Carnegie Mellon University

DATABASE SYSTEMS

Query Execution – Pt.2

LECTURE #14 » 15-445/645 FALL 2025 » PROF. ANDY PAVLO

ADMINISTRIVIA

Project #2 is due Sunday Oct 26th @ 11:59pm
→ See Recitation Video (**@158**)
→ Office Hours: Saturday Oct 25th @ 3:00-5:00pm in GHC 5207

Mid-term Exam grades posted
→ Come to Andy's OH to view your grade and solution.

Homework #4 is due Sunday Nov 2nd @ 11:59pm

Project #3 is due Sunday Nov 16th @ 11:59pm

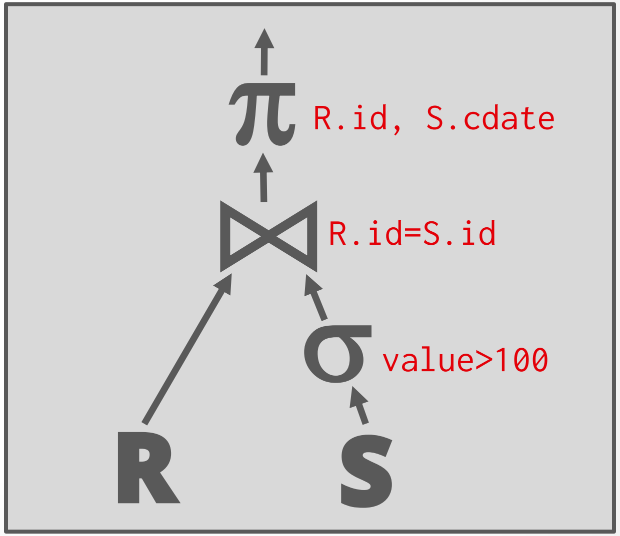
LAST CLASS

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.

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

⚡**DB Flash Talk: SpiralDB / Vortex**

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

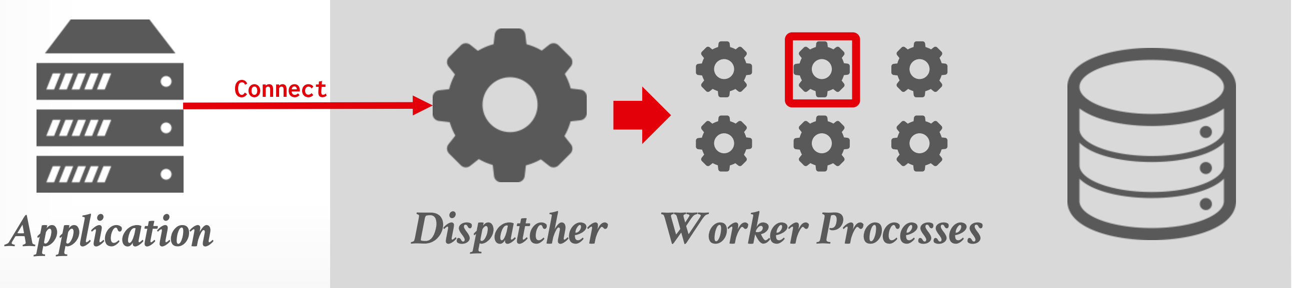
Each worker is a separate OS process.

→ Relies on the OS dispatcher.

→ Use shared-memory for global data structures.

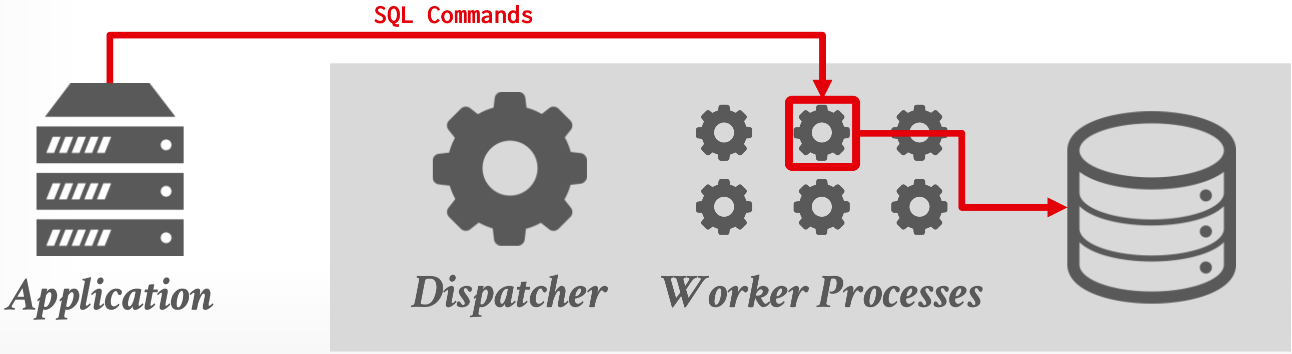
→ A process crash does not take down the entire system.

→ **Examples**: IBM DB2, Postgres, Oracle



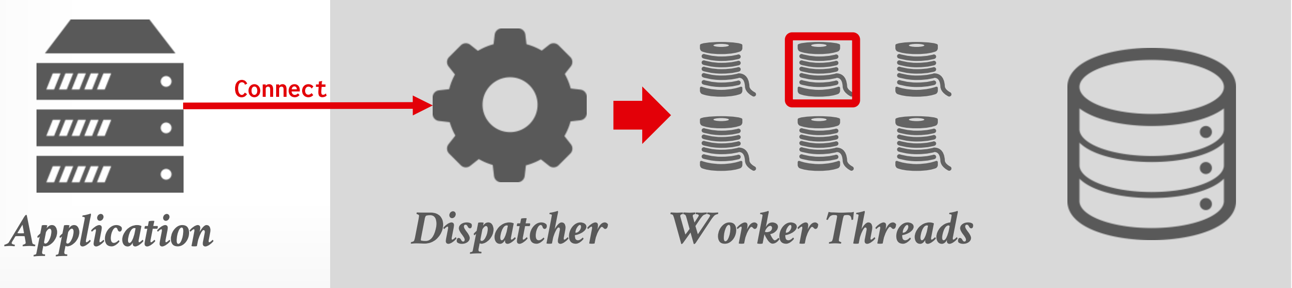
PROCESS PER WORKER

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



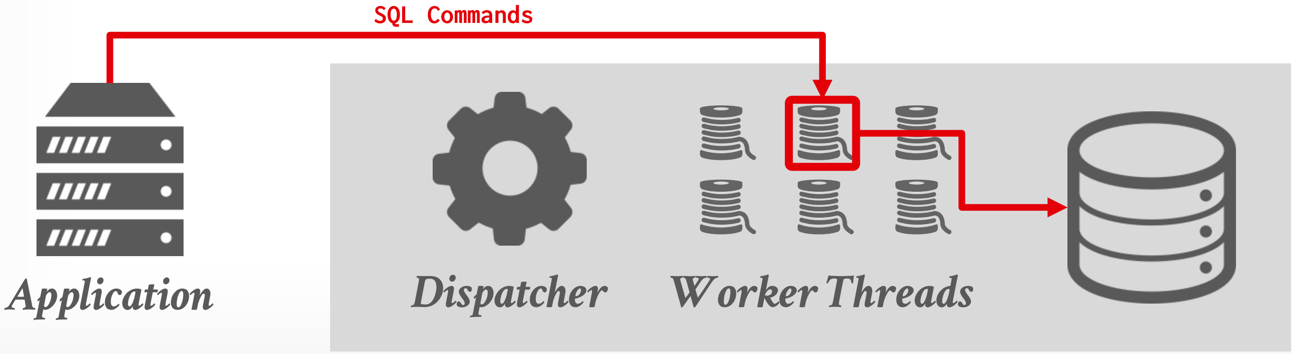
THREAD PER WORKER

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)



THREAD PER WORKER

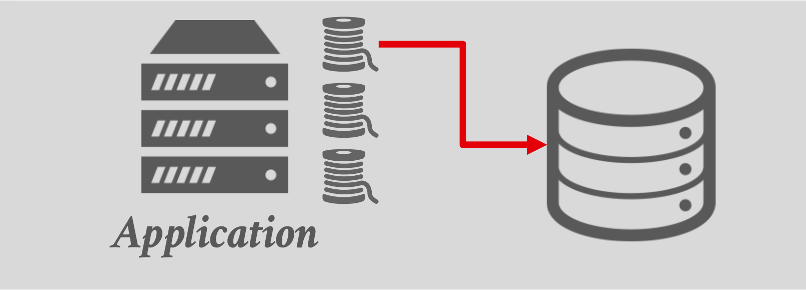
Single process with multiple worker threads.
→ DBMS (mostly) manages its own scheduling.
→ May or may not use a dispatcher thread.
→ Thread crash (may) kill the entire system.
→ **Examples**: MSSQL, MySQL, DB2, Oracle (2014)



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.

