ECE356: Database Systems Winter 2018

Lecture 33 — Parallel Databases

Jeff Zarnett *2023-08-11*

Parallelism in Databases

Although this is not a course on concurrency, parallelism, and synchronization, we will take some small amount of time to discuss how these impact on database implementation. Let's assume that we have a multicore system available because, let's face it, it's not 1999 anymore.

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. Alternatively there could be a more segmented approach where rather than a worker that takes a job from start to finish there are different "segments" of a pipeline and those run in parallel. But that sort of thing is what we have covered in previous discussions.

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.

We'll consider three basic partitioning strategies assuming we have n disks available [SKS11]:

- **Round-Robin**: When data is being written, the ith tuple is sent to disk i modulo n. This gives very even distribution of tuples.
- **Hash Partitioning**: Each tuple is passed through some hash function that produces a result in the range $[0, n-1]$ and each tuple of the relation is sent to the disk that matches the hash function.
- **Range Partitioning**: Assigns ranges of tuples to various disks. For example, is the values can be A, B, C, or D and there are two disks, we might assign A and C to disk 0 and B and D to disk 1 (or any other combination we like).

These techniques as described in the book ignore some other potential ways to decide, such as thinking how often tuples are accessed. Putting frequently used ones near one another might be sensible... or it might make more sense to put them on different disks so they can be retrieved in parallel.

Different approaches are good for different things. A quick recap from [SKS11] follows. There are three operations we consider likely to happen: scanning the entire relation, locating a tuple with some specific attribute equal to a certain value, and range queries (where we have a range of acceptable values of the attribute).

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 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 the 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 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 and the other disks are available to perform other operations.

When the relation is spread over several disks, if things are not spread out evenly, then there is *skew* in the distribution of tuples: a large percentage of them being places in some areas and a very small percentage in others. This can be because of either *attribute-value* skew – that is, some values are more common in the data such as city being "Toronto" being much more common than "Yellowknife"; or it can be because of the way the partitioning function works even if values are evenly distributed [SKS11].

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 (in the statistical sense of the normal distribution, the bell curve), rather than 20 values per disk we would try to cut it so there are about 20% of tuples in each partition.

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

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... It is either managed by the operating system or by the hardware (even better). So why did we learn about this? Because it will be useful in a distributed database system, where there are multiple disks available to the database system. That is a subject we will return to soon.

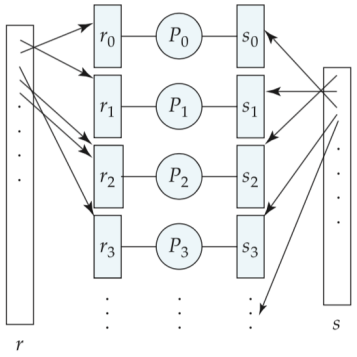
Intraquery Parallelism

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 (parallelizing an individual operation of the query) and interoperation parallelism (doing more than one operation at a time).

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. A number of them were likely things you already learned about in a concurrency course (just usually on some other data items), just now with tuples:

- Searching (e.g., linear search parallelized)
- Sorting (e.g., merge sort)
- Calculation (e.g., compute partial sums and combine)

Partitioned Join. To give a specific example, we will look at parallel join, in particular a partitioned join, as in [SKS11]. 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. 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. Keep in mind that we do probably want to partition things with a histogram approach so each processor has roughly equal work.



