Query Execution (Part 2)

Recap

Processing Model

- A DBMS's **processing model** defines how the system executes a query plan.
  - Different trade-offs for different workloads.
- **Approach 1:** Iterator Model
- **Approach 2:** Materialization Model
- **Approach 3:** Vectorized / Batch Model

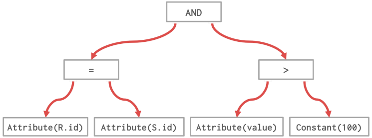
Multi-Index Scan

- If there are multiple indexes that the DBMS can use for a query:
  - Compute sets of record ids using each matching index.
  - Combine these sets based on the query's predicates (union vs. intersect).
  - Retrieve the records and apply any remaining predicates.
- Postgres calls this **Bitmap Scan**.

Expression Evaluation

- The DBMS represents a WHERE clause as an **expression tree**.
- The nodes in the tree represent different expression types:
  - Comparisons (=, <, >, !=)
  - Conjunction (AND), Disjunction (OR)
  - Arithmetic Operators (+, -, *, /, %)
  - Constant Values
  - Tuple Attribute References

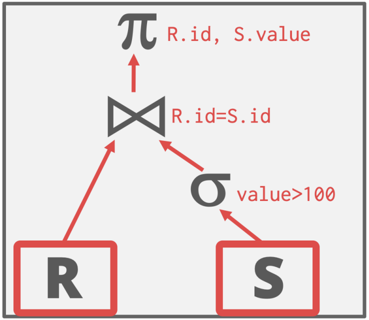
```
SELECT R.id, S.cdate
  FROM R, S
  WHERE R.id = S.id  AND S.value > 100
```



Query Execution

- We discussed last class how to compose operators together to execute a query plan.
- We assumed that the queries execute with a single worker (*e.g.*, thread).
- We now need to talk about how to execute with multiple workers.

```
SELECT R.id, S.cdate
  FROM R, S
  WHERE R.id = S.id AND S.value > 100
```



Today's Agenda

- Overview
- Process Models
- Execution Parallelism
- I/O Parallelism

Overview

Why care about Parallel Execution?

- Increased performance.
  - Throughput
  - Latency
- Increased responsiveness and availability.
- Potentially lower **total cost of ownership** (TCO).

Parallel vs. Distributed

- Database is spread out across multiple **resources** to improve different aspects of the DBMS.
- Appears as a single database instance to the application.
  - SQL query for a single-resource DBMS should generate same result on a parallel or distributed DBMS.

Parallel vs. Distributed

- **Parallel DBMSs:**
  - Resources are physically close to each other.
  - Resources communicate with high-speed interconnect.
  - Communication is assumed to cheap and reliable.
  - Typically rely on **shared memory**.
- **Distributed DBMSs:**
  - Resources can be far from each other.
  - Resources communicate using slow(er) interconnect.
  - Communication cost and problems cannot be ignored.
  - Typically rely on **message passing**.

Process Model

Process Model

- A DBMS's **process model** defines how the system is architected to support concurrent requests from a multi-user application.
- A **worker** is the DBMS component running on the **server** that is responsible for executing tasks on behalf of the **client** and returning the results.

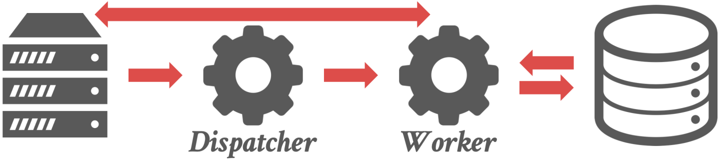
Process Models

- **Approach 1:** Process per DBMS Worker
- **Approach 2:** Process Pool
- **Approach 3:** Thread per DBMS Worker

Process per DBMS Worker

- Each worker is a separate OS process.
  - Relies on OS scheduler.
  - Use shared-memory for global data structures.
  - A process crash doesn't take down entire system.
  - Examples: IBM DB2, Postgres, Oracle

