

COMP 421: Files & Databases

Lecture 14: Query Execution 2: Electric Boogaloo

Announcements

Project 2: B+Tree is due 11/3. That is soon.

Exam handback/discussion in the last 15 minutes of class today

- Exam grades
- Mid-semester grades

Exam question review Wednesday after class

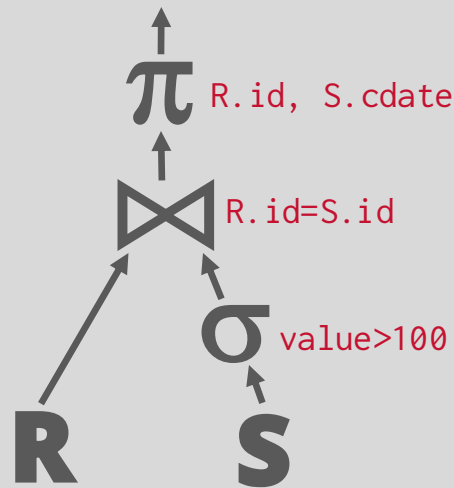
Query Execution 1

We discussed composing operators into a plan to execute a query.

We assumed that queries execute with a single worker (e.g., a thread).

We will now discuss how to execute queries in parallel using multiple workers.

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



Parallel Query Execution

The database is spread across multiple resources to

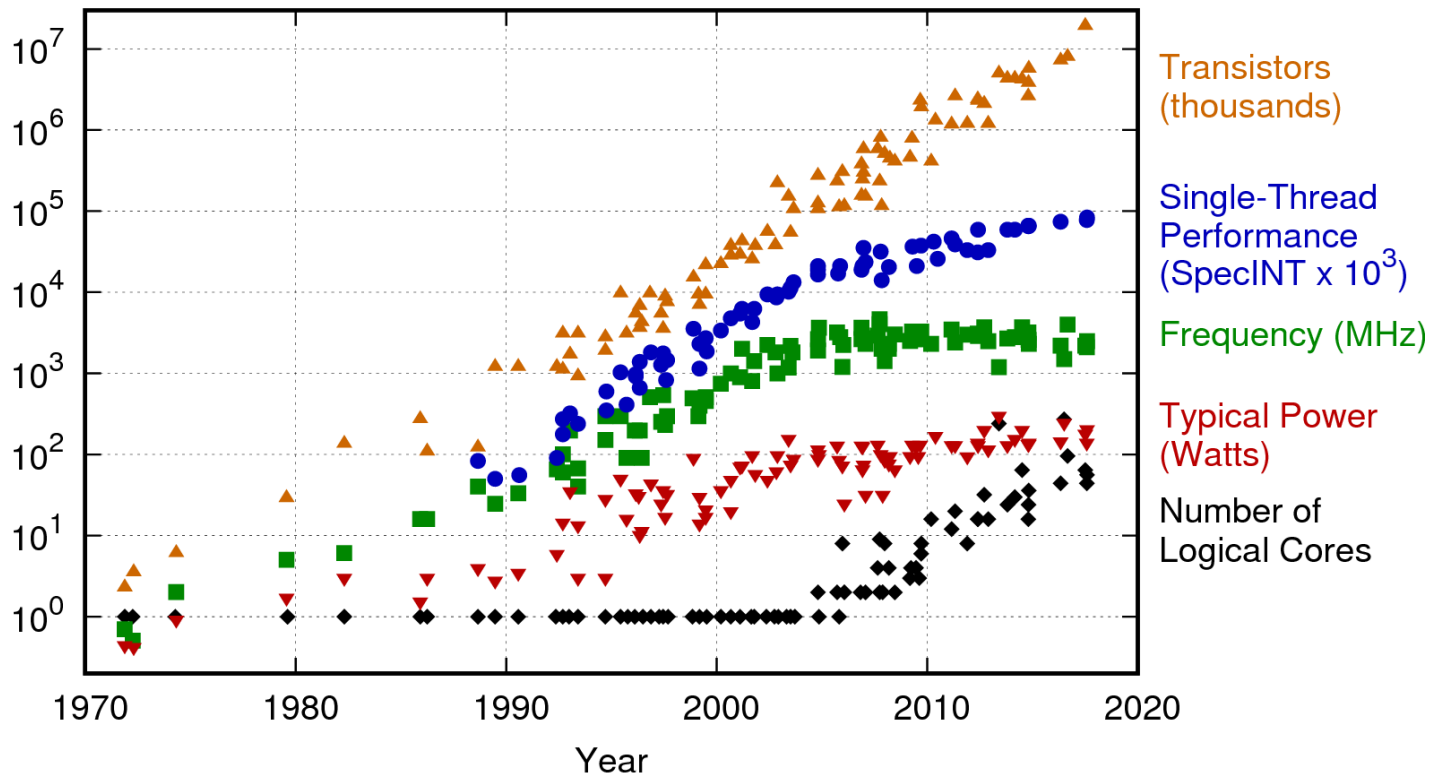
- Deal with large data sets that don't fit on a single machine/node
- Higher performance
- Redundancy/Fault-tolerance

