Lecture #13: Query Processing I

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
Andy Pavlo

Compilation - IR → Annotation Syntax Tree / Directed Acyclic Graph

1 Query Plan

The DBMS converts a SQL statement into a query plan. Operators in the query plan are arranged in a DAG, oftentimes a tree. Data flows from the leaves of this tree towards the root. The output of the root node in the tree is the result of the query. Typically operators are unary or binary (1 to 2 children). The same query plan can be executed in multiple ways.

A **pipeline** refers to a sequence of operators where tuples can continuously flow in-between each pair without intermediate storage. A **pipeline breaker** is an operator that cannot finish until all of its children have emit their tuples. Some examples include Joins (build Side), Subqueries and Order By.

2 Processing Models

A DBMS's *processing model* defines how the system executes a query plan. It specifies things like the direction in which the query plan is evaluated and what kind of data is passed between operators along the way. There are several types of processing models with trade-offs for different workloads (For eg: OLTP vs OLAP).

Each processing model is comprised of two types of execution paths: control flow and data flow. **control flow** dictates how the DBMS invokes the operator, while **data flow** decides how an operator sends its results. An operator's output can be either whole tuples (NSM) or subsets of columns (DSM).

The three execution models that we consider are:

- Iterator Model
- Materialization Model
- Vectorized / Batch Model

Iterator Model ← Most Common

The *iterator model*, also known as the Volcano or Pipeline model, is the most common processing model. It is used by almost every (row-based) DBMS.

The iterator model implements a `Next` function for every operator. Nodes in the query plan, except for leaf nodes, call `Next` on its children. When called, children nodes emit tuples one by one to their parent nodes until there's none. Each tuple is then processed and passed up the plan as far as possible before the next one is retrieved. This design is efficient for disk-based systems, as it maximizes use of in-memory data before accessing new tuples or pages. A sample diagram is shown in Figure 1.

Query plan operators in an iterator model are highly composible and easy to reason about. This is because each operator can be implemented independently from their parent or child operators in the query plan tree as long as it implements a `Next` function as follows:

- On each call to `Next`, the operator returns either a single tuple or a null marker if there are no more tuples to emit.
- The operator implements a loop that calls `Next` on its children to retrieve their tuples and then process them. In this way, one node's `Next` calls will trigger `Next` calls on its child. In response, the child node will return the next tuple that the parent must process.

The iterator model allows for *pipelining* where the DBMS can process a tuple through as many operators as possible before having to retrieve the next tuple. The series of tasks performed for a given tuple in the query plan is called a *pipeline*.

Some operators will block until children emit all of their tuples. Examples of such operators include joins, subqueries, and ordering (`ORDER BY`). Such operators are known as *pipeline breakers*.

Output control works easily with this approach (`LIMIT`) because an operator can stop invoking `Next` on its child (or children) operator(s) once it has all the tuples that it requires.

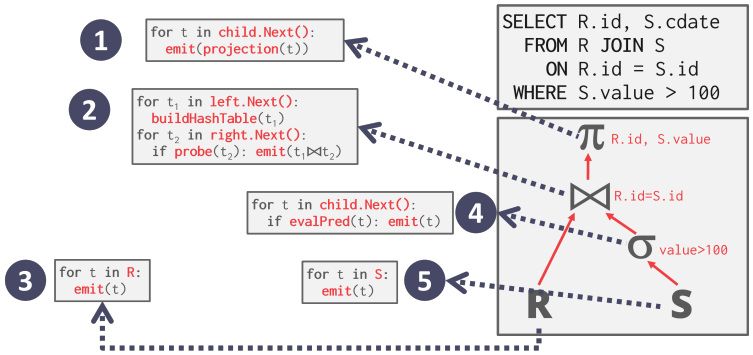

Figure 1: Iterator Model Example – Pseudo code of the different `Next` functions for each of the operators. The `Next` functions are essentially `for`-loops that iterate over the output of their child operator. For example, the root node calls `Next` on its child, the join operator, which is an access method that loops over the relation R and emits a tuple up that is then operated on. After all tuples have been processed, a null pointer (or another indicator) is sent that lets the parent nodes know to move on.

