Lecture #14: Query Execution II

15-445/645 Database Systems (Fall 2026)
https://15445.courses.cs.cmu.edu/fall2026/
Carnegie Mellon University
Jignesh Patel

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 spreads the database over multiple resources. These resources may be computational (e.g., CPU cores, CPU sockets, GPUs, additional machines) or storage (e.g., disks, memory). Spreading the database over many resources allows it to:

- Deal with large data sets that don't fit on a single machine/node
- Have higher performance due to parallel processing
- Have a higher level of redundancy/fault-tolerance

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

2 Parallel vs Distributed Databases

It is important to distinguish between parallel and distributed systems:

- **Parallel DBMS** In a *parallel DBMS*, resources, or nodes, are physically close to each other. These nodes communicate over a high-speed interconnect. It is assumed that communication between resources is not only fast, but also cheap and reliable.
- **Distributed DBMS** In a *distributed DBMS*, resources may be far away from each other; this might mean the database spans racks or data centers in different parts of the world. As a result, resources communicate using a slower interconnect (often over a public network). Communication costs between nodes are higher 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, a SQL query executed against 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 one 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 two major process models that a DBMS may adopt: process per worker and thread per worker. A third common database usage pattern takes an embedded approach.

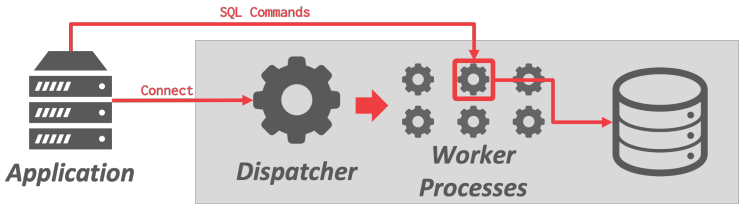


Figure 1: Process per Worker Model

Process per Worker

The most basic approach is *process per worker*. Here, each worker is a separate OS process, and thus relies on the OS scheduler. An application sends a request and opens a connection to the databases system. When some dispatcher receives this request, it selects one of its worker processes to manage the connection. The application then 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.

Relying on the operating system for scheduling effectively reduces the DBMS's control over execution. Further, this model depends on shared memory to maintain global data structures or relies on message passing, which has a higher overhead.

An advantage of the process per worker approach is that a process crash doesn't disrupt the whole system because each worker runs in the context of its own OS process.

This process model 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 shared by workers running in different processes.

Examples of systems that utilize the process-per-worker process model include IBM DB2, Postgres, and Oracle. When these DBMSs were developed, pthreads had not yet become the standard threading model. The semantics of threading varied from OS to OS while `fork()` was better defined.

Thread per Worker

The most common model nowadays 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 2.

Using multi-threaded architecture provides certain advantages. For one, there is less overhead per context switch. Additionally, a shared model does not have to be maintained. However, it is possible that a thread crash can kill the entire database process. Also, the thread per worker model does not necessarily imply that the DBMS supports intra-query parallelism.

Almost every DBMS created in the last 20 years uses this approach, including Microsoft SQL Server and MySQL. IBM DB2 and Oracle have updated their models to provide support for this approach, as well. Postgres and Postgres-derived databases largely still use the process-based approach.

Embedded DBMS

A very different usage pattern for databases involves running the system in the same address space of the application, as opposed to a client-server model where the database stands independent of the application. In this scenario, the application will set up the threads and tasks to run on the database system. The

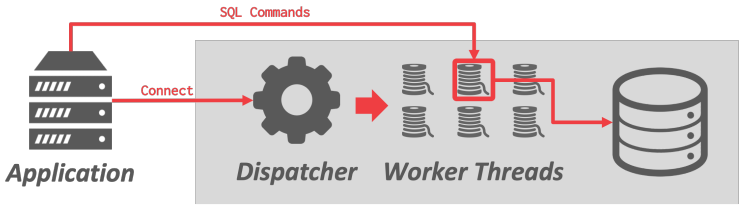


Figure 2: Thread per Worker Model

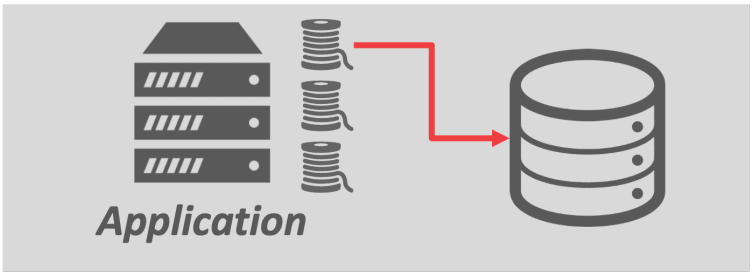


Figure 3: Embedded DBMS Scheduling

application itself will largely be responsible for scheduling. A diagram of an embedded DBMS's scheduling behaviors is shown in Figure 3.

DuckDB, SQLite, and RocksDB are the most famous embedded DBMSs.

Scheduling

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 cores 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 concurrently. Since 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 17.

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.

