Lecture #13: Query Execution II

15-445/645 Database Systems (Fall 2020)
https://15445.courses.cs.cmu.edu/fall2020/
Carnegie Mellon University
Prof. Andy Pavlo

1 Background

Previous discussions of query executions assumed that the queries executed with a single worker (i.e thread). However, in practice, queries are often executed in parallel with multiple workers.

Parallel execution provides a number of key benefits for DBMSs:

- Increased performance in throughput (more queries per second) and latency (less time per query).
- Increased responsiveness and availability.
- Potentially lower *total cost of ownership* (TCO). This cost includes both the hardware procurement and software license, as well as the labor overhead of deploying the DBMS and the energy needed to run the machines.

There are two types of parallelism that DBMSs support: inter- and intra- query parallelism.

2 Parallel vs Distributed Databases

In both parallel and distributed systems, the database is spread out across multiple "resources" to improve parallelism. These resources are either computational (e.g., CPU cores, CPU sockets, GPUs, additional machines) or storage (e.g., disks, memory).

It is important to distinguish between parallel and distributed systems. In a *parallel DBMS*, resources, or nodes, are physically close to each other. These nodes communicate with high-speed interconnect. It is assumed that communication between resources is not only fast, but also cheap and reliable.

In a distributed database, resources a far away from each other; this could mean a database could span racks or data centers in different parts of the world. As a result, resources communicate using slower interconnect over a public network. Communication costs between nodes is slower and failures cannot be ignored.

Even though a database may be physically divided over multiple resources, it still appears as a single logical database instance to the application. Thus, the SQL query for a single-node DBMS should generate the same result on a parallel or distributed DBMS.

3 Process Models

A DBMS *process model* defines how the system supports concurrent requests from a multi-user application/environment. The DBMS is comprised of more or more *workers* that are responsible for executing tasks on behalf of the client and returning the results. An application may send a large request or multiple requests at the same time that must be divided across different workers.

There are three different process models that a DBMS could adopt: process per worker, process pool, and thread per worker.

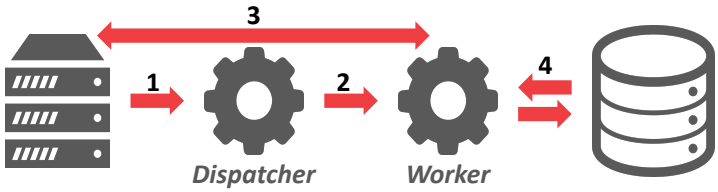

Figure 1: Process per Worker Model

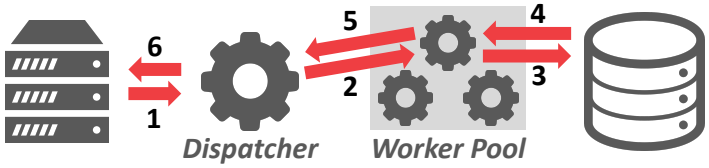

Figure 2: Process Pool Model

Process per Worker

The first and most basic approach is *process per worker*. Here, each worker is a separate OS process, and thus relies on OS scheduler. An application sends a request and opens a connection to the databases system. Some dispatcher receives the request and forks off a worker to handle this connection. The application now communicates directly with the worker who is responsible for executing the request that the query wants. This sequence of events is shown in Figure 1.

This raises the issue of multiple workers on separate processes making numerous copies of the same page. A solution to maximize memory usage, is to use shared-memory for global data structures so that they can be used by multiple processes.

An advantage of the process per worker approach is that a process crash doesn't disrupt the whole system because each process is forked off.

Process Pool

An extension of the process per worker model is the *process pool*. Instead of forking off processes for each connection request, workers are kept in a pool and selected by the dispatcher when a query arrives. Because the processes exist together in a pool, processes can share queries between themselves, or query parallelism. A diagram of the process pool model is shown in Figure 2.

Like process per worker, the process pool also relies on the OS scheduler and shared memory.

A drawback to this approach is poor CPU cache locality as the same processes are not guaranteed to be used between queries.

Thread per Worker

The third and most common model is *thread per worker*. Instead of having different processes doing different tasks, each database system has only one process with multiple worker threads. In this environment, the DBMS has full control over the tasks and threads, it can manage it own scheduling. The multi-threaded model may or may not use a dispatcher thread. A diagram of the thread per worker model is shown in Figure 3.

