Distributed Databases

Alex S.[*]

1 Introduction

For large databases, especially for date warehousing, it often becomes impractical to store and/or process data on a single physical computer. The problem is scalability, of which there are two kinds: vertical scalability and horizontal scalability.

With vertical scalability, to make an application run faster, simply get a faster computer. This quickly hits a wall, as often you cannot get a faster computer (for a reasonable price). For example, at this time, you cannot buy a 100Ghz CPU.

With horizontal scalability, the speed is gained through distributing the load across many computers (each of which may be relatively slow). Instead of hoping for a 100Ghz CPU, get 100 commodity 1Ghz CPUs.

The problem is that vertical scalability is easy. It may be expensive hardware wise, but architecturally, usually nothing changes. Horizontal scalability generally requires a redesign of an existing system, or careful planning of a new system, and many tasks don't lend themselves naturally to being distributed.

Databases are generally considered difficult to scale horizontally. Some easy approaches include:

- multiple read databases and one write database (where the read databases gets updated soon after anyone does an update in a write database).
- splitting databases by region—different branches of the same company could be using a different database. This is how banks were setup at some point (no branch has information about all the customers—your branch where you opened the account would have your info; if you need to get money from a different branch, they would need some mechanism to communicate with your original branch).
- split database by time. Have separate databases for different time periods. Have a very fast database for all the recent data, and then offload it to a slower database if you ever need historical data. Often, the data is simply summarized (gather statistics), backed up to tape, and is never accessed again (and the tapes are shredded after 7 years).

[*]alex@theparticle.com

- split databases by data. With 10 machines, you can distribute user information evenly by doing $machine = userid\%10$. Whenever someone needs to find $userid = 232324$ for example, they would only query $machine$ 4. If someone needs a list of all users they would need to query all machines.

The last approach gets a bit cumbersome if you're doing this at the application level. Imagine your web-application determining which database to connect to based on the username (this is what many web applications do, btw; every web-host has a small database for scratchpad work, such as sessions, and temporary data), and they write permanent data to distributed databases, where the database used depends on the username of the user accessing the page.

The only immediately obvious limitation of such an approach is that data is distributed across multiple computers that don't talk to each other (ie: you cannot join tables that are split across multiple databases in a scheme just described).

2 Distributed Databases

Many databases make record value distribution much easier. Most operate on the idea that the data itself indicates where it is stored. For example, in a table of users, the `userid` may be hashed into a computer number where that record is stored. Some vendors refer to this as 'distribution key'.

Every record (possibly from different tables) that shares the same distribution key will be stored on the same computer.

If a select statement has the distribution key, the select will only hit one computer. In general, most select statement will go against all computers. So if you have 800 computers, table-scan queries will be 800 times faster.

2.1 Distributed Joins

Joins between records that reside on the same computer are done in exactly the same way as any database would do them. For example, if we have `invoice` and `invoicedetails` tables, both of them distributed on `invoiceid`, then doing a join between those two tables on `invoiceid` will simply do a local join on all the separate computers (as we *know* that all matching `invoiceid` are all on the same computer).

The complicated bit comes when the original tables aren't distributed on the join keys.

When one table is not distributed on the join key, you need to *redistribute* that table's data on the join key (create temporary table across all computers that has the join keys as its distribution key). Note that this involves moving all of the data from one table over the network. After this redistribution, you can do a local join (ie: you now know that all the records that satisfy the join condition are on the same computer).

When both tables are not distributed on the join keys, then both tables must be redistributed on the join condition. This involves moving data from both tables across the network. After the redistribution, the we can proceed with a local join.

Redistributions can be avoided by simply picking proper distribution keys in the first place. Ie: If you know that `TABLE_A` will often be joined with `TABLE_B`, you might as well have their join key as the distribution key for both tables—that way, any joins between the two tables can take place in parallel on N computers (ie: if you have 800 computers, then the join is 800 times faster).

2.2 Limitations of Distributed Joins

Problems with such distributed joins is that often, only part of the join condition is part of the distribution key. For example, if you're distributing on `username&email` fields, and doing a join on just `username` (or `username&password`, then you still need to redistribute, as the original distribution isn't helpful in exactly determining the location of joined records.

Other problems involve the not equals condition. Ie: When you're joining on a not equals condition. Such joins usually have other conditions to limit the result set considerably, so if a distributed join picks only the equal join condition, and then does a local join incorporating the not equals condition, usually that works out fine.

Another problem is the range join. For example, joining on 'between' condition, when you have:

```
...
where table1.value between table2.start and table2.end
```

Such joins are a pain in the neck to do in a centralized database, and are much tougher to do in a distributed manner.

One way of handling them in a centralized database is to look at the condition in reverse: instead of matching value to start, and then looking for end, you first sort/index `table1.value`, then for every `table2` record, you can quickly (in logarithmic time) determine if the given range includes any values from `table1`.

Unfortunately, when data is distributed across many computers, such a scheme doesn't work: the problem is that the sorted results (or the index) of `table1.value` values must reside in a single place somewhere.

The way most distributed databases implement this is forwarding the `table2` records to all other nodes (that have `table1` values). This is generally very expensive (if you have 800 nodes, you are potentially transferring 800 times the size of `table2` across the network). For small tables this isn't an issue—but then small tables aren't an issue to handle on a single database.

The distributed nature of the database does allow for a rather clever approach to the between problem. What if instead of doing the 'between' query, we simply quantize the records enough so that we can do an exact match? Note that exact matches, after redistributing, are very efficient.

To do this, we quantize the range of values. For example, if we're doing between on time, we create an extra column that has the 'minute' (we cut off the seconds).

Create a table of all minutes for a day, lets call it `minutes` table. This is a small table with 1400 records per day. Join with `table2` (our table that has start, end). The way this join will work is `minutes` table will be sent to all the database nodes; since it's a small table, it's not a big deal. The join (which is still a 'between' join) will multiply number of records of `table2`—for every minute we'll have a record of `table2`. We then redistribute that table on the minute.

Then we quantize `table1` records on the minute (no need to join with `minutes` table). Distribute the temporary table on minute.

Now since both tables are distributed on minute, we can do a local join. We again also do the 'between' condition, but this time, the join is 'local' within each database node (since only things that fell on a particular minute will be joined on any individual computer).