Materialization Model

The *materialization model* is a specialization of the iterator model where each operator processes its input all at once and then emits its output all at once. Instead of having a next function that returns a single tuple, each operator returns all of its tuples every time it is reached. To avoid scanning too many tuples, the DBMS can propagate down information about how many tuples are needed for subsequent operators (e.g. `LIMIT`). The operator "materializes" its output as a single result. The output can be either a whole tuple (NSM) or a subset of columns (DSM). A diagram of the materialization model is shown in Figure 2.

Every query plan operator implements an `Output` function:

- The operator processes all the tuples from its children at once.
- The return result of this function is all the tuples that operator will ever emit. When the operator finishes executing, the DBMS never needs to return to it to retrieve more data.

The materialization model is better for OLTP workloads where queries typically only access a small number of tuples at a time. Thus, there are fewer function calls to retrieve tuples. It is however not suited for OLAP

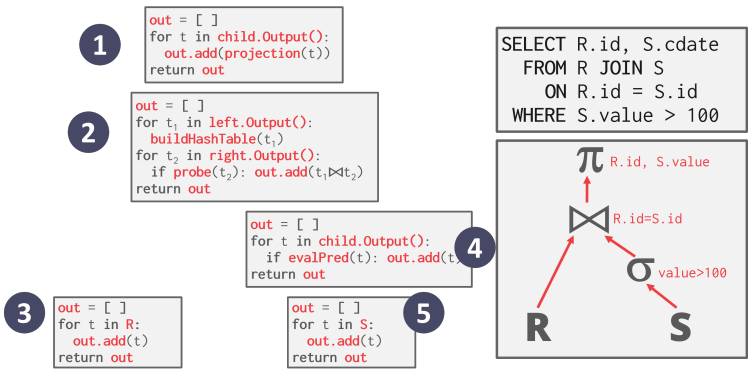

Figure 2: Materialization Model Example – Starting at the root, the `child.Output()` function is called, which invokes the operators below, which returns all tuples back up.

queries with large intermediate results because the DBMS may have to spill those results to disk between operators.

Vectorization Model ← Common

Like the iterator model, each operator in the *vectorization model* implements a `Next` function. However, each operator emits a *batch* (i.e. vector) of data instead of a single tuple. The operator's internal loop implementation is optimized for processing batches of data instead of a single item at a time. The size of the batch can vary based on hardware or query properties. See Figure 3 for an example of the vectorization model.

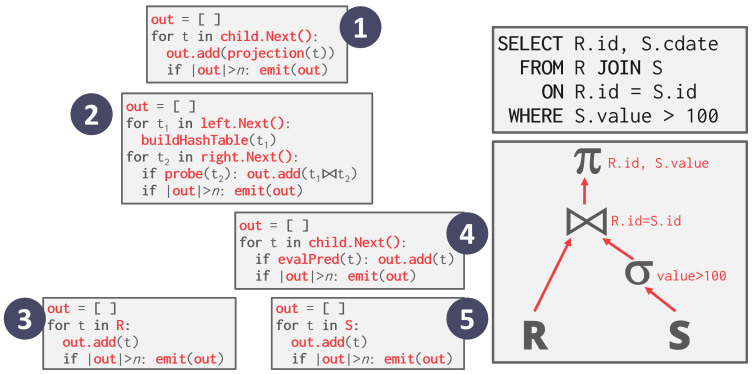

Figure 3: Vectorization Model Example – The vectorization model is very similar to the iterator model except at every operator, an output buffer is compared to the desired emission size. If the buffered output size is larger than the emission size, the buffered tuple batch will be sent up.

The vectorization model approach is ideal for OLAP queries that have to scan a large number of tuples because there are fewer invocations of the `Next` function.

The vectorization model allows operators to more easily use vectorized (SIMD) instructions to process batches of tuples. Modern out-of-order CPUS also work efficiently processing tuple batches. This due to the operators working in tight for-loops over arrays of equal-sized items and having no data or control dependencies.

Processing Direction

The models are implemented to invoke the operators either from **top-to-bottom (pull)** or from **bottom-to-top (push)**. Although the top-to-bottom approach is much more common, the bottom-to-top approach can allow for tighter control of caches/registers in *pipelines*.