Appears as a single logical database instance to the application, regardless of physical organization.

- SQL query for a single-resource DBMS should generate the same result on a parallel or distributed DBMS.

Gimme Gimme Moore

42 Years of Microprocessor Trend Data



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

Parallel vs. Distributed

Parallel DBMSs

- Resources are physically close to each other.
- Resources communicate over high-speed interconnect.
- Communication is assumed to be cheap and reliable.

Distributed DBMSs

- Resources can be far from each other.
- Resources communicate using slow(er) interconnect.
- Communication costs and problems cannot be ignored.

Today's Agenda

Process Models

Execution Parallelism

I/O Parallelism

Process Model

A DBMS's **process model** defines how the system is architected to support concurrent requests / queries.

A **worker** is the DBMS component responsible for executing tasks on behalf of the client and returning the results.

Process Model

Approach #1: **Process** per DBMS Worker

Approach #2: **Thread** per DBMS Worker

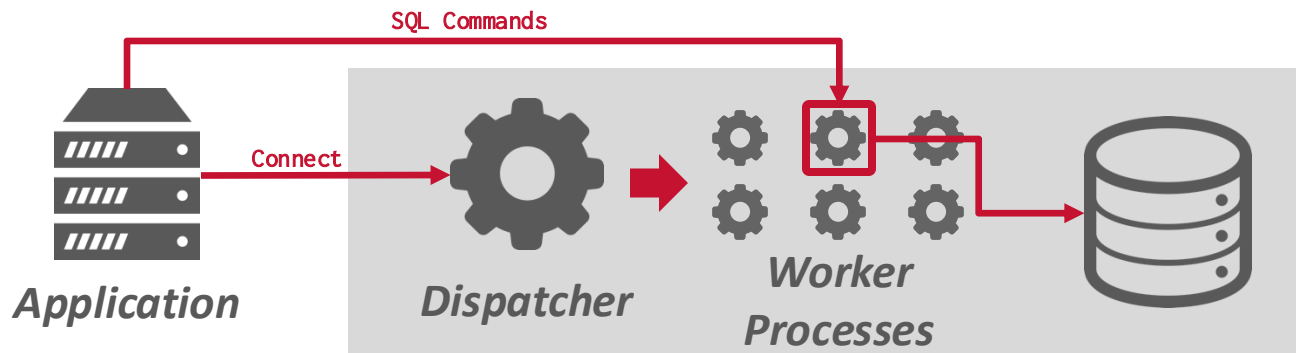
← *Most Common*

Approach #3: **Embedded** DBMS

Process Per Worker

Each worker is a separate OS process.

