6.814 - Lecture 15, *Distributed/Parallel Databases*

Long Nguyen

November 6, 2017

1 Topics

We'll spend a few lectures talking about how to take a conventional database design and extend it for parallelism on multiple machines.

Topics to consider:

1. parallelization concepts,
2. DBMS parallel architectures,
3. parallel SQL query processing,
4. parallel transactions.

2 Goal

We want to encapsulate multiple machines and blackbox them as one machine on which users can query for data. On distributed databases, data is stored across machines.

For our purpose, assume that the machines are cooperative.

3 Performance metrics

- reduce execution time (latency) ← analytics
- increase throughput ← transaction processing
- speedup: old time ÷ new time; want this to be linear, but in practice, the curve levels off—if there are too many cores, resource allocation slows down system
- scaleup: small system, small problem ÷ large system, large problem; want this to be constant

4 Barriers to scaling

- no more parallel work units
- fixed costs, e.g. startup costs
- central bottleneck, e.g. shared exlusive data access, shared access $\rightarrow$ internal data structures, dispatch queues, etc.
- skewed data workload when partitioning, giving one node too much work

5 Taxonomy of shared systems

Shared memory systems (like your laptop) have all CPUs sharing main memory on a single machine. In this scheme, cache coherency between CPUs allows for consistency.

PROS: Easy to build, existing concurrency control recovery works.

CONS: Scalability; can only go up to so much. Not fault-tolerant.

Shared disk systems have multiple machines, each with its own CPU, cache, and memory, that cooperate on one disk of data. There must be some coherence between each machine to ensure consistency.

PROS: More scalable and robust (fault-tolerant) in terms of node failures—if one node fails, others can still access the data, whereas in shared nothing, disk i is inaccessible when node i collapses.

CONS: Less scalable than shared memory.

Shared nothing ("partitioned") systems have n disks, with each disk having a certain partition of the data on it. Each machine $m_1, m_2, \ldots, m_n$ talks to disk i.

PROS: Even more scalable; each node responsible for its own disk/partition only. Potentially cheaper, since we can have multiple 1TB disks instead of asking for a huge 1000TB disk to share.

CONS: More complex for fault tolerance policies. Need a new concurrency control recovery protocol, since a node might need to access multiple disks from other nodes.

6 Types of parallelism

Pipelined: Have $(Op1, M1) \rightarrow (Op2, M2)$, essentially passing off output from one to the next. Problems? Limited pipeline depth. Different costs across the pipeline, which can lead to bottlenecks (perhaps one waits on the others in the pipeline).

Partitioned: We try to remove the bottlenecks in the pipeline by running as much of its independent parts in parallel as possible. In this form, we have identical subproblems on each node (think DP).

Imagine three machines that scan, filter, and sort. They can then pipeline their results to one last operator.

7 Data partitionining strategies

Suppose we need to run a lot of queries of the form `select * from R, S where R.a = S.b`. What would we like to arrange about the data?

7.1 Round-robin

Randomly assign tuples to a machine. With this strategy, joining is potentially difficult, since we might have to look at all the partitions to compute the join.

Thus, to join, we could either

1. send both tables to 1 node, or
2. dynamically repartition the data.

7.2 Range partitioning

Partition data into disks by range. Need to make sure that cardinality of each partition is roughly equal; skewing will introduce bottleneck.

With this strategy, we can take advantage of local joins on each node. These independent local joins can then continue to contribute to the overall join.

7.3 Hash partitioning

When we do this, using the same hash function on both tables of the join, anything that matches is sent to the same hash bucket. This doesn't work well for anything other than equijoins. There are also some queries that run on a very precise part of the data (say 10 out of 1,000,000), in which case parallelism hurts.