CS 784: Foundations of Data Management *Instructor: Paris Koutris*

Lecture 11: Parallel Query Evaluation

In *parallel* database systems, the goal is to speed up the evaluation of queries by throwing more machines to the problem in hand. Why are we interested in using parallelism in the context of databases? It is cheaper and more scalable to use more machines (and thus increase parallelism) rather than using faster and faster processors. SQL is particularly amenable to parallelism, since as we show in previous lectures it is an *embarrassingly parallel* language. Recall that from a theoretical viewpoint we argued that Relational Algebra (and SQL) is in the complexity class AC_0.

In the context of parallel database systems, we typically study two system architectures:

1. **Shared memory**: the computing nodes share RAM and disk. This architecture is relatively easier to program, but expensive to scale.
2. **Shared nothing**: the computing nodes have their own RAM and disk, and they share no resources. The nodes connected through a fast network. This is the cheapest solution and scale up to thousands of machines. However, it is very hard to program, since the developer must specify how the communication and the computation interleave.

In this lecture, we will focus on the *shared-nothing model*, since it is the architecture commonly used by most modern parallel databases, as well as most large-scale data analytics systems (e.g. Spark, MapReduce [MR]).

Parallel DBMSs use three different types of parallelism for query processing:

- **Inter-query parallelism**: a query workload can be distributed among multiple machines, by sending each query to be processed in a different location. A common use case for this is transaction processing, where each transaction can be executed in a different node.
- **Inter-operator parallelism**: a single query can be distributed among multiple processors, by splitting the query plan such that different operators of the plan run in different machines.
- **Intra-operator parallelism**: a single operator can be distributed among multiple processors. This is also commonly referred to as *data parallelism*.

There exists a huge amount of literature for all types of parallelism, but here we will focus on intra-operator parallelism, since this is the one that can provide the largest gains in performance.

11.1 Partitioning the Data

Before we discuss models and algorithms on how to evaluate in parallel relational queries, we will see different ways that we can partition the data across the nodes. Consider the relation

$R(\underline{K}, A, B, C)$, where K is the key. Assume that the relation has n tuples, and the number of nodes is p. Here are different ways to achieve the so-called *horizontal data partitioning*:

- **Block partitioning**: arbitrarily partition the data such that the same amount of data n/p is placed at each node. For example, we can send the first tuple to node 1, the second to node 2, and so on, in a round-robin fashion.
- **Hash partitioning**: partition the data by applying a hash function on one or more attributes of the relation. In particular, we take a hash function h that maps a value from the domain to $1, \ldots, p$, and then send each tuple to the corresponding node.
- **Range partitioning**: partition the data according to a specific range of an attribute (such that close values are in the same or nearby servers). In this, we find separating points $k_1, \ldots, k_p$, and then send to the first node the tuples such that (say we consider attribute A), $-\infty \leq t.A \leq k_1$, the second node get $k_1 < t.A \leq k_2$, and so on.

When we partition our data, we may create *skew* if the data is not evenly distributed evenly among the nodes. Ideally, we want each node to have n/p data. Observe that block partitioning will never create skew, but hash and range partitioning can create skew. For example, if we hash-partition on attribute A, and in the database instance all tuples have the same value a for A, then all tuples will be sent to the same machine, creating unbalanced partitioning. If we partition on the key attribute K, and we assume that the hash function distributes the values well, then we can guarantee that every machine will get the same number of tuples, approximately $O(n/p)$ (since each value of K appears exactly once!).

Now, suppose we want to run a *selection query* on the partitioned relation, for example $\sigma_{A=10}(R)$. If the relation is hash partitioned (say with hash h) and the selection condition is on the attribute we are hashing (i.e. A), it suffices to execute the selection condition only on the machine $h(10)$. Otherwise, we can in parallel execute the selection on every data fragment on each machine.

11.2 Modeling Parallelism

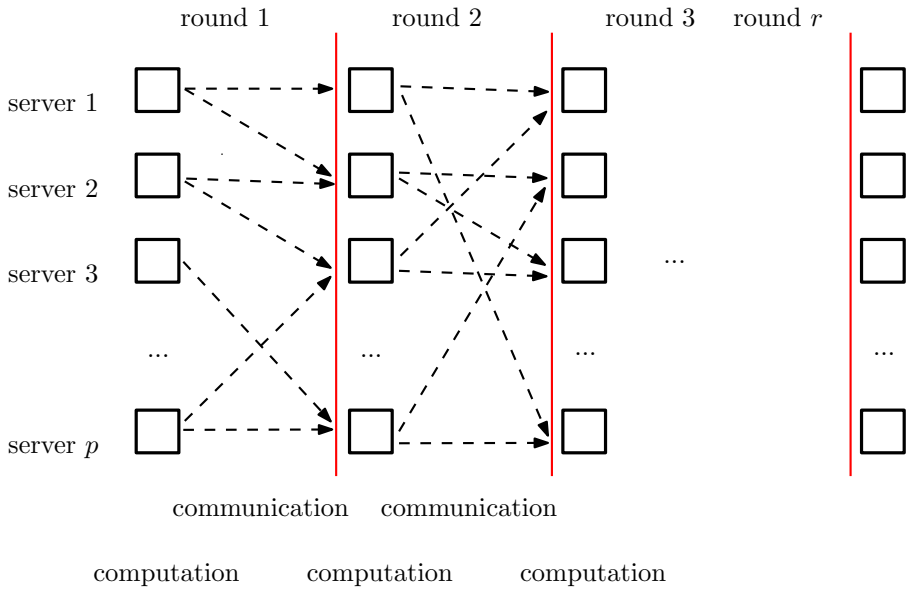
In order to analyze the behavior of a shared-nothing parallel system, we first need to establish a theoretical model of parallel computation, and consider the various parameters in play.