- Relies on the OS dispatcher.
- Use shared-memory for global data structures.
- A process crash does not take down the entire system.
- Examples: IBM DB2, Postgres, Oracle

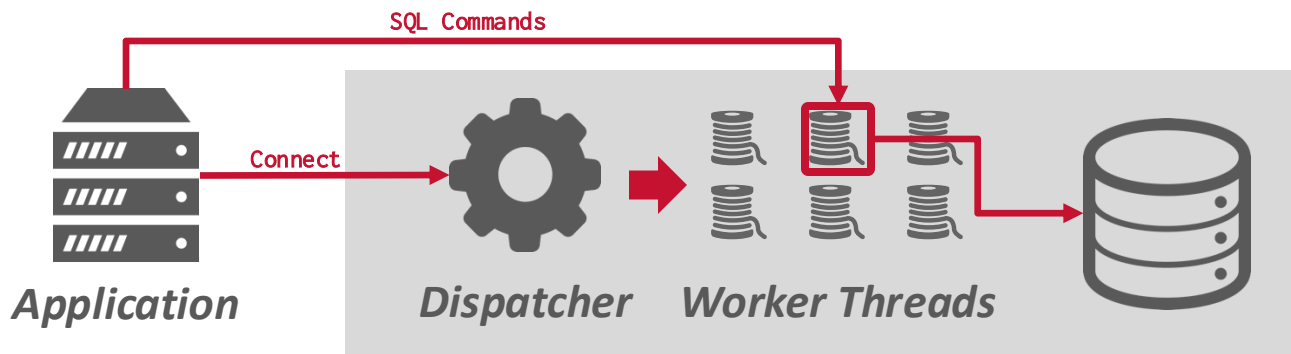


Thread Per Worker

Single process with multiple worker threads.

- DBMS (mostly) manages its own scheduling.
- May or may not use a dispatcher thread.
- Thread crash (may) kill the entire system.
- Examples: MSSQL, MySQL, DB2, [Oracle \(2014\)](#)

Almost every DBMS created in the last 20 years!

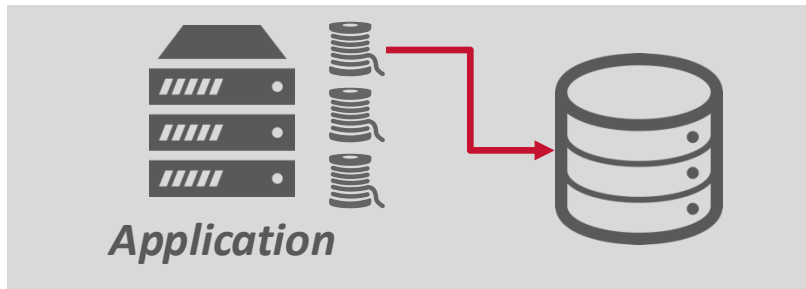


Embedded DBMS

DBMS runs inside the same address space as the application. Application is (primarily) responsible for threads and scheduling.

The application may support outside connections.

→ Examples: BerkeleyDB, SQLite, RocksDB, LevelDB



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 nearly ***always*** knows more than the OS.

Process Models

Advantages of a multi-threaded architecture:

- Less overhead per context switch.
- Do not have to manage shared memory / IPC.

The thread per worker model does not mean that the DBMS supports intra-query parallelism.

DBMS from the last 15 years use native OS threads unless they are Redis or Postgres forks.

Parallel Execution

The DBMS executes multiple tasks simultaneously to improve hardware utilization.

- Active tasks do not need to belong to the same query.
- High-level approaches do not vary on whether the DBMS is multi-threaded, multi-process, or multi-node.

Approach #1: Inter-Query Parallelism

Approach #2: Intra-Query Parallelism

Inter-Query Parallelism

Improve overall performance by allowing multiple queries to execute simultaneously.

→ Most DBMSs use a simple first-come, first-served policy.

If queries are read-only, then this requires almost no explicit coordination between the queries.

→ Buffer pool can handle most of the sharing if necessary.

Lecture #16

If multiple queries are updating the database at the same time, then this is tricky to do correctly...

Intra-Query Parallelism

Improve the performance of a single query by executing its operators in parallel.

→ Think of the organization of operators in terms of a producer/consumer paradigm.