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

Every DBMS from the last 25 years uses 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 #17

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)

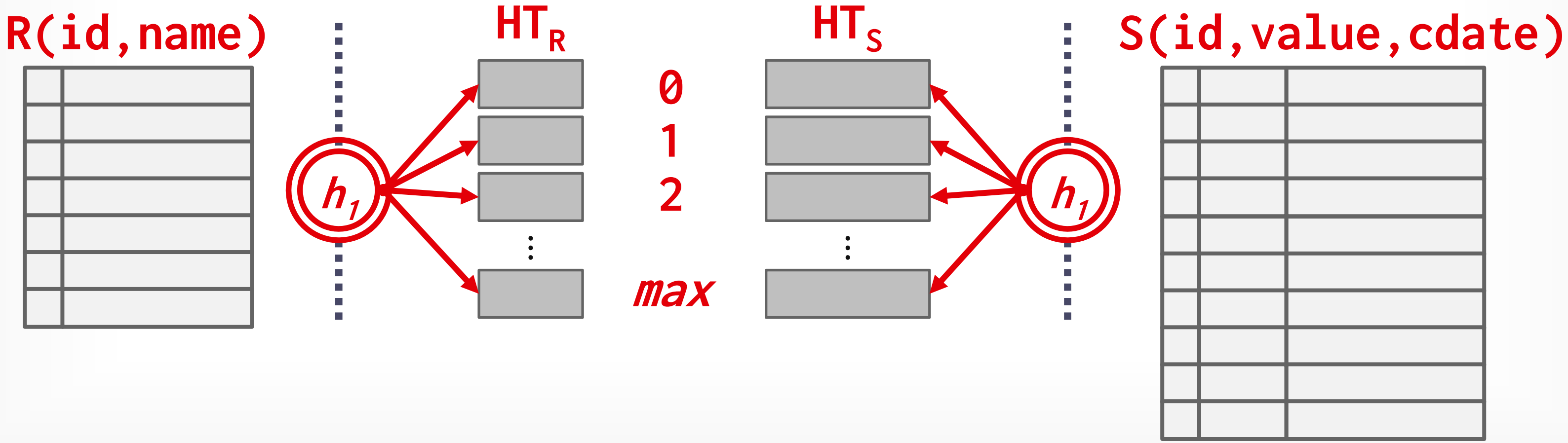
These techniques are not mutually exclusive.

There are parallel versions of every operator.

→ Can either have multiple threads access centralized data structures or use partitioning to divide work up.

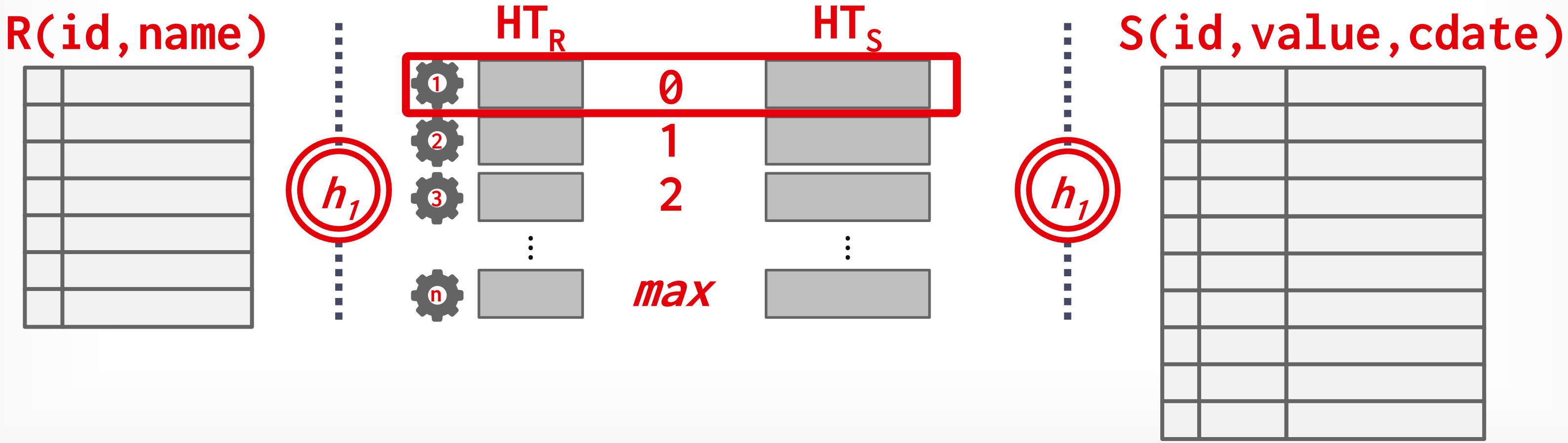
PARALLEL GRACE HASH JOIN

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



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)

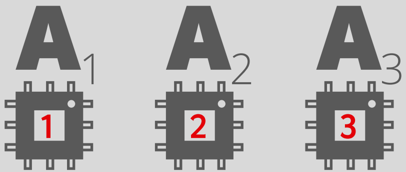
→ Operators are decomposed into independent instances that perform the same function on different subsets of data.

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

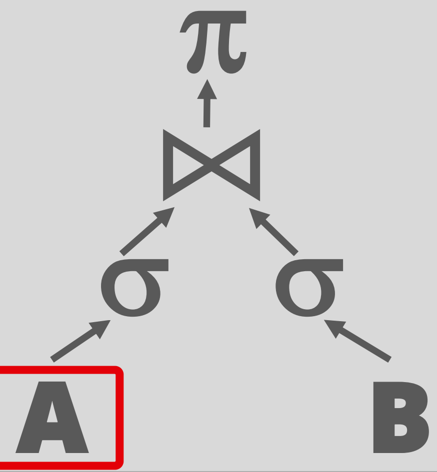
→ PostgreSQL calls these Gather operators.

→ Can combine intermediate results in arbitrary order or merge them according to a sorting key.