- **Approach #1: Top-to-Bottom** ← Most Common
  - Start with the root and "pull" data from children to parents
  - Tuples are always passed with function calls
  - Easy to control via LIMIT
  - Parent operator has to block until its child returns with a tuple
  - Additional overhead from `Next` functions implemented as virtual functions
  - Each `Next` call can involve branching costs
- **Approach #2: Bottom-to-Top** ← Rare
  - Start with leaf nodes and "push" data from children to parents
  - Allows for tighter control of caches / registers in operator pipelines Scheduler
  - Might not have exact control of intermediate result sizes
  - Some iterator might be hard to implement in this design.

3 Access Methods

An *access method* is how the DBMS accesses the data stored in a table. These methods do not have corresponding operators defined in relational algebra because physical retrieval of data is not represented in relational algebra. There are three basic approaches: sequential scan, index scan (with many variants), and multi-index scans.

Sequential Scan

The sequential scan operator iterates over every page in the table and retrieves it from the buffer pool. As the scan iterates over all the tuples on each page, it evaluates the predicate to decide whether or not to emit the tuple to the next operator. The DBMS maintains an internal cursor to track the last examined page/slot.

A sequential table scan is almost always the least efficient method by which a DBMS may execute a query. There are a number of optimizations available to help make sequential scans faster:

- **Compression:** Compression schemes such as RLE (run length encoding) can help retrieve multiple tuples in a single fetch. *Lec#6 - Columnar Compression*
- **Prefetching:** Fetch the next few pages in advance so that the DBMS does not have to block on storage I/O when accessing each page. *Lec#5 - Bufferpool Optimization*
- **Buffer Pool Bypass:** The scan operator stores pages that it fetches from disk in its local memory instead of the buffer pool in order to avoid sequential flooding.
- *Lec#14* **Parallelization:** Execute the scan using multiple threads/processes in parallel.
- **Late Materialization:** DSM DBMSs can delay stitching together tuples until reaching the upper parts of the query plan. This allows each operator to pass the minimal amount of information needed to the next operator (e.g. record ID, offset to record in column). This is only useful in column-store systems. *Lec#11+12*
- *Lec#8* **Heap Clustering:** Tuples are stored in the heap pages using an order specified by a clustering index.
- **Result Caching/Materialized Views:** Storing (caching) the results of subquery/query which are more frequently ocurring.
- *Lec#14* **Code specialization/compilation:** Pre compiling functions ahead of time (JIT) in order to obtain

results faster when it's actually required.

Data Skipping

- **Approximate Queries (Lossy Data Skipping):** Execute queries on a sampled subset of the entire table to produce approximate results. This is typically done for computing aggregations in a scenario that allow a low error to produce a nearly accurate answer.
- **Zone Map (Lossless Data Skipping):** Pre-compute aggregations for each tuple attribute in a page. The DBMS can then decide whether it needs to access a page by checking its Zone Map first. The Zone Maps for each page are stored in separate pages and there are typically multiple entries in each Zone Map page. Thus, it is possible to reduce the total number of pages examined in a sequential scan. Zone maps are particularly valuable in the cloud database systems where data transfer over a network incurs a bigger cost. See Figure 4 for an example of a Zone Map.

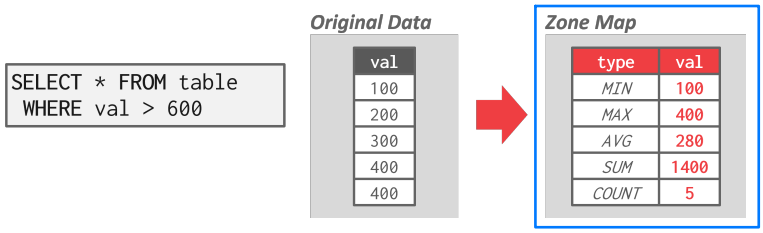

Figure 4: Zone Map Example – The zone map stores pre-computed aggregates for values in a page. In the example above, the select query realizes from the zone map that the max value in the original data is only 400. Then, instead of having to iterate through every tuple in the page, the query can avoid accessing the page at all since none of the values will be greater than 600.

The limitations of the sequential model include function overhead and lack of parallelization (e.g. not able to take advantage of vector operations).

Index Scan

In an *index scan*, the DBMS picks an index to find the tuples that a query needs.

