Lecture 33 — Parallel Databases

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

December 16, 2017

Concurrency and Parallelism

The first way that we could achieve parallelism or concurrency is based on how we decide to implement the server.

If each incoming transaction to be processes is assigned a thread, then we could have *n* worker threads and a transaction gets picked up by a worker.

Keeping in mind that locking and coordination are needed, but the maximum theoretical speedup is limited by the number of processors.

Or a pipeline architecture?

I/O Parallelism

If the database is aware of the various disks available in the system, we can speed up performance via I/O parallelism.

If we need to read some blocks from disk, we could let multiple requests run in parallel.

If there are three blocks we need, and three disks, we could get all three results at (roughly) the same time.

I/O Parallelism: Disk Partitioning

We’ll consider three basic partitioning strategies assuming we have *n* disks:

- **Round-Robin**
- **Hash Partitioning**
- **Range Partitioning**

I/O Parallelism

There are three operations we consider likely to happen:

(1) Scanning the entire relation;

(2) Locating a tuple with some specific attribute equal to a certain value, and

(3) Range queries (where we have a range of acceptable values of the attribute).

ound Robin

Round Robin is good for when we have to read the whole relation.

But no advantage is gained when a specific query or range query is done as we have to search over everything anyway.

Hash Partitioning

Hash partitioning is advantageous when we do specific queries.

This is only beneficial, of course, if the items are partitioned based on that particular attribute.

In theory it search time could be reduced to $1/n$ of the original time.

Otherwise no advantage is gained over round robin.

This is also not especially suitable for range queries.

Range Partitioning

Range partitioning is good for specific as well as range queries on a particular attribute.

A nice advantage of this is if we know that a query will be only on one disk, it's possible to send the query to just one disk.

The other disks are available to perform other operations.

Skew

When the relation is spread over several disks, if things are not spread out evenly, then there is skew in the distribution of tuples.

This can be because of either attribute-value skew: city = “Toronto” being much more common than “Yellowknife”, for example.

Or it can be because of the way the partitioning function works even if values are evenly distributed.

Balancing Skew

To balance this out, one suggested strategy is the histogram approach.

Suppose there are, for example, 5 disks and 100 possible values.

The simple range approach is 20 values in each partition.

If the data has a normal distribution however... Rather than 20 values per disk we would try to cut it so there are about 20% of tuples in each partition.

Slice and Dice!

An alternative approach is to cut it all up more. More? Suppose there are *n* processors and the work is divided up into 5*n* ranges.

If the data is evenly distributed, each CPU will do 5 chunks.

If the work is unevenly divided, however, some CPUs will do a smaller number of large chunks.

Others will do a large number of smaller chunks and some will be in between.

The uneven distribution of values is then spread out over several processors.

Realistic? Maybe...

Admittedly, in a single server database this is not likely to happen even when there are multiple disks.

Usually the disks are arranged in a RAID array and that is done outside the purview of the database server...

So why did we learn about this? Because it will be useful in a distributed database.

Intraquery Parallelism

The subject of inter-query parallelism is actually somewhat boring and redundant as it is just the same as previous discussions of parallelism and concurrency.

Lock shared data, you know the drill.

But there are numerous opportunities to get parallelism inside a single query, and it is a much more interesting subject.

We'll consider two things: intraoperation parallelism and interoperation parallelism.

Intraoperation Parallelism

When we have a single operation that can be, in some way, divided up (partitioned) it is a good candidate for intraoperation parallelism.

- Searching
- Sorting
- Calculation

Partitioned Join

To give a specific example, we will look at parallel join, in particular a partitioned join.

Let's assume we want to join relations *r* and *s*.

If the join is an equality condition such as one attribute in *r* equalling another in *s* then we can divide this up to multiple processors.

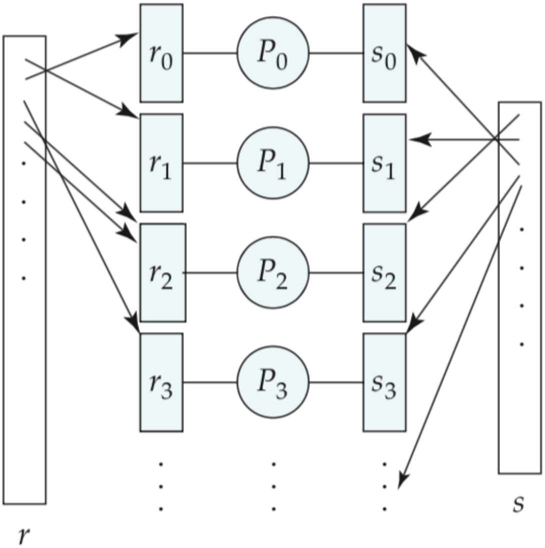
Partitioned Join

If there are n processors we could divide each relation into n pieces and send each processor P_i its piece of r and s.

That processor then compute its part.

These parts are then combined at the end to produce the correct data.