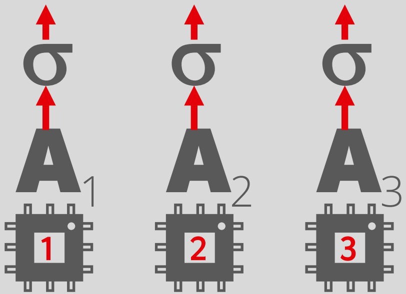
INTRA-OPERATOR PARALLELISM



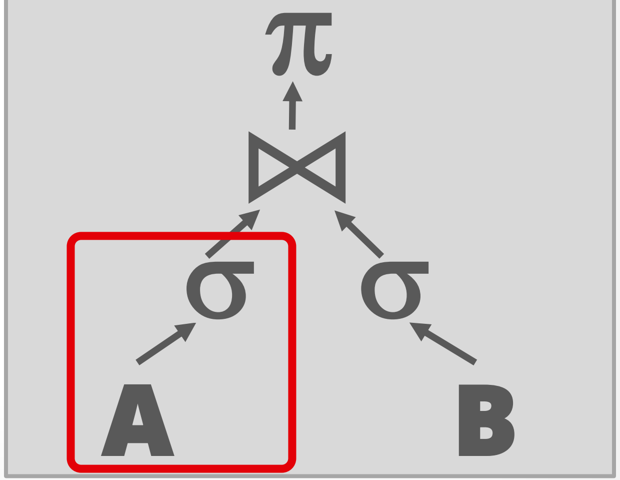

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



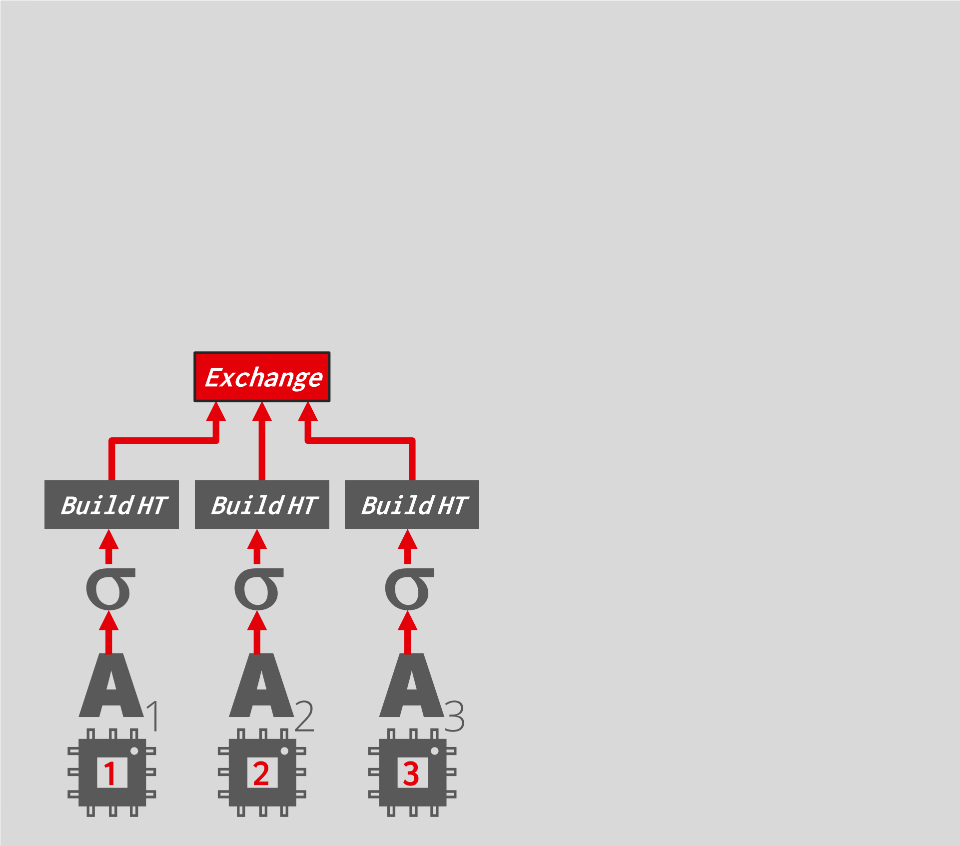
INTRA-OPERATOR PARALLELISM



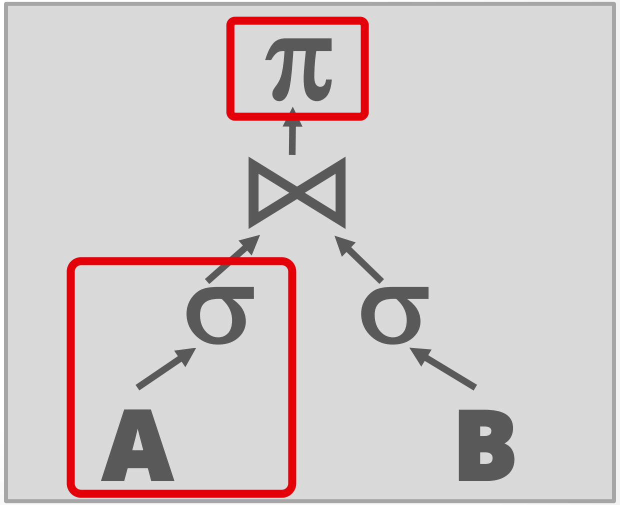

```sql