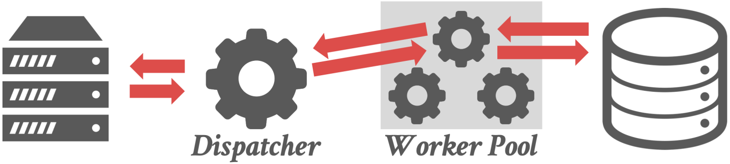
Process per DBMS Worker



Process Pool

- A worker uses any process that is free in a pool
  - Still relies on OS scheduler and shared memory.
  - Bad for CPU cache locality.
  - Examples: IBM DB2, Postgres (2015)

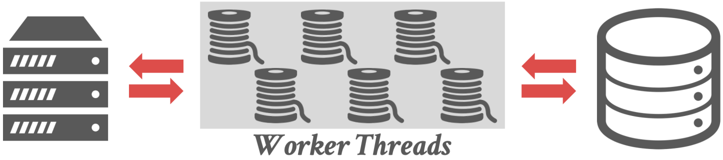
Process Pool



Thread per DBMS Worker

- Single process with multiple worker threads.
  - DBMS manages its own scheduling.
  - May or may not use a dispatcher thread.
  - Thread crash (may) kill the entire system.
  - Examples: IBM DB2, MSSQL, MySQL, Oracle (2014)

Thread per DBMS Worker



Process Models

- Using a multi-threaded architecture has several advantages:
  - Less overhead per context switch.
  - Do not have to manage shared memory.
- The thread per worker model does **not** mean that the DBMS supports intra-query parallelism.
- Most DBMSs in the last decade use threads (unless they are Postgres forks).

Scheduling

- For each query plan, the DBMS decides where, when, and how to execute it.
  - 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?
- The DBMS always knows more than the OS.

Execution Parallelism

Inter- VS. Intra-Query Parallelism

- **Inter-Query:** Different queries are executed concurrently.
  - Increases throughput & reduces latency.
- **Intra-Query:** Execute the operations of a single query in parallel.
  - Decreases latency for long-running queries.

Inter-Query Parallelism

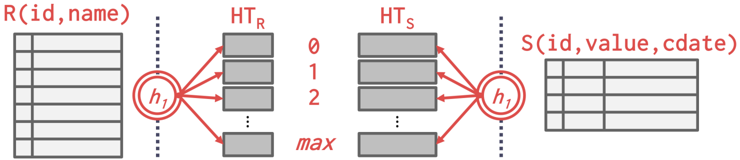
- Improves overall performance by allowing multiple queries to execute simultaneously.
- If queries are **read-only**, then this requires little coordination between queries.
- If multiple queries are **updating** the database at the same time, then this is hard to do correctly.
- ACID: Isolation of concurrent workers to ensure correctness.

Inta-Query Parallelism

- Improve the performance of a single query by executing its operators in parallel.
- Think of organization of operators in terms of a **producer/consumer** paradigm.
- There are parallel algorithms for every relational operator.
  - Can either have multiple threads access centralized data structures in a **synchronized manner** or use **partitioning** to divide work up.

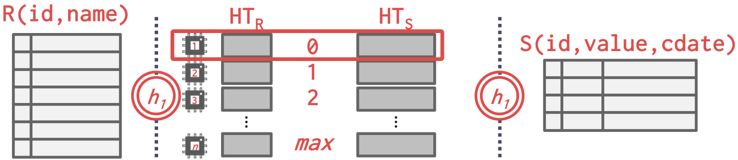
Parallel Grace Hash Join

- Use a **separate worker** to perform the join for each level of buckets for R and S after partitioning.



Parallel Grace Hash Join

- Use a **separate worker** to perform the join for each level of buckets for R and S after partitioning.