Partitioned Parallel Join



Fragment and Replicate

Partitioning is not suitable for all types of joins, however; because we put in the restriction that there is a join on an equality condition.

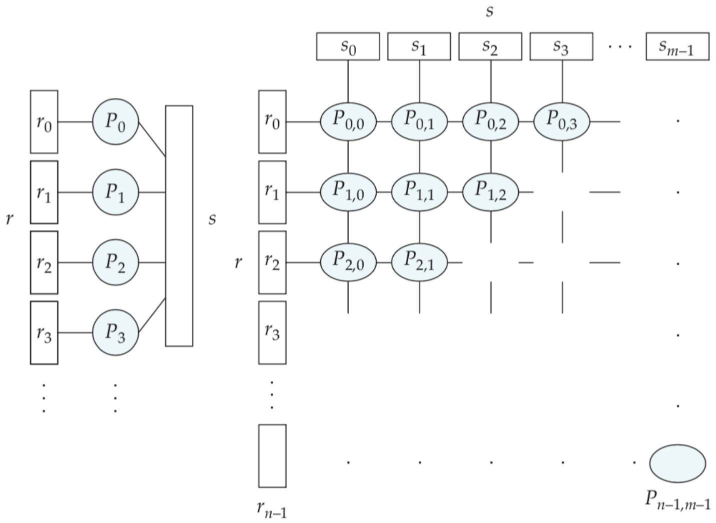
A more general approach is the asymmetric fragment-and-replicate join.

Fragment and Replicate

1. Partition one of the relations (r) and hand out the partitions to the processors.
2. Send copies of the whole other relation (s) to every processor.
3. Each processor computers its join (through whatever method) and the data is recombined.

It would be preferable to choose the smaller relation to be the one that is copied.

Fragment and Replicate



(a) Asymmetric fragment and replicate

(b) Fragment and replicate

Partitioned Parallel Hash Join

Recall from much earlier the hash join: we can parallelize it.

If we have n processors labelled $P_0, P_1, ...P_n$ and we have two relations r and s such that they are spread out across multiple disks.

Remember that we want to choose the smaller relation as the “build” relation, so choose s as the smaller one.

Partitioned Parallel Hash Join

1. Choose a hash function $h1()$ that takes the join attribute value of each tuple in r and maps the tuples to one of the n processors. The subset of tuples of r that are mapped to P_i are called r_i; the tuples of s are similarly divided across the processors and the divisions are called s_i. Each processor P_i reads the tuples of s on its disk and sends them to the appropriate processor based on $h1()$.
2. As a processor receives the tuples from s_i, it further partitions them by a second hash function $h2()$ which is used to compute the hash join locally. This step can be done independent of other processors.
3. Once the tuples of s have been distributed, then r is distributed to the processors based on $h1()$; when it gets there it is repartitioned by $h2()$.
4. Each processor then executes the build and probe phases of the hash-join algorithm on the local partitions r_i and s_i to produce part of the final result.
5. As the last step, the results need to be recombined to produce the final result.

Parallelism Roundup

A quick rundown on how parallelism affects operations other than selection and join:

- **Duplicate Elimination**
- **Projection**
- **Aggregation**

Interoperation Parallelism

We already discussed the idea of pipelining.

This, unfortunately, does not scale well: a pipeline with 4 stages cannot really take advantage of 16 processors.

The next way that we can use parallelism between various operations.

If we are to join four tables, we could compute two pairs of joins in parallel and then combine them in a third operation for the last step.

Change is Hard

We would normally expect that when changes are made to the database schema such as data migration, the database is taken temporarily off the line.

Sometimes we don’t have the option to take things offline for an extended period.

Change is Hard

If a new index is to be constructed, for example, we can’t just lock the relation in shared mode.

We have to allow insertion, deletion, and update while the index is being added.

This is usually done by just keeping track of all the changes as they come in.

Then, adjust the index when generated to account for all the changes that happened while it was being built.

Parallel Database Architecture

Now if we have a parallel database, we should consider the architecture. how it is set up.

We will consider three possible options: shared memory, shared disk, and shared nothing.

Shared Memory

This is our typical multicore or multi-CPU architecture we have likely already become familiar with when discussing concurrency.

Memory is shared between the various processors and they have a common set of disks.

There is no redundancy, however, as it is all wrapped up into one machine.

This means that communication can take place between the threads using shared memory.

Shared Disk

In this architecture all processors can access disks directly, but every processor has its own memory.

This has some degree of redundancy as we might be able to keep working if something goes wrong at one of the systems.

If one of the systems crashes, every system can continue executing without any problem, at some reduced performance level.

The tradeoff is that communication has to take place using the disk.

Shared Nothing

In a shared nothing system, well, it’s exactly what it sounds like: nothing is shared between the systems.

If they wish to communicate, then communication takes place over the network.

This means the systems are as redundant as possible: if any one system goes down we might be able to carry on.

Shared Nothing

This sort of architecture isn’t so much parallel as it actually is distributed…