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



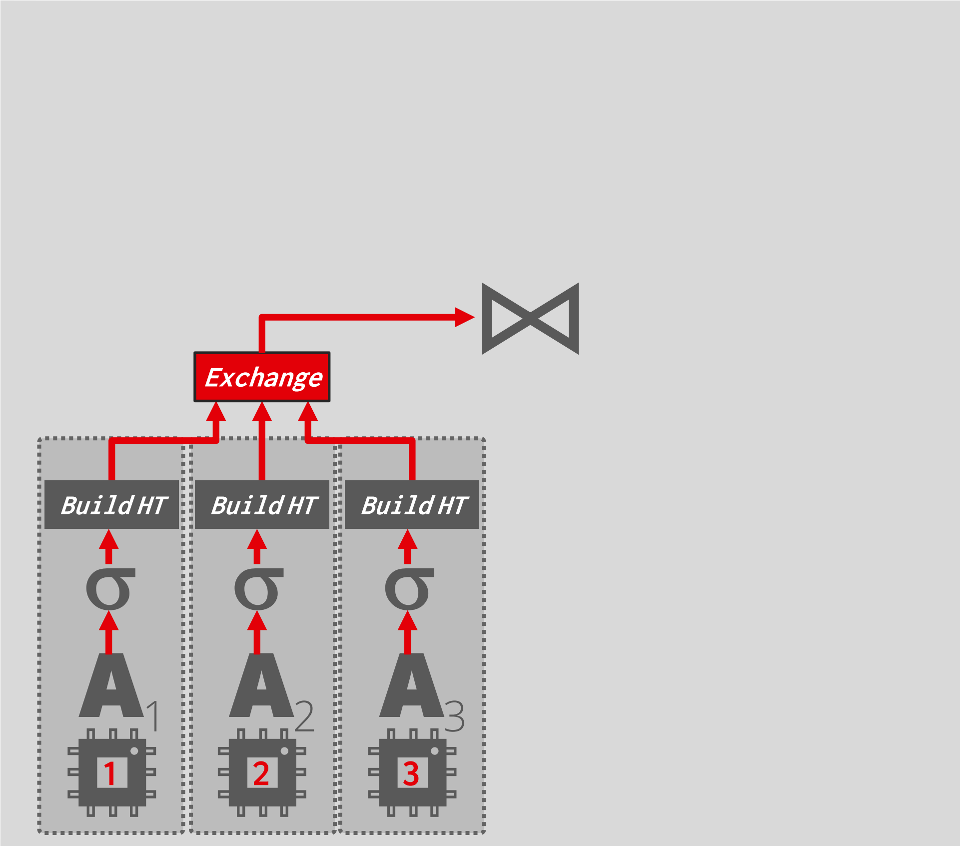
INTRA-OPERATOR PARALLELISM



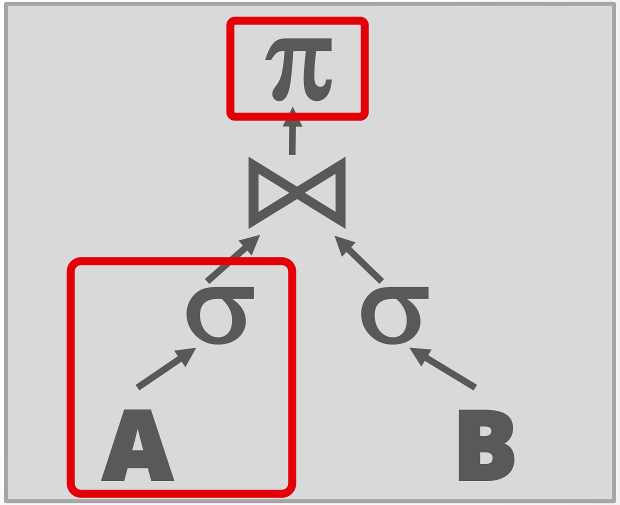

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



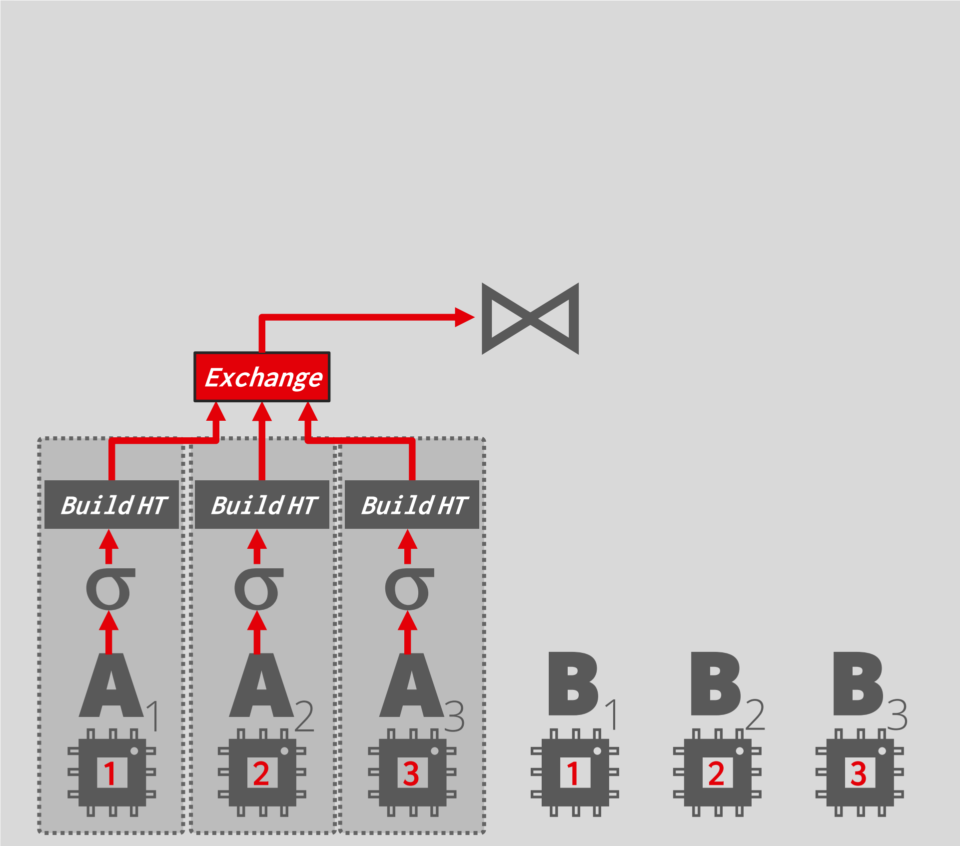
INTRA-OPERATOR PARALLELISM



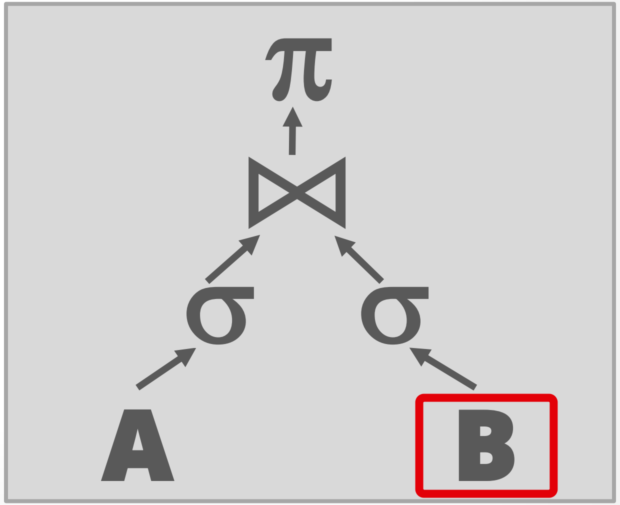