Intra-Query Parallelism

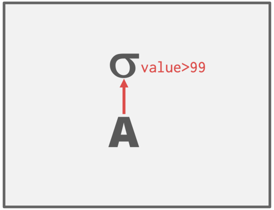
- **Approach 1:** Intra-Operator (Horizontal)
- **Approach 2:** Inter-Operator (Vertical)
- **Approach 3:** Bushy

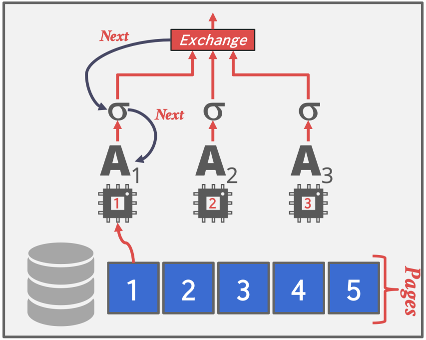
Intra-Operator Parallelism

- **Intra-Operator (Horizontal)**
  - Decompose operators into independent **fragments** 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.
- Exchange operator encapsulates parallelism and data transfer.

Intra-Operator Parallelism

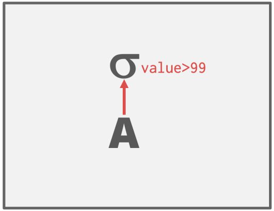
```
SELECT * FROM A
 WHERE A.value > 99
```

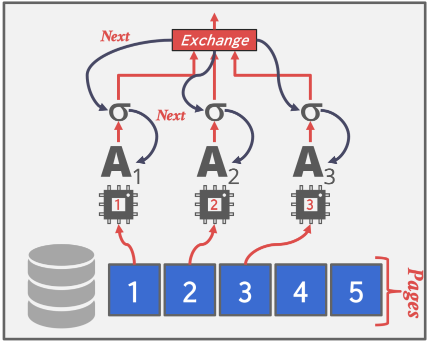




Intra-Operator Parallelism

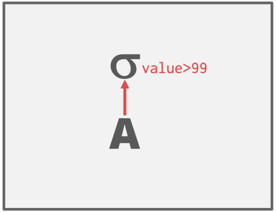
```
SELECT * FROM A
 WHERE A.value > 99
```

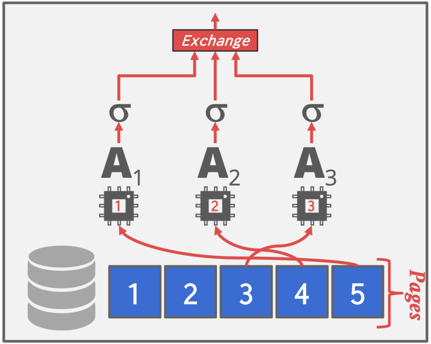




Intra-Operator Parallelism

```
SELECT * FROM A
 WHERE A.value > 99
```



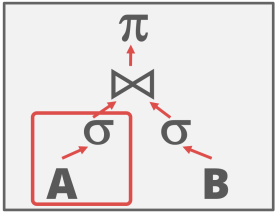


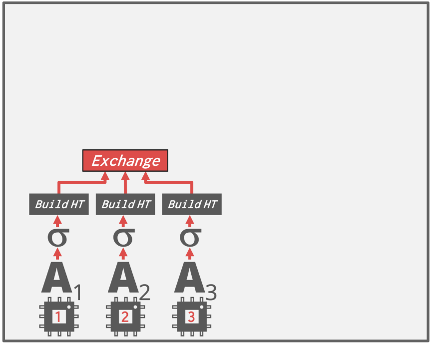
Exchange Operator

- **Exchange Type 1 – Gather**
  - Combine the results from multiple workers into a single output stream.
  - Query plan root must always be a gather exchange.
  - N input pipelines, 1 output pipeline.
- **Exchange Type 2 – Repartition**
  - Reorganize multiple input streams across multiple output streams.
  - N input pipelines, M output pipelines.
- **Exchange Type 3 – Distribute**
  - Split a single input stream into multiple output streams.
  - 1 input pipeline, M output pipelines.