Approach #1: Intra-Operator (Horizontal)

Approach #2: Inter-Operator (Vertical)

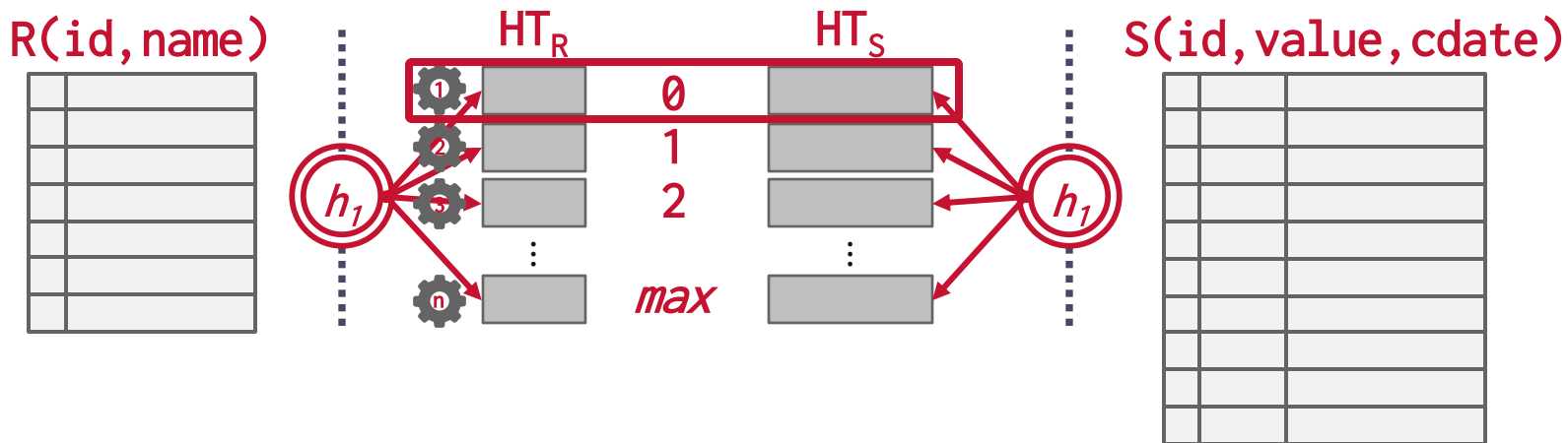
These techniques are not mutually exclusive.

There are parallel versions of every operator.


→ Can either have multiple threads access centralized data structures 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.



Intra-Query Parallelism

Approach #1: **Intra-Operator** (Horizontal)  *Most Common*

Approach #2: **Inter-Operator** (Vertical)  *Less Common*

Approach #3: **Bushy**  *Higher-end Systems*

Intra-Operator Parallelism

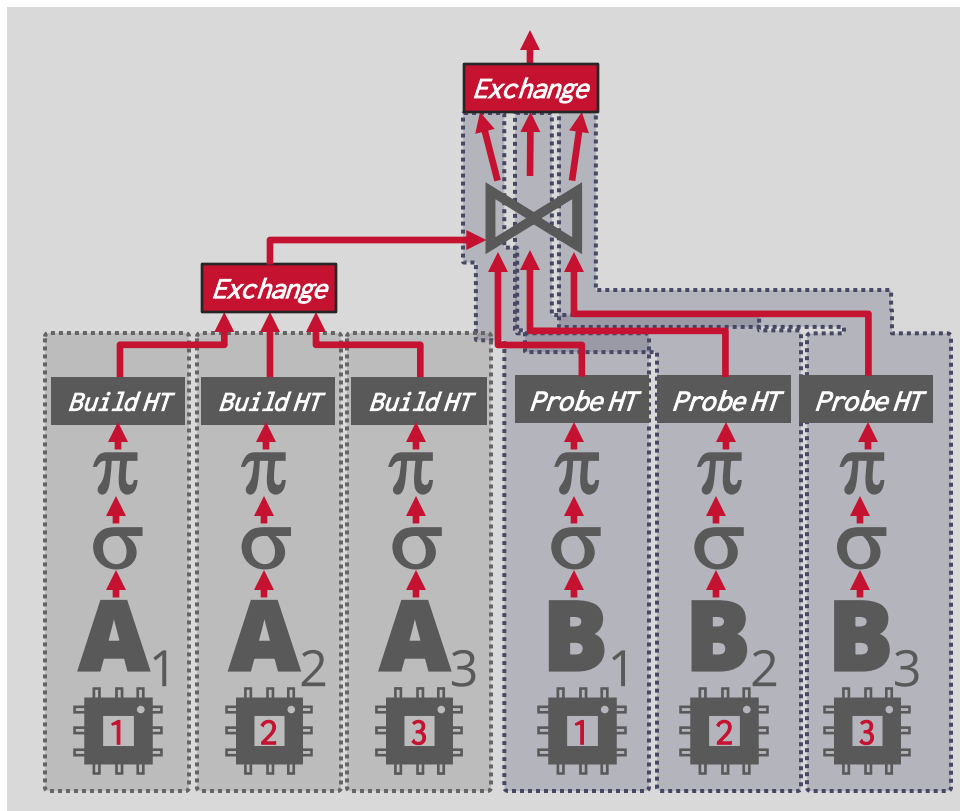
Approach #1: Intra-Operator (Horizontal)