```sql
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



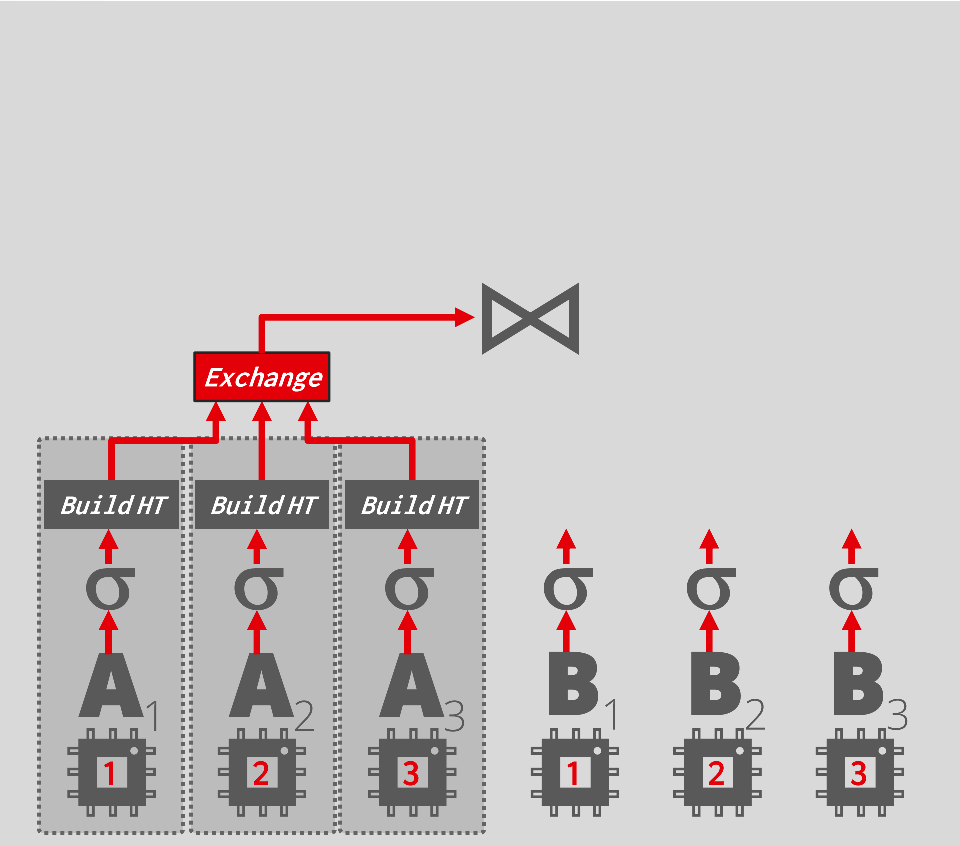
INTRA-OPERATOR PARALLELISM



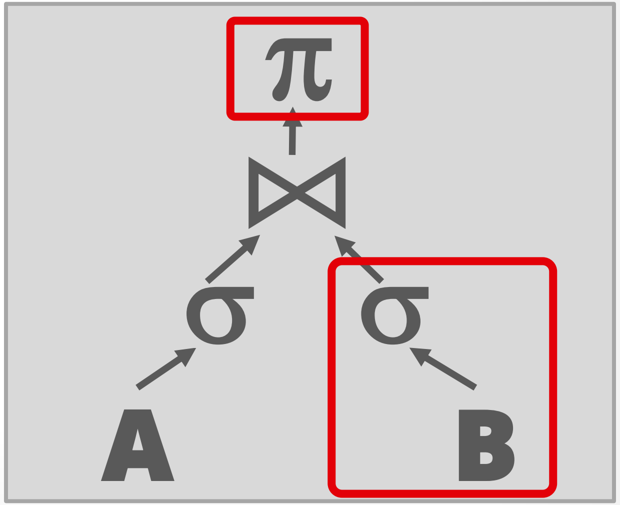

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



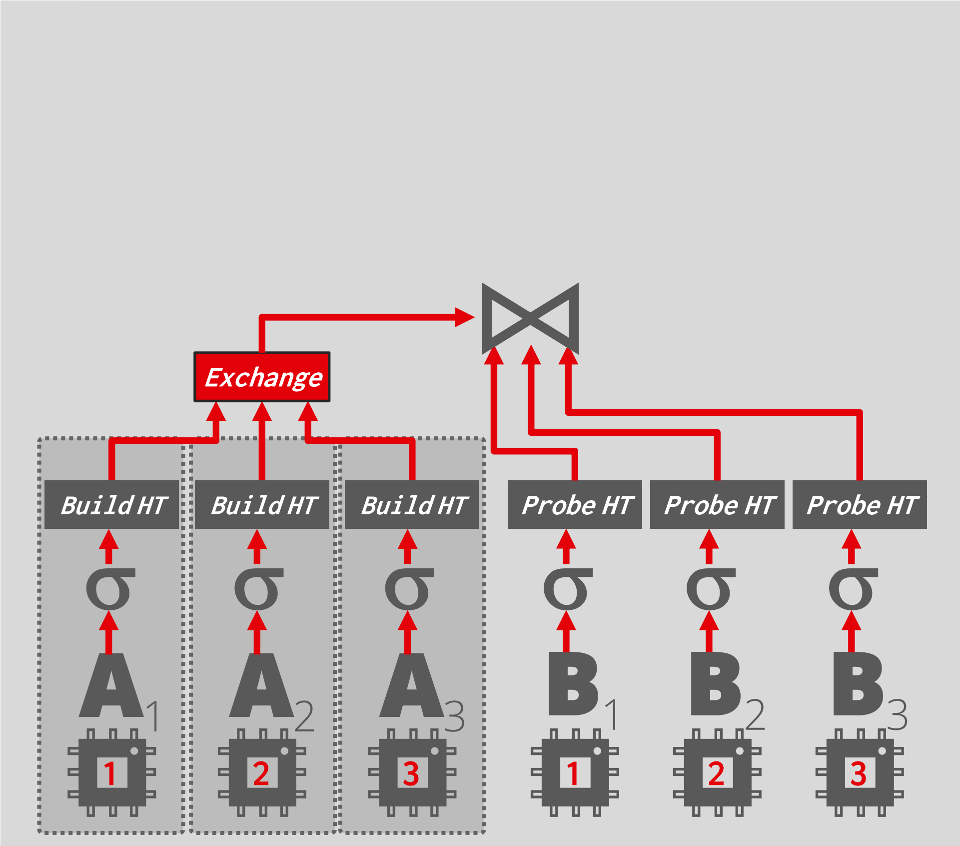
INTRA-OPERATOR PARALLELISM



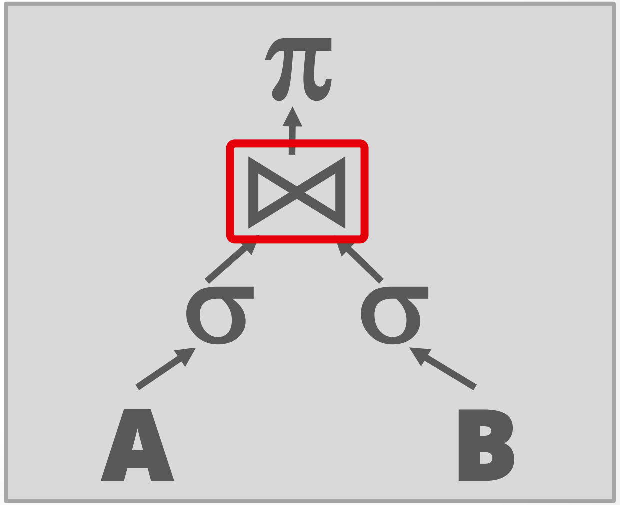

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



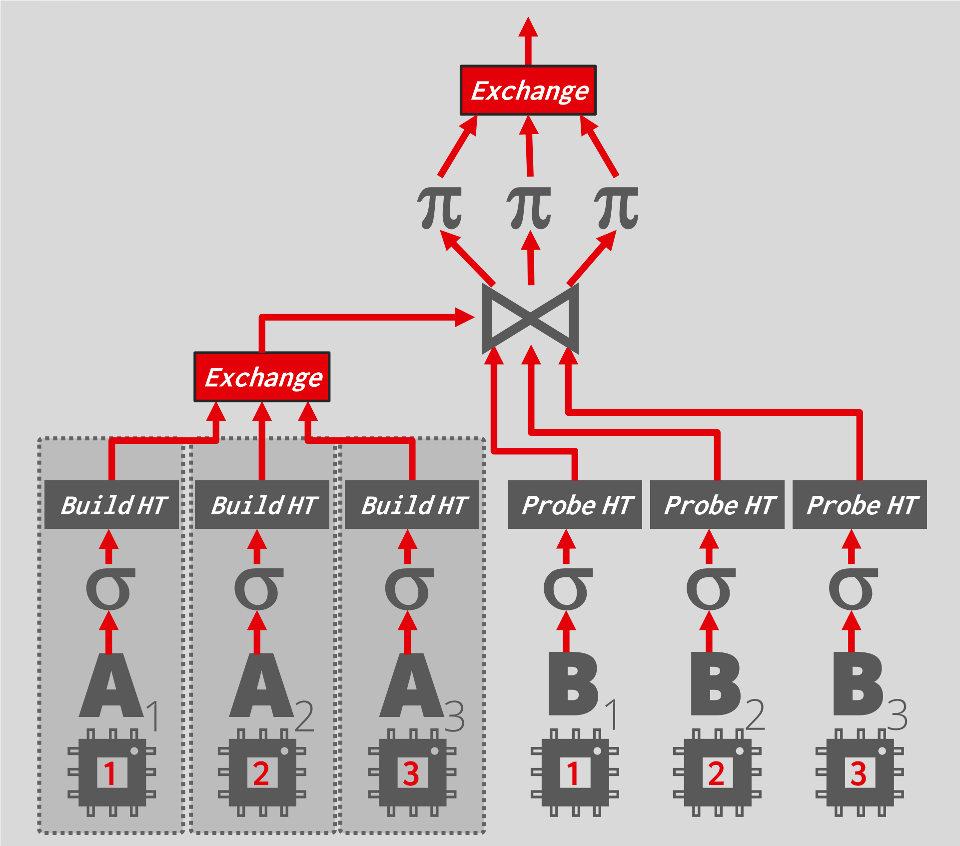
INTRA-OPERATOR PARALLELISM



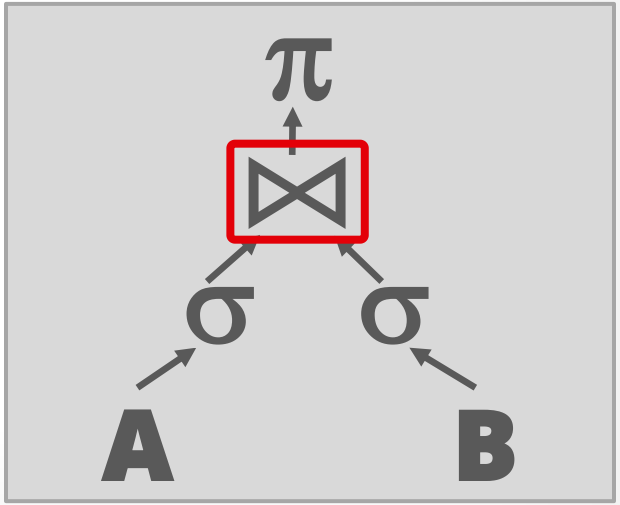

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



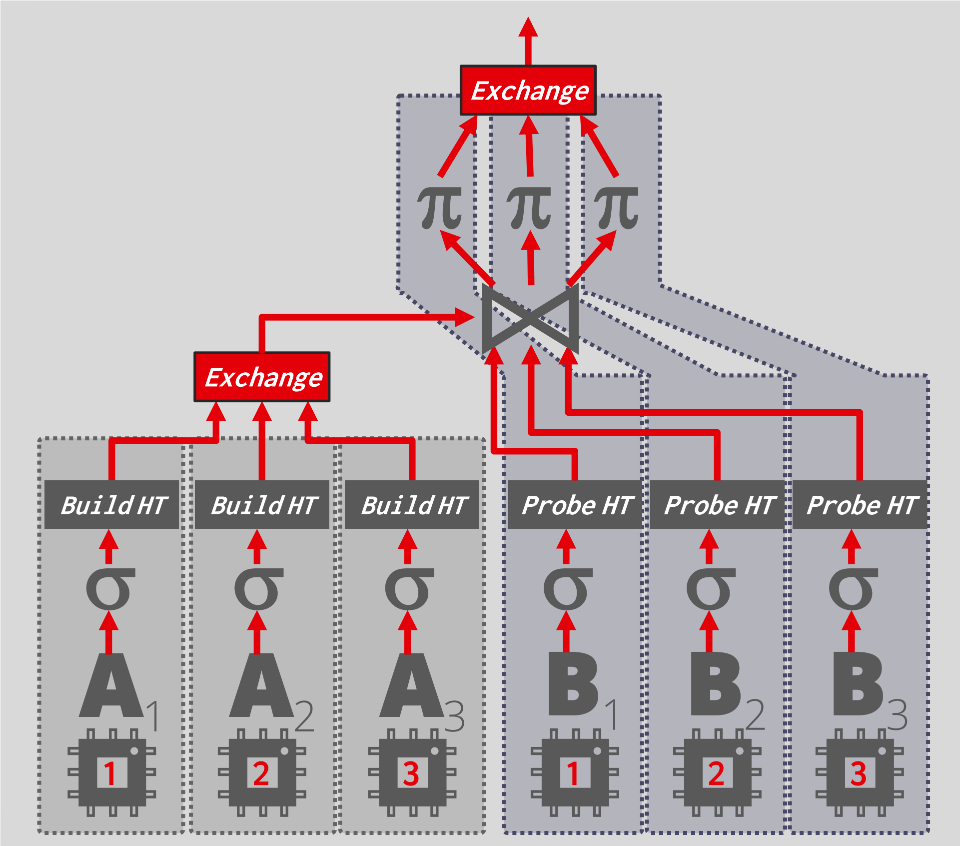
INTRA-OPERATOR PARALLELISM



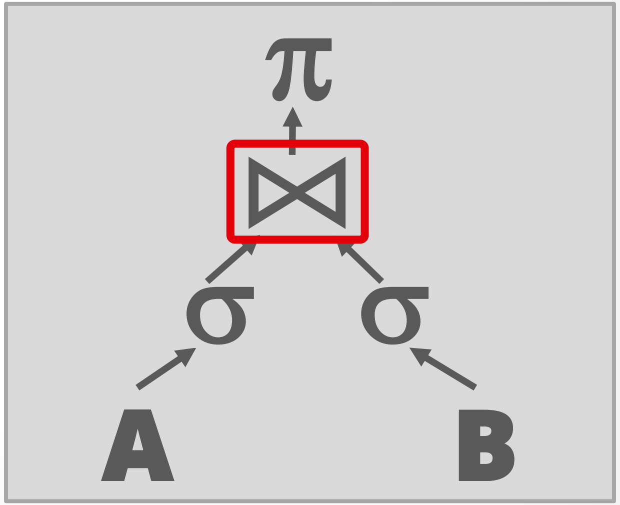

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



INTRA-OPERATOR PARALLELISM



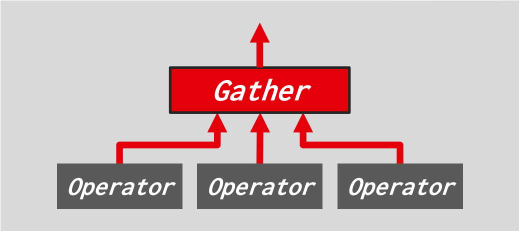

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



EXCHANGE OPERATOR

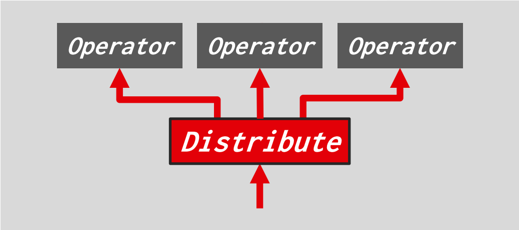
Exchange Type #1 – Gather

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



Exchange Type #2 – Distribute

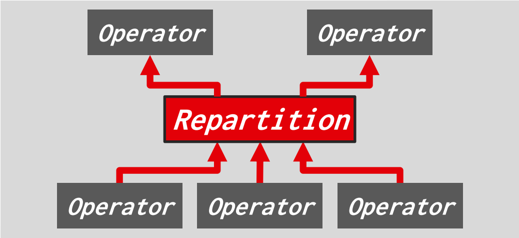
→ Split a single input stream into multiple output streams.



