CS W186
Fall 2019

Parallel Query Processing

1 Introduction to Parallelism

Parallel processing is used to efficiently solve problems that can be decomposed into independent subproblems. In this note, we see parallelism in two main ways:

- Parallelism through pipelining - each operator processes the previous operator's output as it is being produced
- Parallelism through Partitioning - partition data across multiple machines so that algorithms can be executed in parallel

When it comes to hardware that can handle parallel execution, three main architectures exist:

- **Shared Memory** - one or more CPUs with both shared memory and disk on one machine
- **Shared Disk** - one or more CPUs with independent memory and shared disk on one machine
- **Shared Nothing** - one of more machines connected by a network

Out of these architectures, Shared Nothing architectures are the most common for databases as scaling is made simple by adding more machines to the network when accommodating higher loads.

2 Query Parallelism

Queries can be parallelized using either **Inter-Query parallelism** or **Intra-Query parallelism**. Inter-Query parallelism involves executing multiple queries at once and relies on transactions while Intra-Query parallelism involves splitting a single query into operators that can be executed concurrently.

Intra-Query parallelism can be further separated into two categories: **Inter-Operator parallelism** and **Intra-Operator parallelism**. Inter-Operator parallelism works by either pipelining operators or executing operators in parallel. Meanwhile, Intra-Operator parallelism focuses on parallelizing single operators by partitioning data across machines using **range**, **hash**, or **round robin** and executing algorithms locally on each machine.

For parallel operations, network costs must be considered and these costs are measured by looking at the total size of pages sent over the network in bytes.

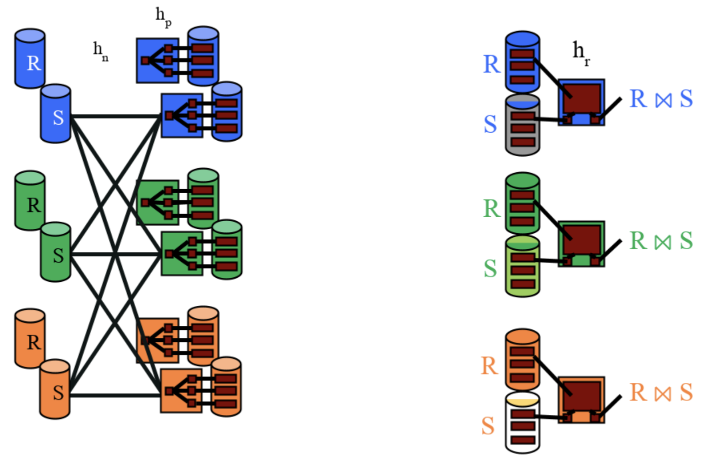
An important note is that if a database has a Shared Nothing architecture, large relations are already initially distributed across machines. When partitioning, each machine's data is redistributed to a machine associated with a given hash or key range.

3 Parallel Hashing

External hashing can be parallelized by partitioning data across machines using a hash function h_1. As the data is distributed, each machine will independently run the external hashing algorithm using new hash functions $h_2, ..., h_k$.

4 Parallel Grace Hash Join

Grace Hash Join can be parallelized similarly to how external hashing is parallelized. Since a join involves two relations, each of the relation's data is partitioned across machines using a hash function h_1 and joined locally by running Grace Hash Join. Once a machine starts receiving data, it can execute independently from all other machines.



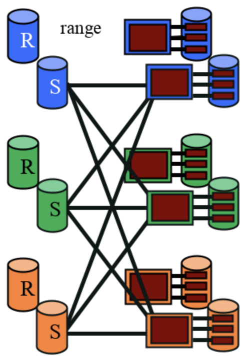
5 Parallel Sorting

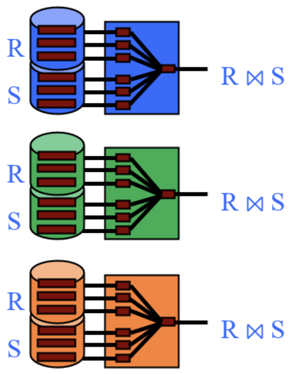
External sorting can be parallelized by partitioning using ranges. Each machine is assigned a range of keys and data with keys falling in a specified range are sent to that associated machine. As the data is distributed, each machine will independently run the external sorting algorithm.

A potential downside of range partitioning is **data skew**. Different machines can end up with drastically different amounts of data if lots of data fall within some ranges and not others. This can be fixed by sampling the data and determining a distribution before assigning ranges so that data can be spread more evenly across machines.

6 Parallel Sort Merge Join

Sort Merge join can be parallelized in the same way that external sorting is parallelized. Data from each relation in the join is range partitioned across machines and each machine independently executes the sort merge join algorithm locally.





7 One-Sided Shuffle Join

If one relation is already roughly partitioned across machines, partition the other relation and run a local join at every machine. The method of partitioning and the join algorithm depend on how the first relation is initially distributed.

8 Broadcast Join

If one relation is small, it is initially stored only on one machine and broadcasted to all other machines containing a partition of the other relation so that local joins can be performed in parallel.

Assuming n = size(file1) in pages, m = size(file2) in pages, $n < m$, k = # of machines storing partitions of the 2nd file, and the machine storing the 1st file does not contain a partition of the 2nd file, the way to determine when a a broadcast join is advantageous is when the following condition is met:

$$n * k < n + m$$

If the condition is satisfied, it means that sending the smaller relation to k machines costs less than partitioning both relations across all machines.