→ Operators are decomposed into independent instances that perform the same function on different subsets of data.

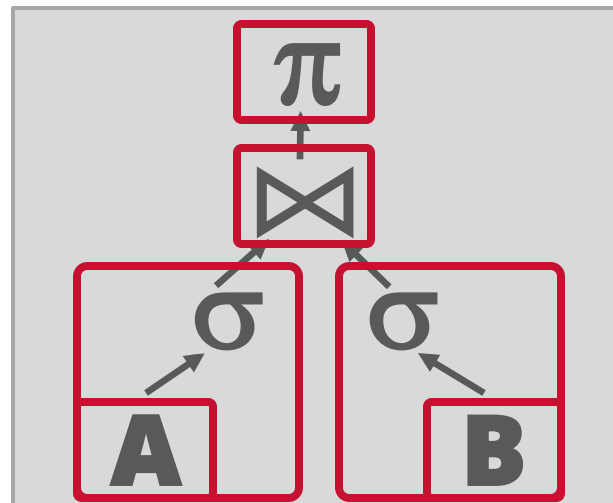
The DBMS inserts an exchange operator into the query plan to coalesce/split results from multiple children/parent operators.

→ Postgres calls this “gather”

Intra-Operator Parallelism



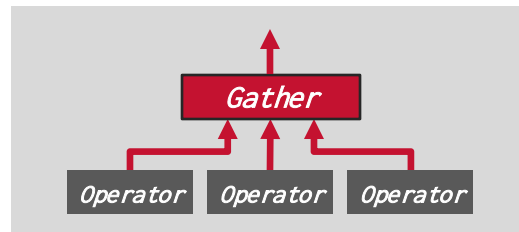
```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```



Exchange Operator

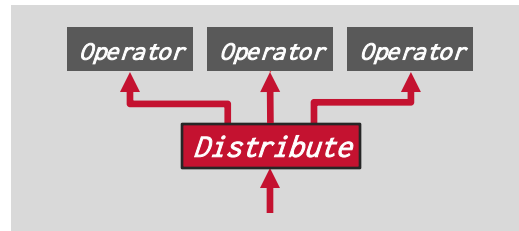
Exchange Type #1 – Gather

→ Combine the results from multiple workers into a single output stream.