Exchange Type #3 – Repartition

→ Shuffle multiple input streams across multiple output streams.

→ Some DBMSs always perform this step after every pipeline (e.g., Google BigQuery).



Source: Craig Freedman

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**.

PULSAR

kafka

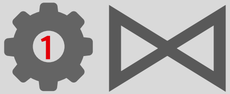
Flink

APACHE STORM™

HERON

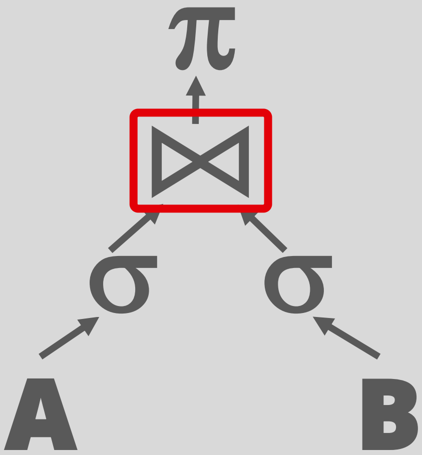
APACHE Spark™

INTER-OPERATOR PARALLELISM




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

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



INTER-OPERATOR PARALLELISM

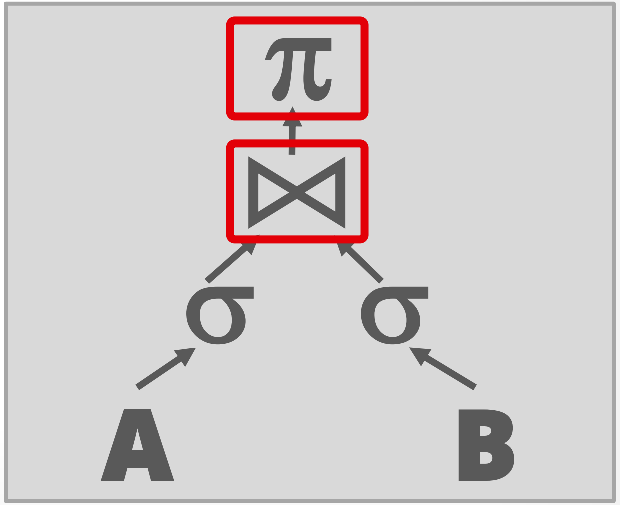
2 π

for $r \in$ incoming:
  emit($\pi(r)$)

1 $\bowtie$

for $r_1 \in$ outer:
  for $r_2 \in$ inner:
    emit($r_1 \bowtie r_2$)

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



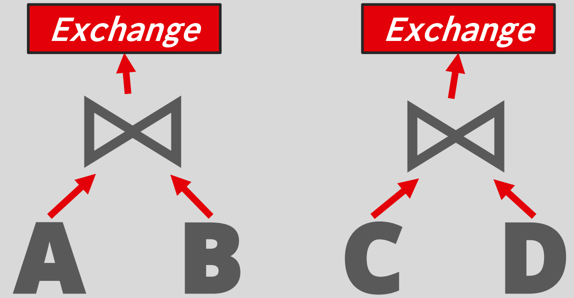
BUSHY PARALLELISM

Approach #3: Bushy Parallelism

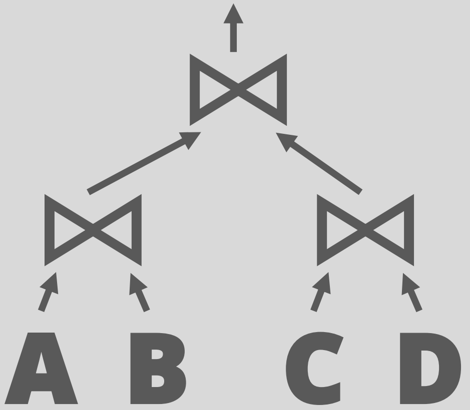
→ Hybrid of intra- and inter-operator parallelism where workers execute multiple operators from different segments of a query plan at the same time.

→ Still need exchange operators to combine intermediate results from segments.