Intra-Operator Parallelism

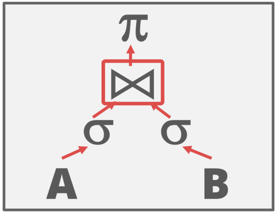
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

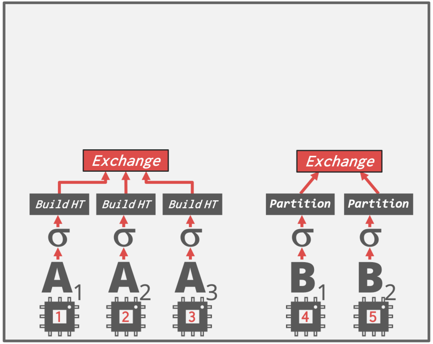




Intra-Operator Parallelism

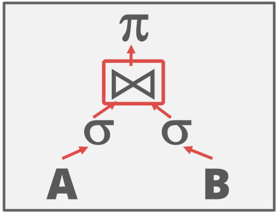
```sql
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

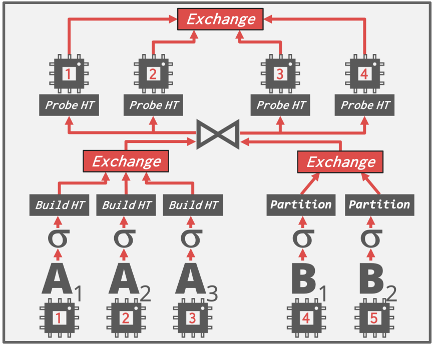




Intra-Operator Parallelism

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



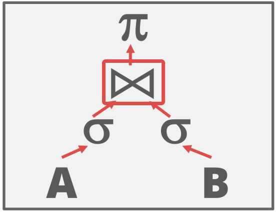


Inter-Operator Parallelism

- **Inter-Operator (Vertical)**
  - Operations are overlapped in order to pipeline data from one **stage** to the next without materialization.
- Also called **pipelined parallelism**.

Inter-Operator Parallelism

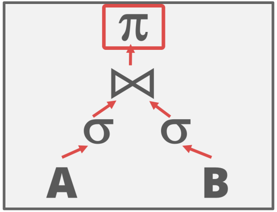
```sql
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

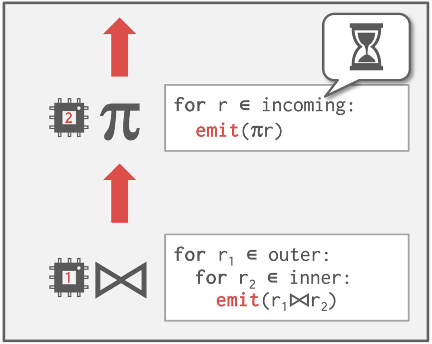



```
for r1 ∊ outer:
  for r2 ∊ inner:
    emit(r1⋈r2)
```

Inter-Operator Parallelism

```sql
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

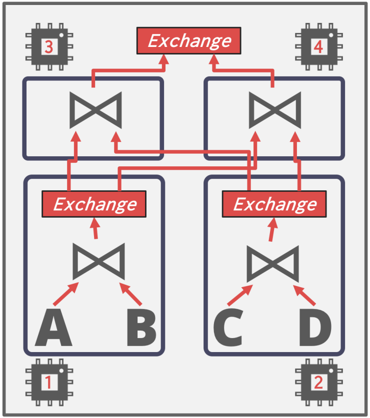




Bushy Parallelism

- **Bushy Parallelism**
  - Extension of inter-operator parallelism where workers execute multiple operators from different segments of a query plan **at the same time**.
  - Still need exchange operators to combine intermediate results from segments.