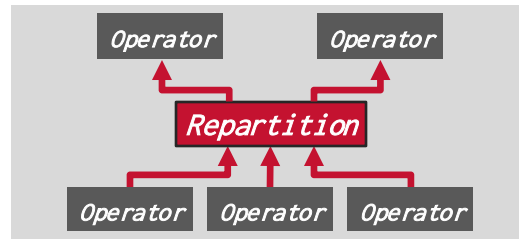
Exchange Type #2 – Distribute

→ Split a single input stream into multiple output streams.



Exchange Type #3 – Repartition

→ Shuffle multiple input streams across multiple output streams.
→ Some DBMSs always perform this step after every pipeline (e.g., Dremel/BigQuery).



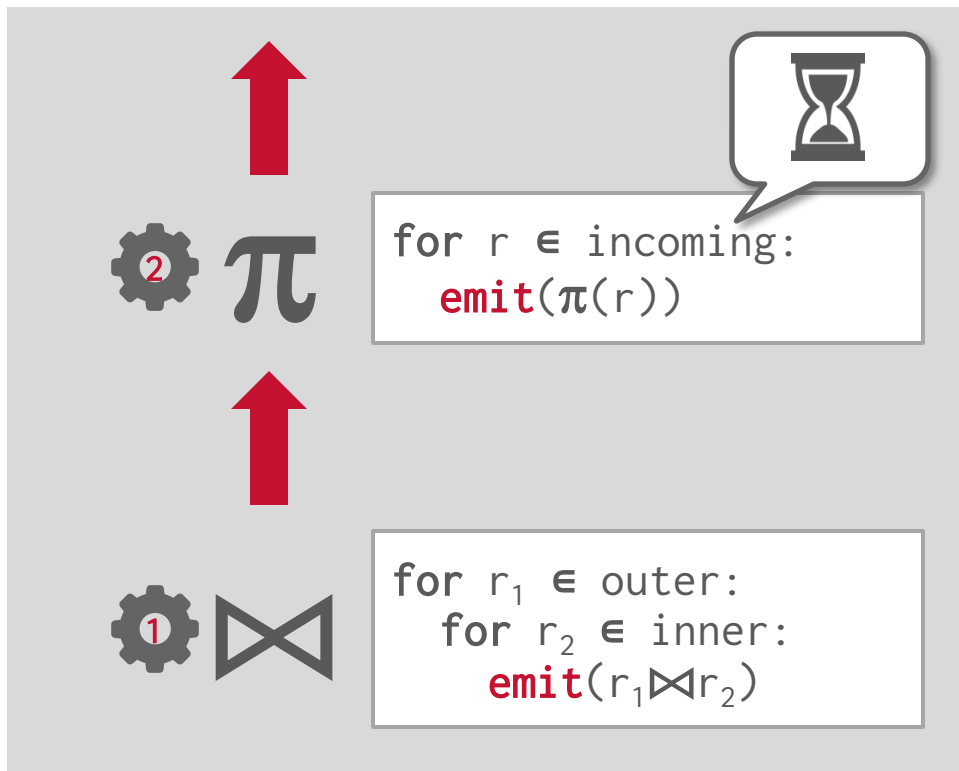
Inter-Operator Parallelism

Approach #2: Inter-Operator (Vertical)

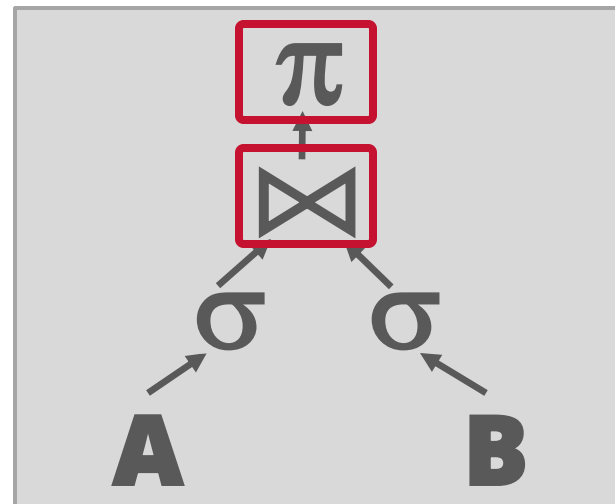
- Operations are overlapped to pipeline data from one stage to the next without materialization.