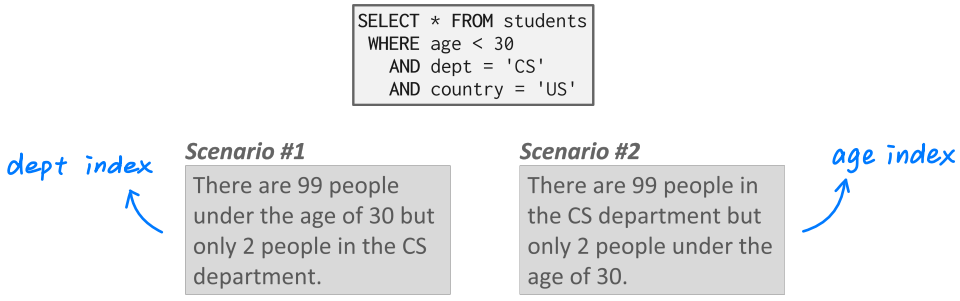

Figure 5: Index Scan Example – Consider a single table with 100 tuples and two indexes: `age` and `department`. In the first scenario, it is better to use the `department` index in the scan because it only has two tuples to match. Choosing the `age` index would not be much better than a simple sequential scan. In the second scenario, the `age` index would eliminate more unnecessary scans and is the optimal choice.

Lec#15

There are many factors involved in the DBMSs' index selection process, including:

- What attributes the index contains;
- What attributes the query references;

- The attribute's value domains;
- Predicate composition;
- Whether the index has unique or non-unique keys.

A simple example of an index scan is shown in Figure 5.

Multi Index Scan

More advanced DBMSs support multi-index scans. When using multiple indexes for a query, the DBMS computes sets of record IDs using each matching index, combines these sets based on the query's predicates, and retrieves the records and apply any predicates that may remain. The DBMS can use bitmaps, hash tables, or Bloom filters to compute record IDs through set intersection. See Figure 6 for an example that makes use of a multi-index scan.

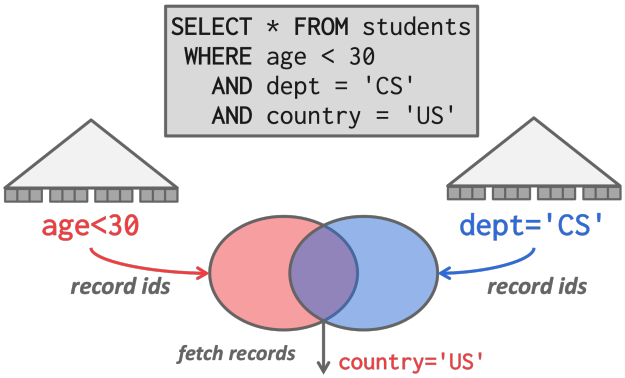

Figure 6: Multi-Index Scan Example – Consider the same table in Figure 5. With multi-index scan support, we first compute the sets of record IDs satisfying the predicate for `age` and `dept` respectively, using the corresponding index. We then compute the intersection of the two sets, fetch the corresponding records, and apply the remaining predicate `country='US'`.

4 Modification Queries

Operators that modify the database (`INSERT`, `UPDATE`, `DELETE`) are responsible for checking constraints and updating indexes. For `UPDATE/DELETE`, child operators pass Record IDs for target tuples and must keep track of previously seen tuples.

There are two implementation choices on how to handle `INSERT` operators:

- **Choice #1:** Materialize tuples inside of the operator. → Materialization Model
- **Choice #2:** Operator inserts any tuple passed in from child operators. → Iterator Model

Halloween Problem - also known as the Update Query Problem

The Halloween Problem is an anomaly in which an update operation changes the physical location of a tuple, causing a scan operator to visit the tuple multiple times. This can occur on clustered tables or index scans.

This phenomenon was originally discovered by IBM researchers while building **System R** on Halloween day in 1976. The solution to this problem is to keep track of the modified record IDs for each query.

5 Expression Evaluation

The DBMS represents a WHERE clause as an *expression tree* (see Figure 7 for an example). The nodes in the tree represent different expression types.

