CS 838: Foundations of Data Management **Spring 2016**

Lecture 10: Parallel Query Evaluation

Instructor: Paris Koutris 2016

In *parallel database systems*, we want to speed up the evaluation of relational queries by throwing more machines to the problem in hand. Contrast this to *distributed database systems*, where data may be stored across different machines in potentially different databases.

Why are we interested in using parallelism in the context of databases? It is always cheaper and more scalable to use more machines (and thus increase parallelism) rather than finding faster and faster processors. SQL is particularly amenable to parallelism, since we discussed that it is *embarrassingly parallel*. Recall that from a theoretical viewpoint this meant that relational algebra was in the complexity class AC_0.

There are two critical notions that measure performance in the presence of parallelism:

1. **Speed-up**: using more processors, how much faster does the query run (if the problem size is fixed)?
2. **Scale-up**: using more processors, does performance remain the same as we increase the problem size?

In parallel database systems, we typically study three different systems architectures:

1. **Shared memory**: the nodes share RAM + disk. This architecture is easy to program, but expensive to scale.
2. **Shared disk**: the nodes access the same disk; this is also hard to scale.
3. **Shared nothing**: the nodes have their own RAM+disk, 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!

We will focus mostly on the shared-nothing model, since it is commonly used by most modern parallel databases, and other large-scale data analytics systems (e.g. Spark, MapReduce, etc).

10.1 Types of Parallelism

Parallel DBMSs use three different types of parallelism:

- **Inter-query parallelism**: each query runs on one processor, but different queries can be distributed among different nodes. A common use case for this is transaction processing, where each transaction can be executed in a different node.
- **Inter-operator parallelism**: each query runs on multiple processors. The parallelism corresponds to different operators of a query running in different processors.
- **Intra-operator parallelism**: a single operator is distributed among multiple processors. This is also commonly referred to as *data parallelism*.

We will focus mostly on intra-operator parallelism, since this is the most interesting case and also provides the largest speed-up gains.

10.2 Types of Data Partitioning

Before we discuss some simple algorithms of how we can 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 n. 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.
- **Hash partitioning**: partition the data by applying a hash function on some attribute 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.

Example 10.1. *Consider the relation $R(\underline{K}, A, B, C)$, where K is the key. What kind of partitioning will create skew? Why having skew could be a problem?*

10.3 Parallel Operators

We will discuss how to parallelize some basic operators, as well as a general technique to compute multiway joins.

- **Selection $\sigma_{A=10}(R)$:** if the data is hash partitioned (say with hash h) and the selection condition is on the attribute we are hashing, 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.
- **Join $R(A,B) \bowtie S(B,C)$:** 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 quantity, 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 steps. In the first step, we perform a parallel hash join between R, S to obtain an intermediate relation $RS(x,y,z)$. In the second step, 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 step.

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 step! 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

[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*