- Workers execute multiple operators from different segments of a query plan at the same time.
- Still need exchange operators to combine intermediate results from segments.

Also called pipelined parallelism.

Inter-Operator Parallelism



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

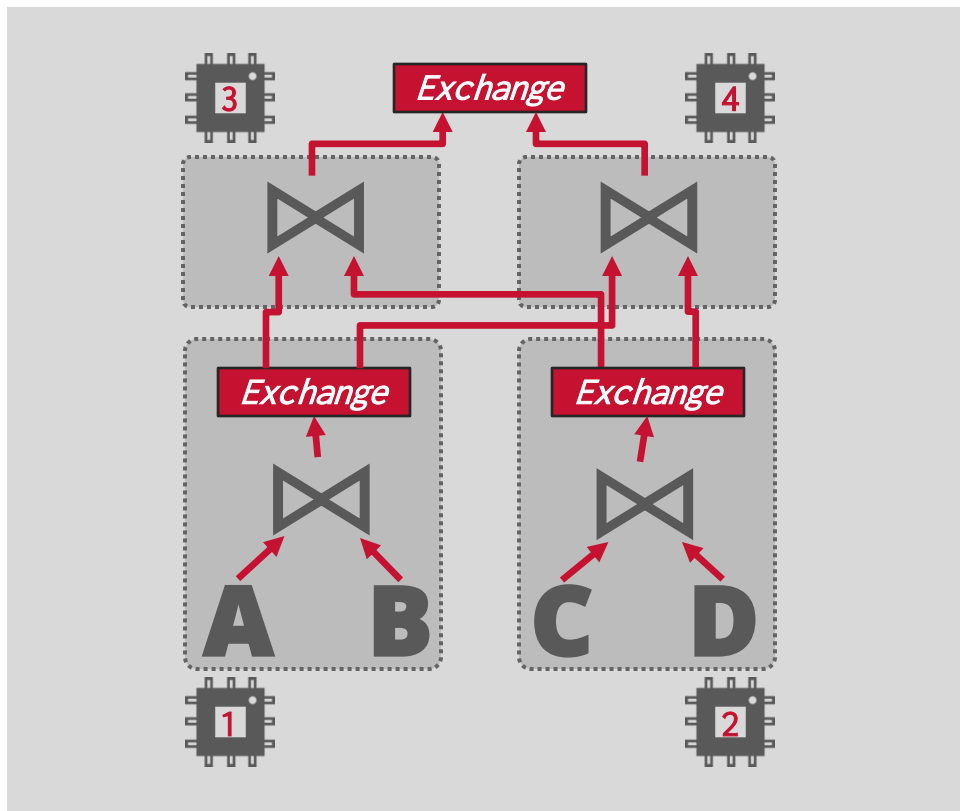


Bushy Parallelism

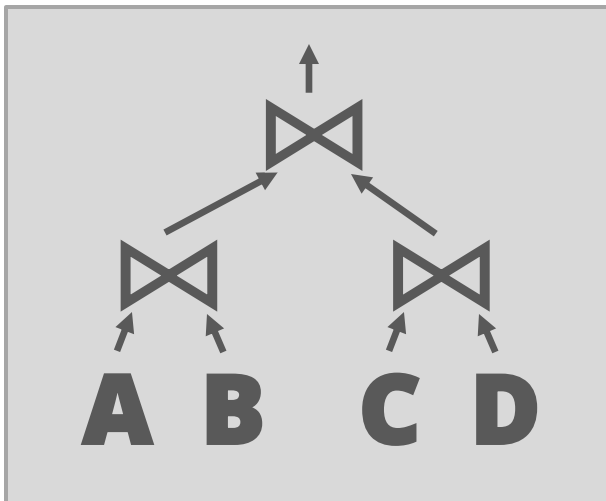
Approach #3: Bushy Parallelism

- Hybrid of intra- and 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.

Bushy Parallelism

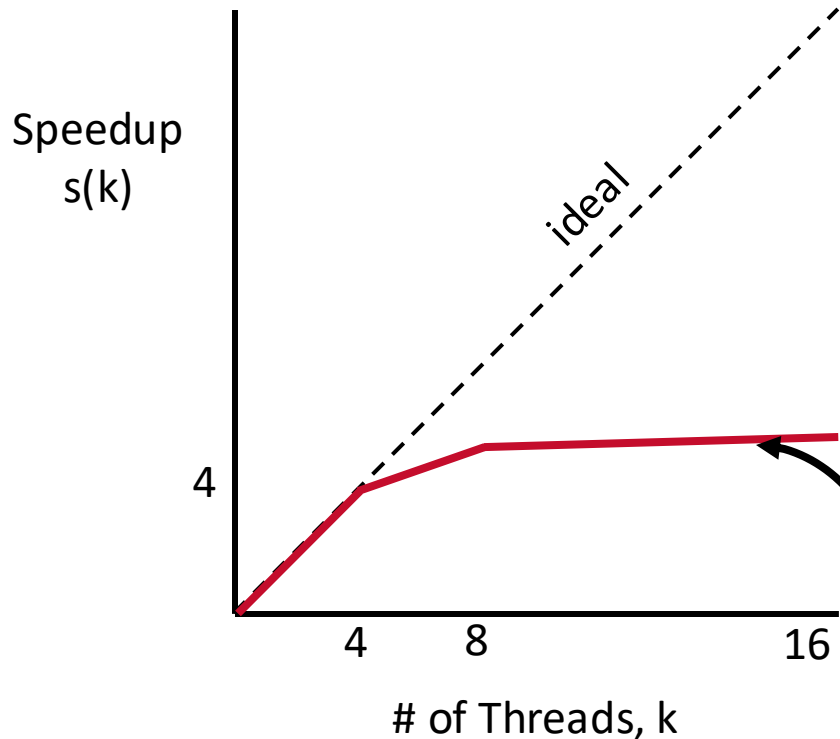