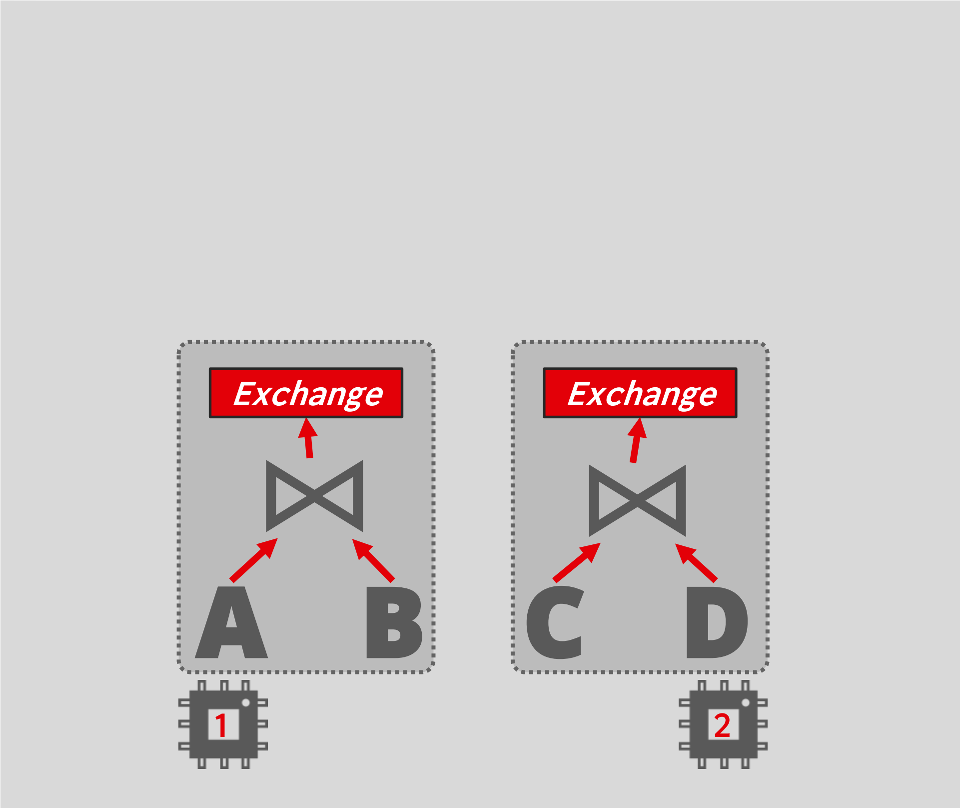
BUSHY PARALLELISM



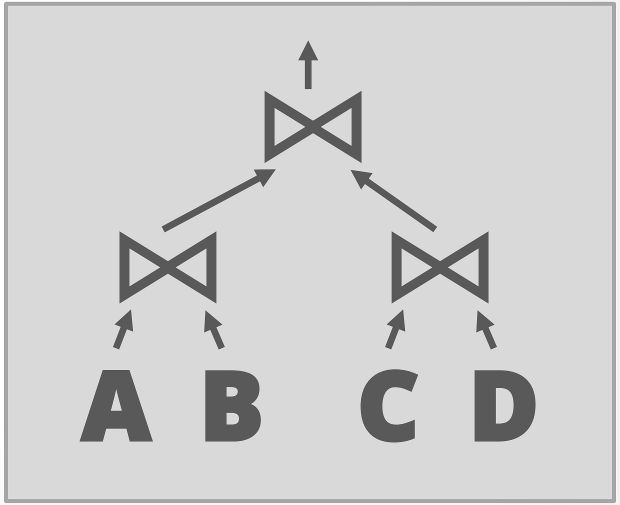

```
SELECT *
  FROM A
  JOIN B
  JOIN C
  JOIN D
```



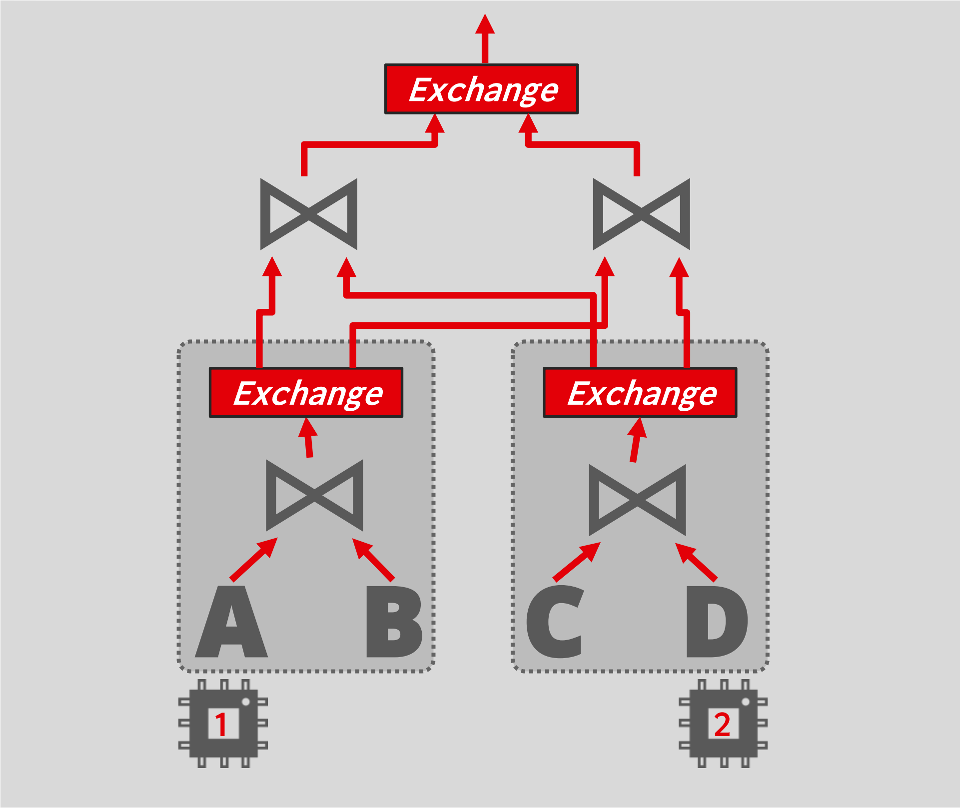
BUSHY PARALLELISM



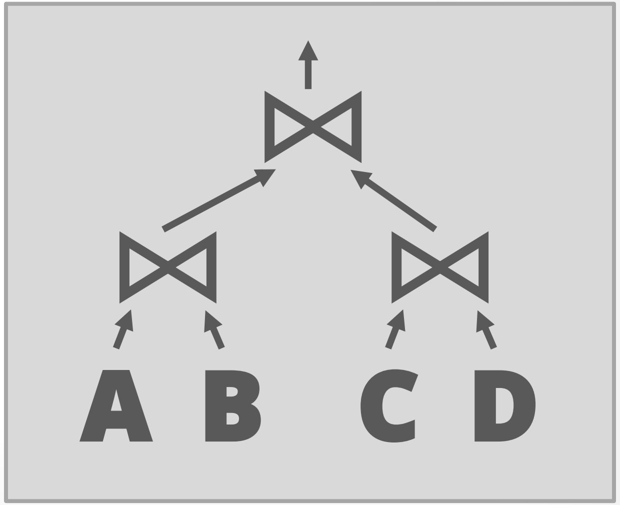

```
SELECT *
  FROM A
  JOIN B
  JOIN C
  JOIN D
```