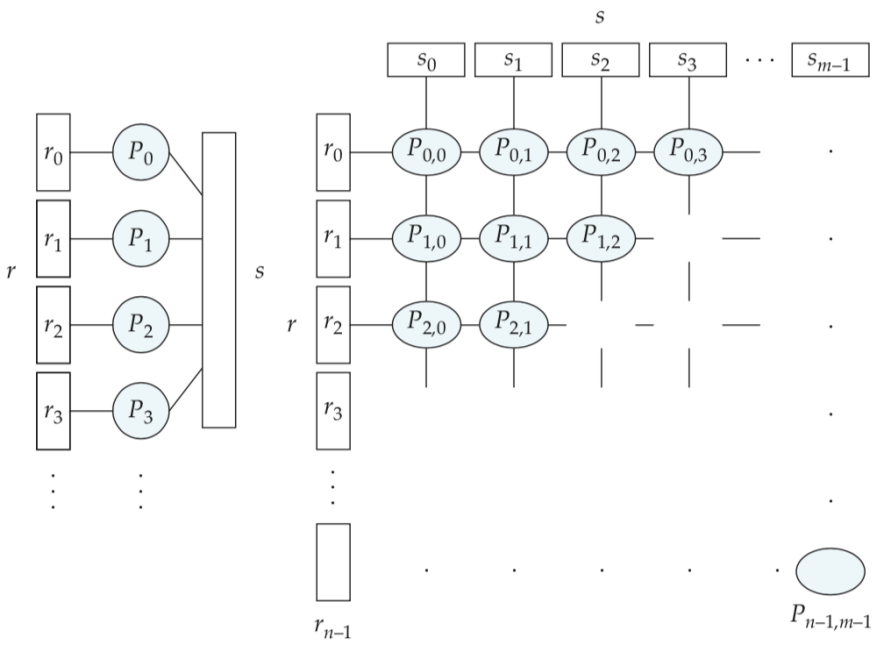
Partitioned Parallel Join [SKS11].

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* which has three steps [SKS11]:

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 computes 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 as it would obviously mean less data needs to be copied to execute this algorithm.

The diagram below shows the asymmetric fragment-and-replicate join; the general case appears on the right where there are $m \times n$ processors rather than just n. In a typical server, the asymmetric approach is suitable. But in theory if you had a very large number of cores then you might use the general replication routine.



Fragment and Replicate [SKS11].

A quick rundown on how parallelism affects operations other than selection and join [SKS11]:

- **Duplicate Elimination**: Duplicates are typically removed by sorting, which itself is something we could do in parallel. Also, if we partition a relation on an attribute, each processor can easily remove duplicates of that attribute.
- **Projection**: Projection can be done as each item is read from disk; just take what you need.
- **Aggregation**: Aggregation can be partially done locally and then the results can be combined to produce a global value. That is easy to imagine for something like sum (each processor computes its local sum, they are all added together). The same is true for count. Minimum and maximum involve comparison rather than addition to combine their values. Average, however, is more interesting: we would likely take the numerator (sum) and denominator (number of tuples) and recombine those.

Interoperation Parallelism. We already discussed the idea of pipelining: the partial result of one step can be moved on to the next stage and processing in that next stage will be able to begin processing. 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. You probably will not want to join more than a few tables in any single query so the amount that this can scale is also limited.

Change is Hard

We would normally expect that when changes are made to the database schema such as data migration, changing column structure, or adding an index are made, the database is taken temporarily off the line (out of production) so the changes can be made while nothing else is going on. You have surely experienced this when you've tried to use an application on the web that tells you it's currently offline for maintenance. Such maintenance is usually planned for what are considered "off hours" (when utilization is low) and some downtime window is planned. Oftentimes when the downtime is significant it's because a database change needs to be made. And all things considered, when you have the option to do the change offline, you probably should.

Sometimes we don't have the option to take things offline for an extended period. The system is critical or there are other business reasons why you can't just turn things off. If the change is small it's not really a big problem. But we can do online migration if we must.

If a new index is to be constructed, for example, we can't just lock the relation in shared mode and 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 and adjusting the index when generated to account for all the changes that happened while it was being built.

Parallel Database Architectures

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. There are multiple processors (or multiple cores, processing units – as far as we are concerned all we want is workers). 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. Every processor has its own cache and cache coordination is a factor, but the hardware helps us out there (for more details take ECE 459!).

We should already be familiar with this sort of coordination from earlier: semaphores, mutexes, and the like, are used to achieve the coordination we need.

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. So to coordinate (e.g., lock an item) the lock information has to be written to disk and that means that disk access is likely to be a serious bottleneck in the system.

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.

This sort of architecture isn't so much parallel as it actually is *distributed* and it comes with its own problems, such as what happens if we need data that is on the disk of some other system, since it does, after all, take longer to get data from a remote location than if it is locally available...

References

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