Carnegie Mellon University

Database Systems

15-445/645 FALL 2026

JIGNESH PATEL



Lecture #14

Query Execution: Part 2

ADMINISTRIVIA

Project #3 is due Sunday Nov 15th, 2026 @ 11:59pm
→ You may (not required) to use coding agents
→ You still need to establish "code ownership"

HW #3 is due Sunday Nov 1st, 2026 @ 11:59pm

As noted in the last class, I won't hold OH today

About half the class has been requested to meet with our TAs for a CRI session

UPCOMING DATABASE TALKS

databricks (DB Seminar)
→ Monday Oct 26th at 4:30pm ET, on Zoom



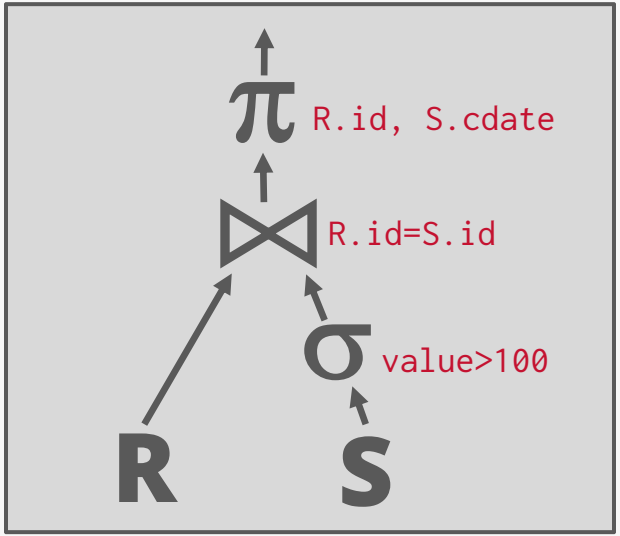
LAST CLASS

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

Query Parallelism

Data 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

Approach #3: **Embedded** DBMS

PROCESS MODEL

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

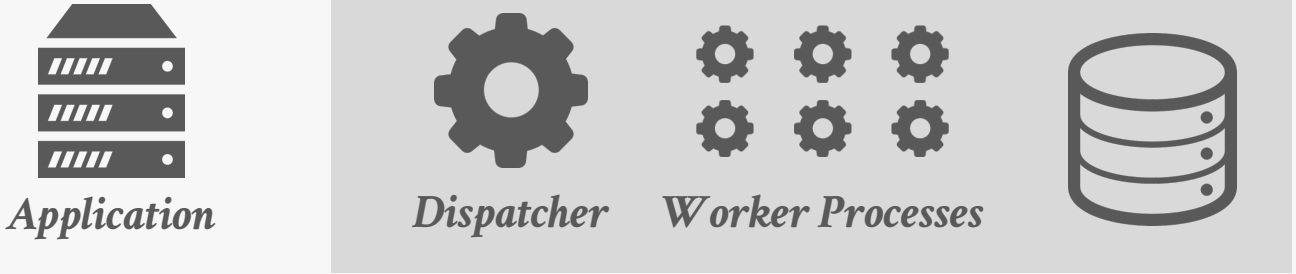
Approach #2: **Thread** per DBMS Worker ← *Most Common*

Approach #3: **Embedded** DBMS

PROCESS PER WORKER

Each worker is a separate OS process.

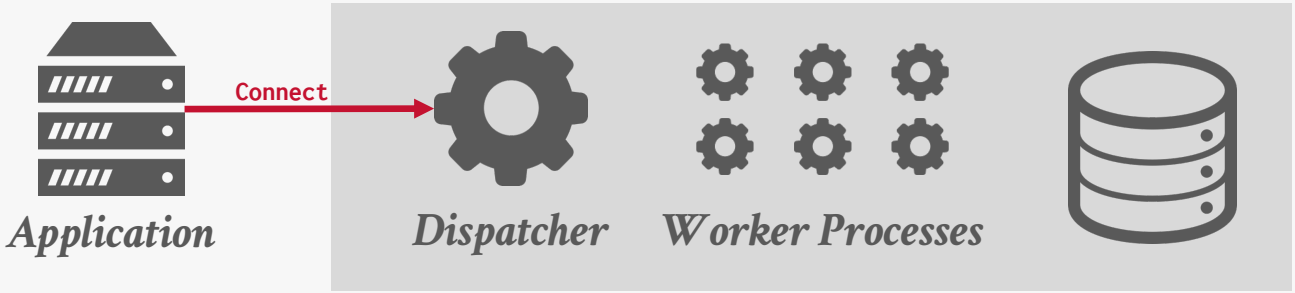
→ Relies on the OS dispatcher.
→ Use shared-memory for global data structures.
→ A process crash does not take down the entire system.
→ **Examples**: IBM DB2, Postgres, Oracle



PROCESS PER WORKER

Each worker is a separate OS process.