```
SELECT *
        FROM A JOIN B JOIN C JOIN D
```



I/O Parallelism

Observation

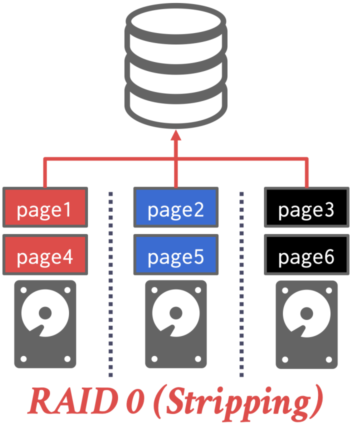
- Using additional processes/threads to execute queries in parallel won't help if the disk is always the main bottleneck.
  - Can make things worse if each worker is reading different segments of disk.

I/O Parallelism

- Split the DBMS installation across multiple storage devices.
  - Multiple Disks per Database
  - One Database per Disk
  - One Relation per Disk
  - Split Relation across Multiple Disks

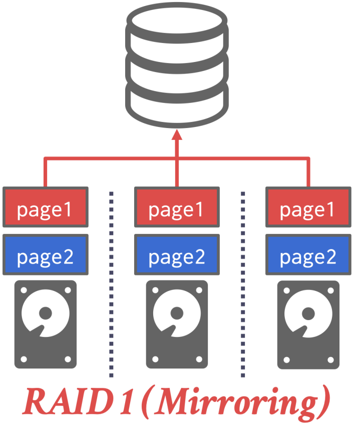
Multi-Disk Parallelism

- Configure OS/hardware to store the DBMS's files across multiple storage devices.
  - Storage Appliances
  - RAID Configuration
- This is transparent to the DBMS.



Multi-Disk Parallelism

- Configure OS/hardware to store the DBMS's files across multiple storage devices.
  - Storage Appliances
  - RAID Configuration
- This is transparent to the DBMS.



Database Partitioning

- Some DBMSs allow you specify the disk location of each individual database.
  - The buffer pool manager maps a page to a disk location.
- This is also easy to do at the filesystem level if the DBMS stores each database in a separate directory.
  - The log file might be shared though

Database Partitioning

- Split single logical table into disjoint physical segments that are stored/managed separately.
- Ideally partitioning is transparent to the application.
  - The application accesses logical tables and does not care how things are stored.
  - Not always true in distributed DBMSs.

Vertical Relation Partitioning

- Store a table's attributes in a separate location (e.g., file, disk volume).
- Have to store tuple information to reconstruct the original record.

```sql
CREATE TABLE foo (
        attr1 INT,
        attr2 INT,
        attr3 INT,
        attr4 TEXT
);
```

Partition #1

Tuple#1	attr1	attr2	attr3
Tuple#2	attr1	attr2	attr3
Tuple#3	attr1	attr2	attr3
Tuple#4	attr1	attr2	attr3

Partition #2

Tuple#1	attr4
Tuple#2	attr4
Tuple#3	attr4
Tuple#4	attr4

Horizontal Relation Partitioning

- Divide the tuples of a table up into disjoint segments based on some partitioning key.
  - Hash Partitioning
  - Range Partitioning
  - Predicate Partitioning

```
CREATE TABLE foo (
        attr1 INT,
        attr2 INT,
        attr3 INT,
        attr4 TEXT
);
```

Partition #1

Tuple#1	attr1	attr2	attr3	attr4
Tuple#2	attr1	attr2	attr3	attr4

Partition #2

Tuple#3	attr1	attr2	attr3	attr4
Tuple#4	attr1	attr2	attr3	attr4

Conclusion

- Parallel execution is important.
- (Almost) every DBMS supports this.
- This is really hard to get right.
  - Coordination Overhead
  - Scheduling
  - Concurrency Issues
  - Resource Contention
- Next Class
  - Scheduling