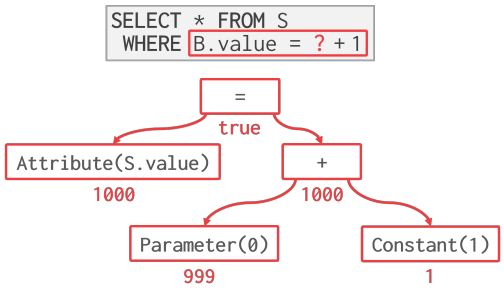

Figure 7: Expression Evaluation Example – A WHERE clause and a diagram of its corresponding expression.

Some examples of expression types that can be stored in tree nodes:

- Comparisons (=, <, >, !=);
- Conjunction (AND), Disjunction (OR);
- Arithmetic Operators (+, -, *, /, %);
- Constant and Parameter Values;
- Tuple Attribute References.

To evaluate an expression tree at runtime, the DBMS maintains a context handle that contains metadata for the execution, such as the current tuple, the parameters, and the table schema. The DBMS then walks the tree to evaluate its operators and produce a result.

Evaluating predicates in this manner is slow for the CPU because the DBMS must traverse the entire tree and determine the correct action to take for each operator. A better approach is to just evaluate the expression directly (think JIT compilation). Based on a internal cost model, the DBMS would determine whether code generation will be adopted to accelerate a query.

A even better approach is to **vectorize** it to evaluate batch of tuples at the same time.

There are ways to optimize expression evaluations:

- **Constant folding:** Reducing the overhead by identifying operations that can be performed just once (for example: UPPER on a constant value) and using it's results rather than performing it every single time.
- **Sub Expression limitation:** Identifying repeated subexpressions in an expression tree and computing its result once to reuse it for all occurences in the plan.

Lecture #14: Query Execution II

15-445/645 Database Systems (Fall 2025)
https://15445.courses.cs.cmu.edu/fall2025/
Carnegie Mellon University
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 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.

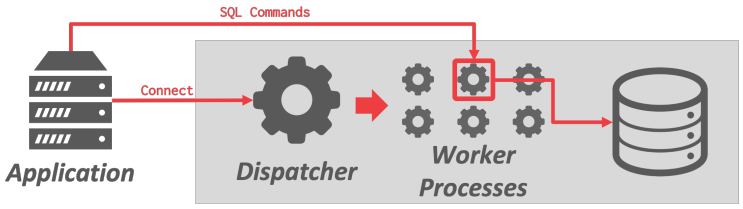

Figure 1: Process per Worker Model

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.

Pros

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.

Cons

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.

historical problem

Thread per Worker

← Most Common

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.

Pros

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.

Cons

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

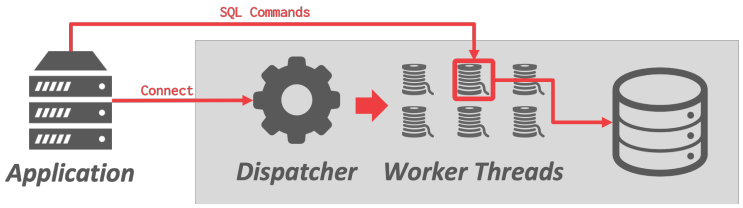

Figure 2: Thread per Worker Model

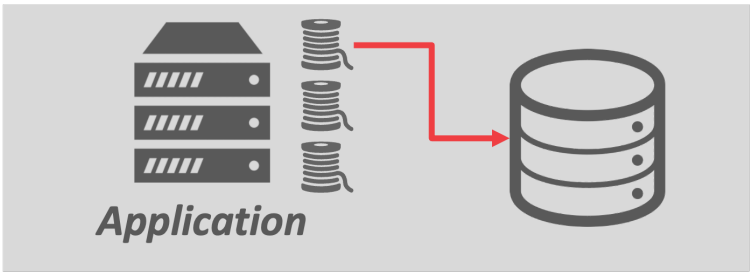

Figure 3: Embedded DBMS Scheduling

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.

The OS is not your friend!

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.

e.g. FCFS

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.

Lec#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.