Using multi-threaded architecture provides certain advantages. For one, there is less overhead per context

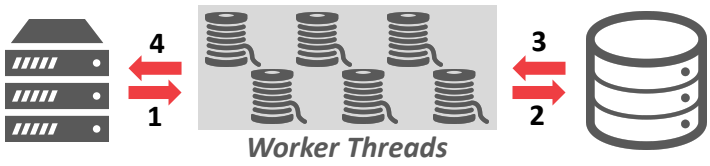

Figure 3: Thread per Worker Model

switch. Additionally, a shared model does not have to be maintained. However, the thread per worker model does not allow for intra-query parallelism.

In conclusion, for each query plan, the DBMS has to decide where, when, and how to execute. Relevant questions include:

- How many tasks should it use?
- How many CPU cores should it use?
- What CPU core should the tasks execute on?
- Where should a task store its output?

When making decisions regarding query plans, the DBMS always knows more than the OS and should be prioritized as such.

4 Inter-Query Parallelism

In *inter-query parallelism*, the DBMS executes different queries are concurrently. Because multiple workers are running requests simultaneously, overall performance is improved. This increases throughput and reduces latency.

If the queries are read-only, then little coordination is required between queries. However, if multiple queries are updating the database concurrently, more complicated conflicts arise. These issues are discussed further in lecture 16.

5 Intra-Query parallelism

In *intra-query parallelism*, the DBMS executes the operations of a single query in parallel. This decreases latency for long-running queries.

The organization of intra-query parallelism can be thought of in terms of a *producer/consumer* paradigm. Each operator is a producer of data as well as a consumer of data from some operator running below it.

Parallel algorithms exist for every relational operator. DBMSs can either have multiple threads access centralized data structures or use partitioning to divide work up.

Within intra-query parallelism, there are three types of parallelism: intra-operator, inter-operator, and bushy. It is of note that these approaches are not mutually exclusive. Part of the DBMSs responsibility is to combine these techniques in a way that optimizes performance of a given workload.

Intra-Operator Parallelism (Horizontal)

In *intra-operator parallelism*, query plan's operators are decomposed into independent instances that perform the same function on different subsets of data.

The DBMS inserts an *exchange* operator into the query plan to coalesce results from children operators. The

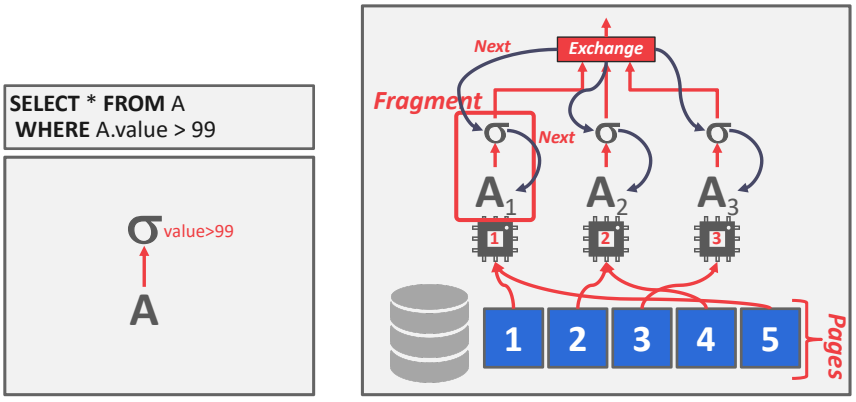

Figure 4: Intra-operator Parallelism – The query plan for this select is a sequential scan on A that is fed into a filter operator. To run this in parallel, the query plan is divided among different fragments. A given plan fragment is operated on a by a distinct page. The exchange operator calls Next concurrently on all fragments which then retrieve data from their respective pages.

exchange operator prevents the DBMS from executing operators above it in the plan until it receives all of the data from the children. An example of this is shown in Figure 4.

In general, there are three types of exchange operators:

- **Gather:** Combine the results from multiple workers into a single output stream. This is the most common type used in parallel DBMSs.
- **Repartition:** Reorganize multiple input streams across multiple output streams. This allows the DBMS take inputs that are partitioned one way and then redistribute them in another way.
- **Distribute:** Split a single input stream into multiple output streams.

Inter-Operator Parallelism (Vertical)

In *inter-operator parallelism*, the DBMS overlaps operators in order to pipeline data from one stage to the next without materialization. This is sometimes called *pipelined parallelism*. See example in Figure 5.

This approach is widely used in *stream processing systems*, which are systems that continually execute a query over a stream of input tuples.

Bushy Parallelism

Bushy parallelism is an extension of inter-operator parallelism where workers execute multiple operators from different segments of a query plan at the same time.