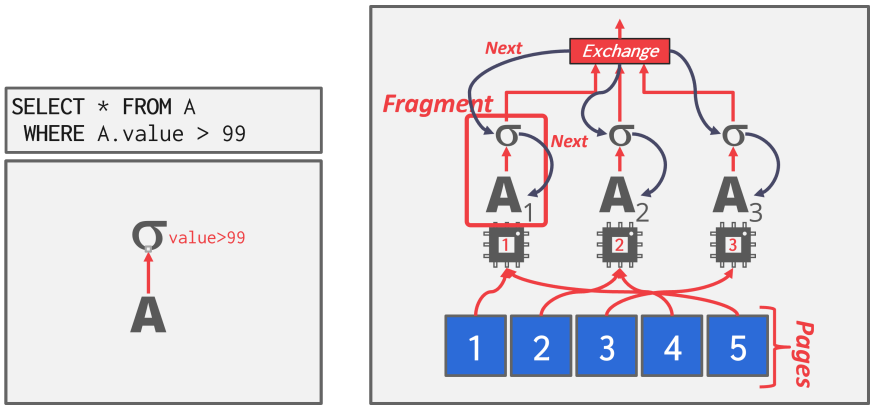


Figure 4: Intra-Operator Parallelism – The query plan for this SELECT is a sequential scan on A that is fed into a filter operator. To run this in parallel, the query plan is partitioned into disjoint fragments. A given plan fragment is operated on a by a distinct worker. The exchange operator calls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

Parallel algorithms exist for every relational operator. The DBMS 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. These approaches are not mutually exclusive. It is the DBMS' responsibility to combine these techniques in a way that optimizes performance on a given workload.

Intra-Operator Parallelism (Horizontal)

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

The DBMS inserts an *exchange* operator into the query plan to coalesce results from child operators. The 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.
- **Distribute:** Split a single input stream into multiple output streams.
- **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.

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.

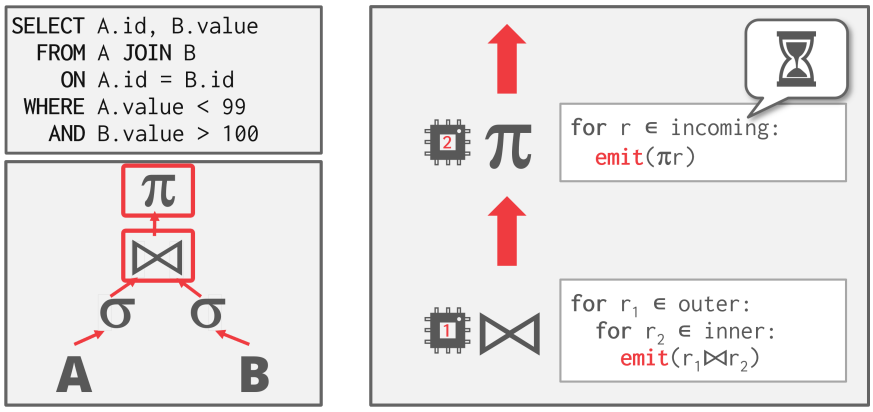


Figure 5: Inter-operator Parallelism – In the JOIN statement to the left, a single worker performs the join and then emits the result to another worker that performs the projection and then emits the result again.

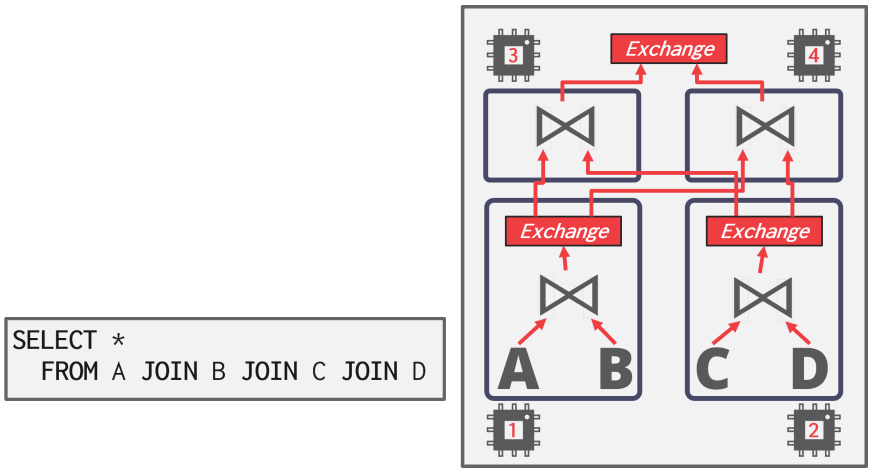


Figure 6: Bushy Parallelism – To perform a 4-way JOIN on three tables, the query plan is divided into four fragments as shown. Different portions of the query plan run at the same time, in a manner similar to inter-operator parallelism.

Bushy Parallelism

Bushy parallelism is a hybrid of intra-operator and inter-operator parallelism where workers execute multiple operators from different segments of the 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. Therefore, 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.* Two approaches to I/O parallelism are multi-disk parallelism and database 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 or RAID configuration based on performance, durability, and capacity constraints. 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.

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.

We will cover these approaches later in the semester when discussing distributed databases.