [SKS11]. This works only if the

sets are disjoint – no tuples are duplicated. If we have non-disjoint fragments then we might have the possibility of duplicate data, so we would need some sort of de-duplication to produce r again in a consistent way.

Vertical Fragmentation. Vertical integration is splitting up by attributes rather than tuples.

In relational algebra fragmentation is done using projection: $r_i = \Pi_{R_i}(r)$ and they are recombined using the natural join [SKS11]. For

Now, the tables have different definitions – and there are some columns for each row in one database and some in the other.

Replication

Transaction Management

In addition to the usual transaction management, there is the global transaction manager that is, more or less, the boss of the other transaction managers. This may be a position where one database server holds the role permanently (it is the boss), or it might be that the database that handles the actual execution, or the one that started it.

Our biggest concern is making sure that everyone agrees. At a fundamental level that means everyone needs to know about the transactions, and needs to agree about whether the transactions are committed or aborted. The first strategy we will examine is the two-phase commit protocol [EN11]. A transaction is sent out to all the participating databases and they will do it.

In phase one, all databases signal the coordinator they are finished with actual execution. When the coordinator has received that from all participating databases, it sends out a message telling all of them to prepare for commit. Then all the databases force write all log records and anything else the recovery process demands to disk. If all goes well they send back a signal saying OK; if something goes wrong they send back a signal that says "Not OK". If a timeout is reached without any response then a "Not OK" is assumed.

In phase two, if everyone said OK, and the coordinator also says OK, it sends a commit message to all the databases. Every database then proceeds with the commit. If something else happened, then the abort message is sent and all systems roll back the transaction.

At the end of this, all systems have either committed or aborted the transaction and a consistent state is ensured whether the transaction succeeds or not. If a failure occurs during step one it usually results in rolling back, but if a failure occurs during phase 2 it usually results in commit [EN11].

This protocol is good but it has some drawbacks. It is a blocking protocol; if the coordinator system fails, all the participating databases end up waiting until the coordinator system is back online. In the meantime they are locking data elements. Alternatively, if a coordinator and participant that have committed crash together, then we might get an indeterminate result [EN11].

The proposed solution is the three phase locking protocol (MORE PHASES!). The third phase is added in between phase 1 and what previously was phase 2. The new second phase is that the coordinator shares with all participants the results of phase 2. Then phase 3 works just as the commit phase in the two-phase protocol. If the coordinator crashes now, then some other participant can step up and take the coordinator role and see the transaction through. Thus, we can be sure the transaction will finish no matter what system crashes [EN11]. This limits the wait time: based on the phase 1 information we know how the other systems will behave and we can just go ahead and commit; if no such message has been received, abort and proceed with other operations.

References

[EN11] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 6th Edition*. Addison-Wesley, 2011.

[SKS11] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts, 6th Edition*. McGraw Hill, 2011.