The DBMS still uses exchange operators to combine intermediate results from these segments. An example is shown in Figure 6.

6 I/O Parallelism

Using additional processes/threads to execute queries in parallel will not improve performance if the disk is always the main bottleneck. Thus, it is important to be able to split a database across multiple storage devices.

To get around this, DBMSs use I/O parallelism to *split installation across multiple devices*. Three ap-

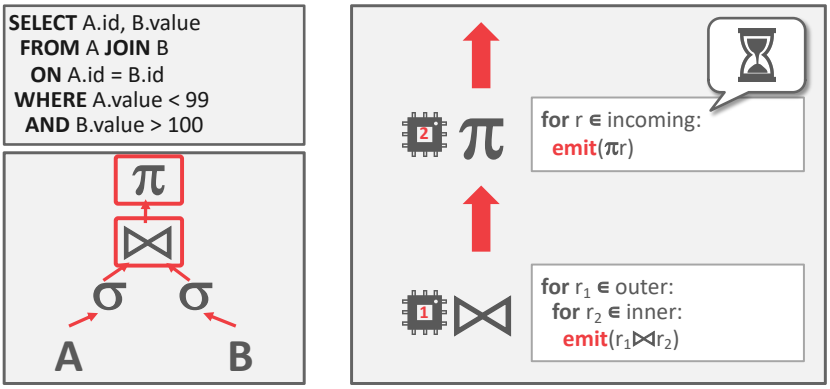

Figure 5: Inter-operator Parallelism – In the join statement to the left, a single worker performs the join and then emits the result to another worker that performs the projection and then emits the result again.

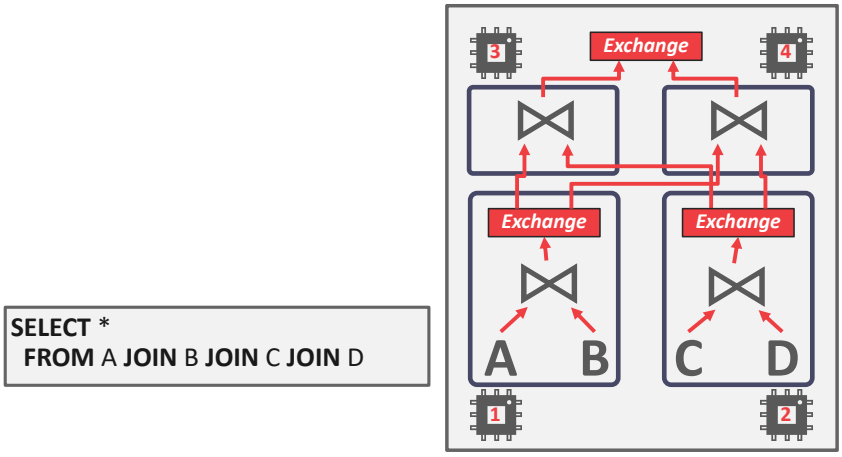

Figure 6: Bushy Parallelism – To perform a 4-way join on three tables, the query plan is divided into four fragments as shown. Different portions of the query plan run at the same time, similarly to inter-operator parallelism.

proaches to I/O parallelism are multi-disk parallelism, database partitioning, and partitioning.

Multi-Disk Parallelism

In *multi-disk parallelism*, the OS/hardware is configured to store the DBMS's files across multiple storage devices. This can be done through storage appliances and RAID configuration. All of the storage setup is transparent to the DBMS so workers cannot operate on different devices because the DBMS is unaware of the underlying parallelism.

Database Partitioning

In *database partitioning*, the database is split up into disjoint subsets that can be assigned to discrete disks. Some DBMSs allow for specification of the disk location of each individual database. This is easy to do at the file-system level if the DBMS stores each database in a separate directory. The log file of changes made is usually shared.

Logical Partitioning

The idea of *logical partitioning* is to split single logical table into disjoint physical segments that are stored/-managed separately. Such partitioning is ideally transparent to the application. That is, the application should be able to access logical tables without caring how things are stored.

The two approaches to partitioning are vertical and horizontal partitioning.

In *vertical partitioning,* a table's attributes are stored in a separate location (like a column store). The tuple information must be stored in order to reconstruct the original record.

In *horizontal partitioning*, the tuples of a table are divided into disjoint segments based on some partitioning keys. There are different ways to decide how to partition (e.g., hash, range, or predicate partitioning). The efficacy of each approach depends on the queries.