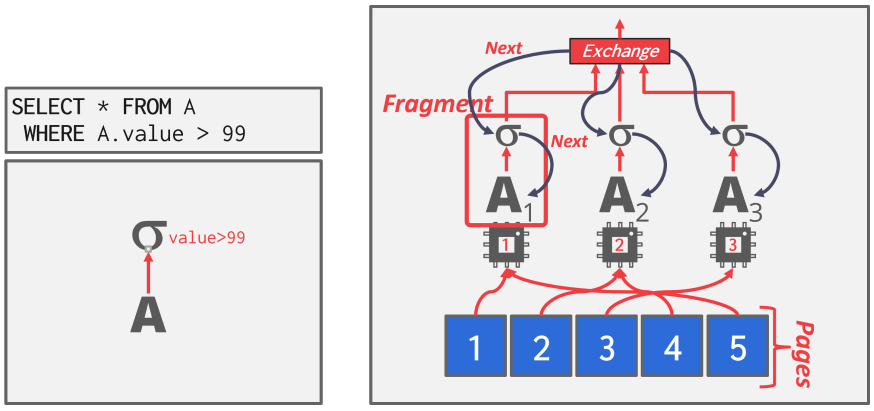

Figure 4: Intra-Operator Parallelism – The query plan for this SELECT is a sequential scan on A that is fed into a filter operator. To run this in parallel, the query plan is partitioned into disjoint fragments. A given plan fragment is operated on a by a distinct worker. The exchange operator calls 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) ← Most Common

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) ← Less Common

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.

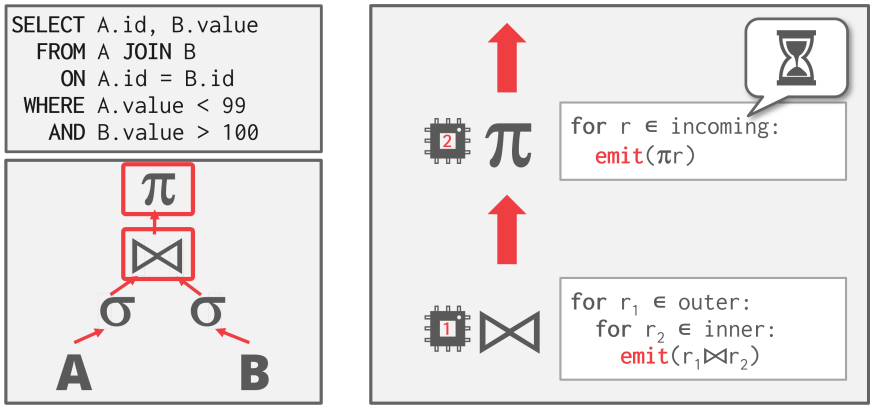

Figure 5: Inter-operator Parallelism – In the JOIN statement to the left, a single worker performs the join and then emits the result to another worker that performs the projection and then emits the result again.

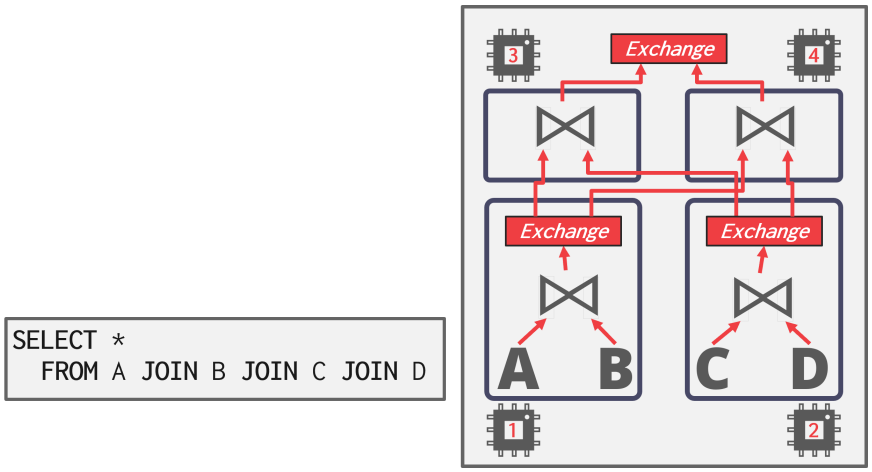

Figure 6: Bushy Parallelism – To perform a 4-way JOIN on three tables, the query plan is divided into four fragments as shown. Different portions of the query plan run at the same time, in a manner similar to inter-operator parallelism.

Bushy Parallelism ← High-end Systems

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.

→ RAID 0: Striping
→ RAID 1: Mirroring

Database Partitioning

(1)

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.

(2)

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.