```
SELECT *
FROM A
JOIN B
JOIN C
JOIN D
```



Speedup Functions

Speedup Function, $s(k)$, says how many times faster a query run on k threads as opposed to one thread



Ideal:

$$s(k) = k$$

"run k times faster on k threads"

Actual?

what happened here?

Observation

Disk has limited bandwidth: a maximum number of GB/s that can be read/written

Fundamentally, if the query must scan 1 TB and the disk can read 300 MB/s, the best we can hope for is $1 \text{ TB} / (300 \text{ MB/s}) \approx 56 \text{ minutes}$ running time

Given additional threads, eventually **the disk becomes the bottleneck**

I/O Parallelism + RAID

Split the DBMS across multiple storage devices to improve disk bandwidth latency.

Many different options that have trade-offs:

- Multiple Disks per Database
- One Database per Disk
- One Relation per Disk
- Split Relation across Multiple Disks

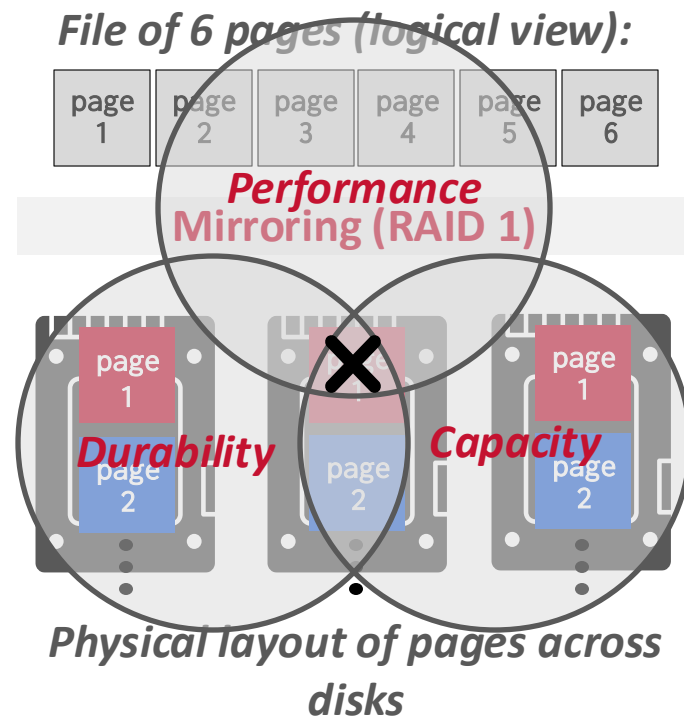
Some DBMSs support this natively. Others require admin to configure outside of DBMS.

RAID for Multi-Disk Parallelism

Store data across multiple disks to improve performance + durability.

Hardware-based: I/O controller makes multiple physical devices appear as single logical device.
→ Transparent to DBMS (e.g., RAID).

Software-based: DBMS manages erasure codes at the file/object level.
→ Faster and more flexible.



Conclusion

Parallel execution is important, which is why (almost) every major DBMS supports it.

However, it is hard to get right.

- Coordination Overhead
- Scheduling
- Concurrency Issues
- Resource Contention

Next Class

Query Optimization

- Logical vs Physical Plans
- Search Space of Plans
- Cost Estimation of Plans