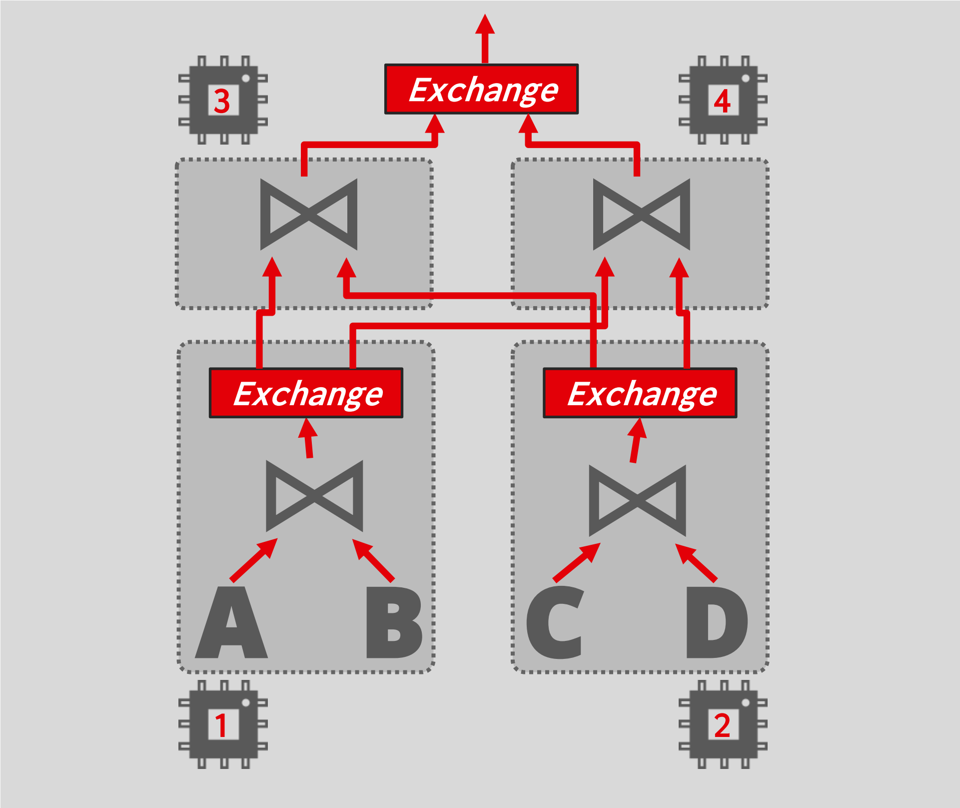
BUSHY PARALLELISM



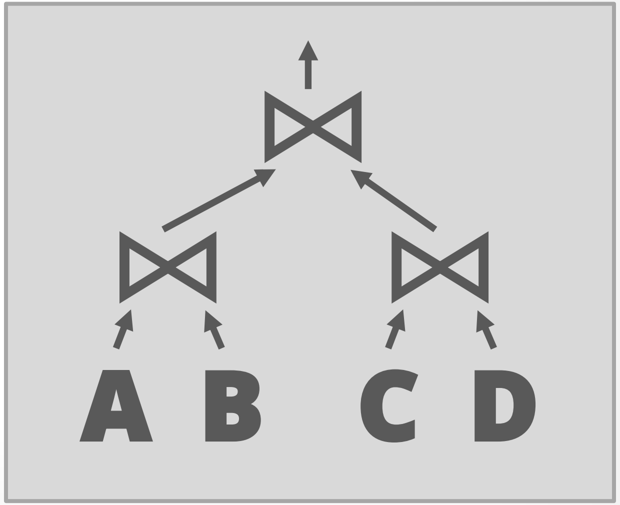

```
SELECT *
  FROM A
  JOIN B
  JOIN C
  JOIN D
```



BUSHY PARALLELISM




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



OBSERVATION

Using multiple workers for parallel query execution will not improve the DBMS's performance if the disk is always the bottleneck.

It can sometimes make the DBMS's performance worse if a worker is accessing different segments of the disk at the same time.

I/O PARALLELISM

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

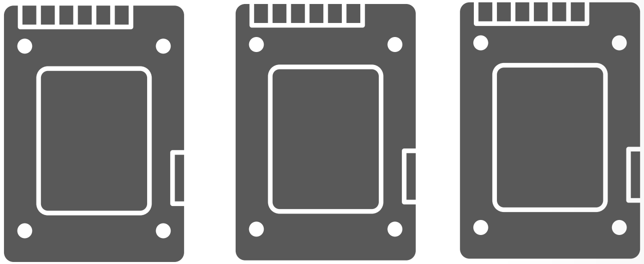
Some DBMSs support this natively (e.g., PostgreSQL Tablespace).
Others require admin to configure outside of DBMS.

MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

File of 6 pages (logical view):

page 1	page 2	page 3	page 4	page 5	page 6

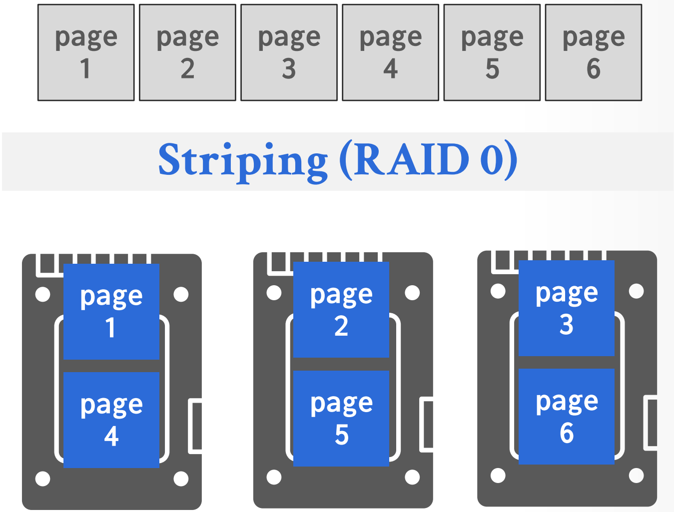


Physical layout of pages across disks

MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

File of 6 pages (logical view):



Physical layout of pages across disks

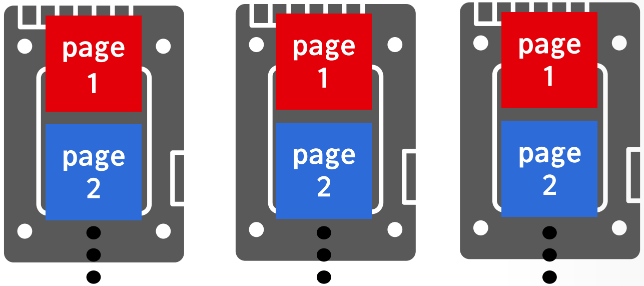
MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

File of 6 pages (logical view):

page 1	page 2	page 3	page 4	page 5	page 6

Mirroring (RAID 1)



Physical layout of pages across disks

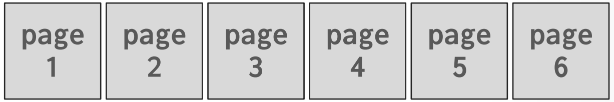
MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

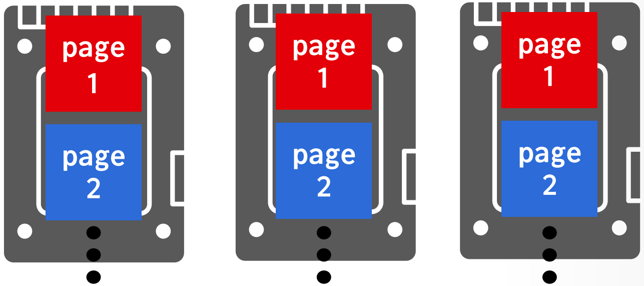
Hardware-based: I/O controller makes multiple physical devices appear as single logical device.
→ Transparent to DBMS (e.g., RAID).

s
→ Faster and more flexible.
→ s erasure codes at the file/object level.

File of 6 pages (logical view):



Mirroring (RAID 1)



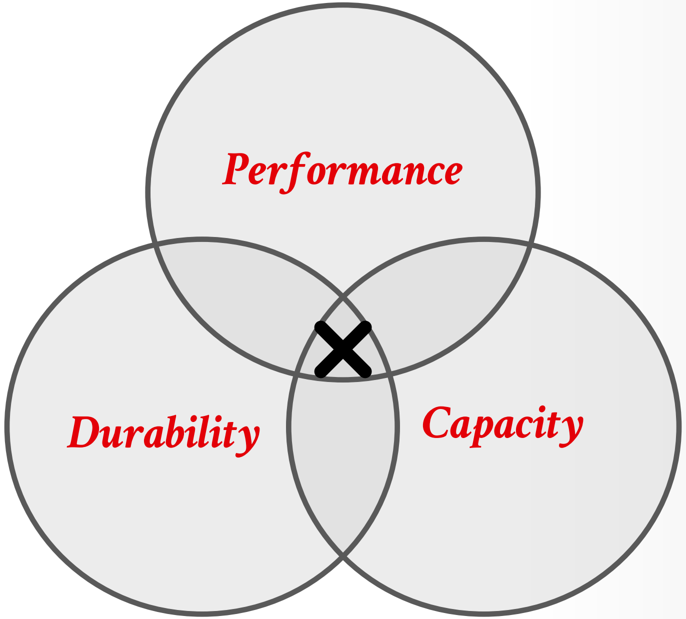
Physical layout of pages across disks

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).

s
→ Faster and more flexible.
→ s erasure codes at the file/object level.



DATABASE PARTITIONING

Some DBMSs allow you to 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 DBMS recovery log file might still be shared if transactions can update multiple databases.

PARTITIONING

Split a single logical table into disjoint physical segments that are stored/managed separately.

Partitioning should (ideally) be transparent to the application.

→ The application should only access logical tables and not have to worry about how things are physically stored.

We will cover this further when we talk about distributed databases.

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