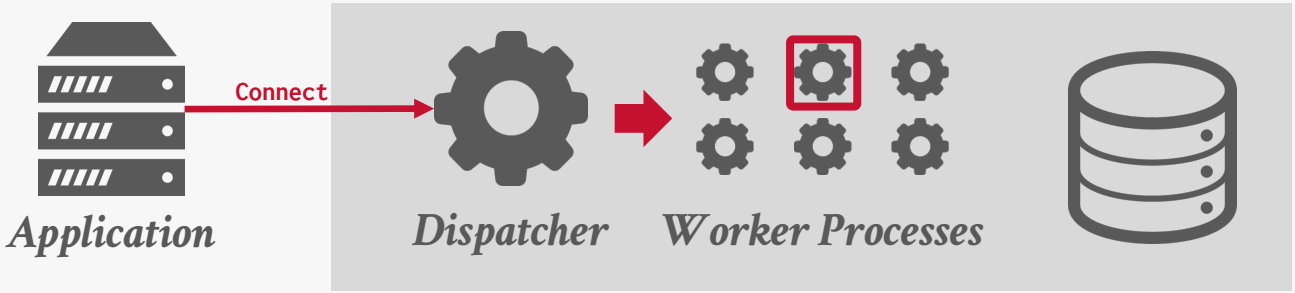
→ Relies on the OS dispatcher.
→ Use shared-memory for global data structures.
→ A process crash does not take down the entire system.
→ **Examples**: IBM DB2, Postgres, Oracle



PROCESS PER WORKER

Each worker is a separate OS process.

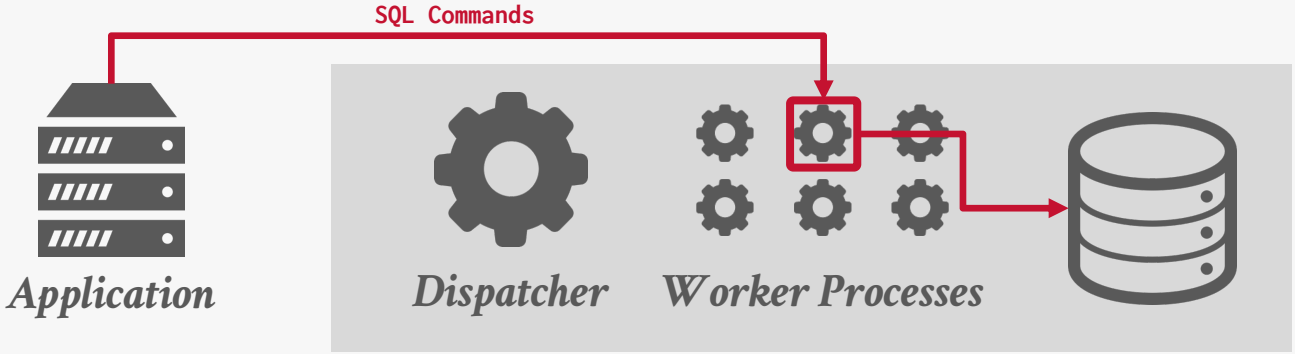
→ Relies on the OS dispatcher.
→ Use shared-memory for global data structures.
→ A process crash does not take down the entire system.
→ **Examples**: IBM DB2, Postgres, Oracle



PROCESS PER WORKER

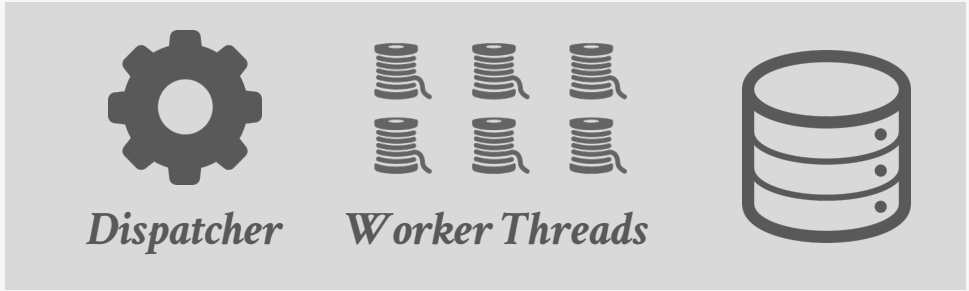
Each worker is a separate OS process.