The first important parameter is the *number of computing nodes* (or machines, or processors) that we use: we will denote that by p.

Second, we need to specify how the communication and computation interleaves. If we allow an arbitrary interleaving of communication and computation, then we model the parallel system as an *asynchronous system*. Some systems allow asynchronicity, but most parallel systems work in a *synchronous* manner. In such a system, the computation proceeds in *rounds*: each round consists of some local computation followed by global exchange of data between the machines. At the end of each round, the machines have to synchronize, i.e. wait for all machines to finish before

proceeding to the next round. This type of parallelism with synchronization is typically called BSP-style computation (from the Bulk-Synchronous-Parallel Model [BSP]). We denote the number of rounds by r: the number of rounds measures the amount of synchronization in a computation.

Third, we need to measure the amount of *communication* during the computation. Instead of measuring the total communication, we will focus on the *maximum load*, which is the maximum amount of data that a machine can receive at any round during computation. A smaller load means that data is more evenly distributed, and the amount of data communicated is smaller. We typically denote this parameter by L. Ideally, we want to achieve $L = n/p$, where n is the size of the input data.



11.3 Computing Joins in Parallel

We will discuss here how to parallelize the basic join, as well as how to parallelize multiway joins (so Conjunctive Queries without projections and selections). Suppose initially we want to compute the binary join:

$$q(x, y, z) \text{ :- } R(x, y), S(y, z)$$

Hash Join. The standard parallel join algorithm is called *parallel hash join*. The idea is to use a hash function h to partition both R, S on their join attribute (which in this case is B). Each tuple $R(a, b)$ will go to machine $h(b)$, and each tuple $S(b, c)$ will go to machine $h(b)$. After the data has been partitioned, each machine will compute the join by joining its local data fragments.

Cartesian Product. If we want to compute the cartesian product $q(x, y)$:- $R(x), S(y)$, the problem becomes more interesting. Suppose that $|R| = n_R$ and $|S| = n_S$. A naive way for a parallel implementation would be to broadcast the smallest relation to all machines, and then partition evenly the largest relation. But we can do better!

The key idea is to organize the p machines into a $p_R \times p_S$ rectangle, such that $p = p_R \cdot p_S$. Then, we can identify each machine with a pair of coordinates. We then pick two hash functions, $h_R : dom \to \{1, \ldots, p_R\}$ and $h_S : dom \to \{1, \ldots, p_S\}$. Then, we send each tuple $R(a)$ to all the machines with coordinates $(h_R(a), *)$, so to all the machines of a specific row chosen by our hash function. Similarly, we send $S(b)$ to all the machines with coordinates $(*, h_S(b))$. After the data is partitioned, each machine again locally computes the cartesian product. The algorithm is correct, since each tuple (a, b) can be discovered in the machine with coordinates $(h_R(a), h_S(b))$.

We can now calculate how much data each machine receives. Assuming that the hash functions behave well, each machine will get $\frac{n_R}{p_R} + \frac{n_S}{p_S}$ tuples. To minimize this load, we have to make $\frac{n_R}{p_R} = \frac{n_S}{p_S}$, and so we can choose $p_R = \sqrt{p\frac{n_R}{n_S}}$ and $p_S = \sqrt{p\frac{n_S}{n_R}}$.

Multiway Joins. Consider the triangle query $q(x, y, z)$:- $R(x, y), S(y, z), T(z, x)$. The standard way to compute this query in parallel is to use two rounds. In the first round, we perform a parallel hash join between R, S to obtain an intermediate relation $RS(x, y, z)$. In the second round, we perform another parallel hash join between RS and T (where we can join on one or two attributes). The potential problem with this approach is that the intermediate result RS can be very large, which would mean very expensive communication in the second round.

An alternative way to compute the triangle query is to use the same idea as the cartesian product and compute the query in a single round! The algorithm we present is called in the literature the SHARES or HYPERCUBE algorithm [AU10]. Here, we organize the p machines into a 3-dimensional hypercube (observe that the number of dimensions is the same as the number of variables): $p = p_x \times p_y \times p_z$. Each machine now identifies with a unique point in the 3-dimensional space. The algorithm uses a different hash function for each variable.

Now, each tuple $R(a, b)$ will be sent to all machines with coordinates $(h_x(a), h_y(b), *)$. Similarly, each tuple $S(b, c)$ will be sent to coordinates $(*, h_y(b), h_z(c))$ and each tuple $T(c, a)$ to $(h_x(a), *, h_z(c))$. After the data has been communicated, each machine will locally compute the triangles, using any single-machine algorithm.

Assume for the moment that we choose $p_x = p_y = p_z = p^{1/3}$. Then, we can see that each tuple will be replicated to $p^{1/3}$ machines.

At this point, we should note two things. First, the idea of the triangle query can be generalized to compute any conjunctive query using a single round. Second, we need to find the optimal size of each dimension. This is not a trivial task, since the optimal size will have a complex connection to the relation sizes and the structure of the query (for more details see [BKS14]).

References

[MR] J. DEAN, and S. GHEMAWAT, "MapReduce: simplified data processing on large clusters", *OSDI, 2004*

[S86] M. STONEBRAKER, "The Case for Shared Nothing.", *Database Engineering, 1986*

[AU10] F. AFATI and J. ULLMAN, “Optimizing Joins in a Map-Reduce Environment.”, *EDBT, 2010*

[BKS14] P. BEAME, P. KOUTRIS and D.SUCIU, “Skew in parallel query processing.”, *PODS, 2014*

[BSP] L. VALIANT, "A bridging model for parallel computation", *CACM, 1990*