→ 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



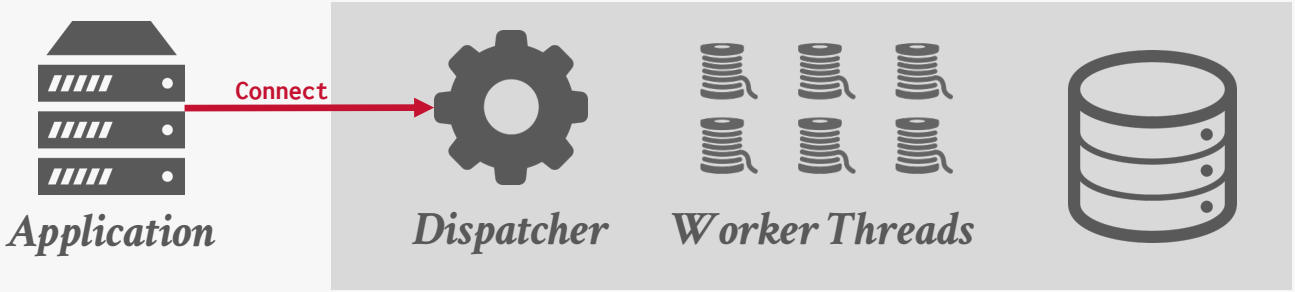
THREAD PER WORKER

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)



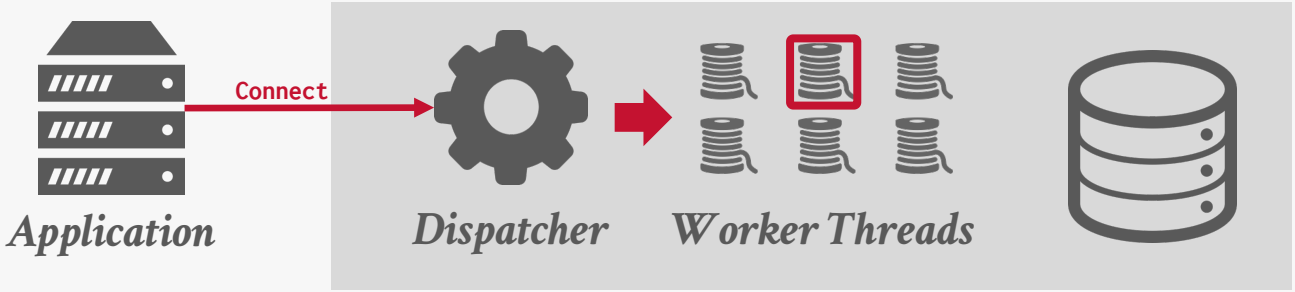
THREAD PER WORKER

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)



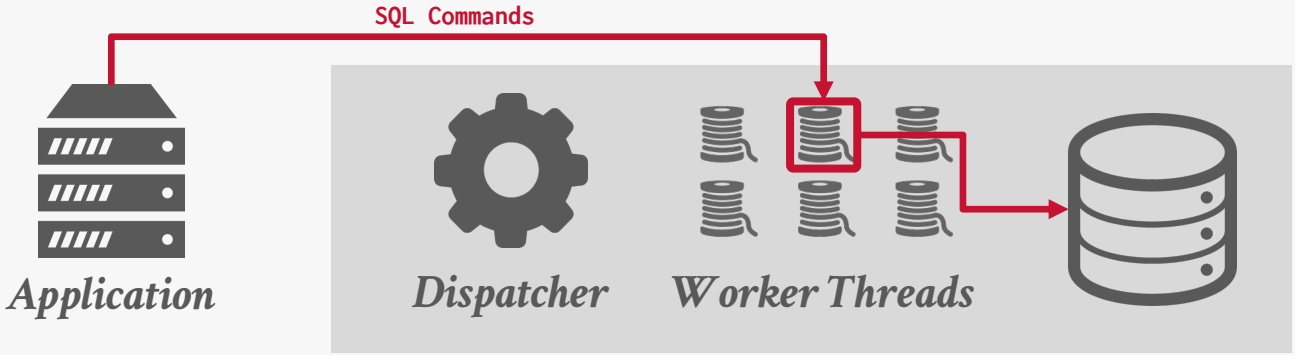
THREAD PER WORKER

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



SQLite
ORACLE BERKELEY DB
RocksDB
DuckDB
levelDB
YottaDB
WIREDTIGER
SPLINTERDB
bitcask

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

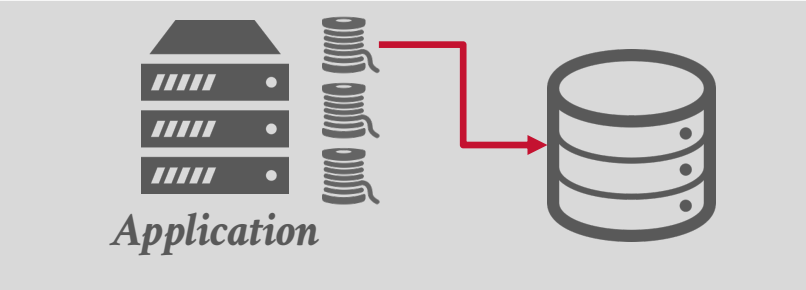


SQLite
ORACLE BERKELEY DB
RocksDB
DuckDB
levelDB
YottaDB
WIREDTIGER
SPLINTERDB
bitcask

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



SQLite
ORACLE BERKELEY DB
RocksDB
DuckDB
levelDB
YottaDB
WIREDTIGER
SPLINTERDB
bitcask

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.

QUERY PARALLELISM

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.

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

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.

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

Lecture #17

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)

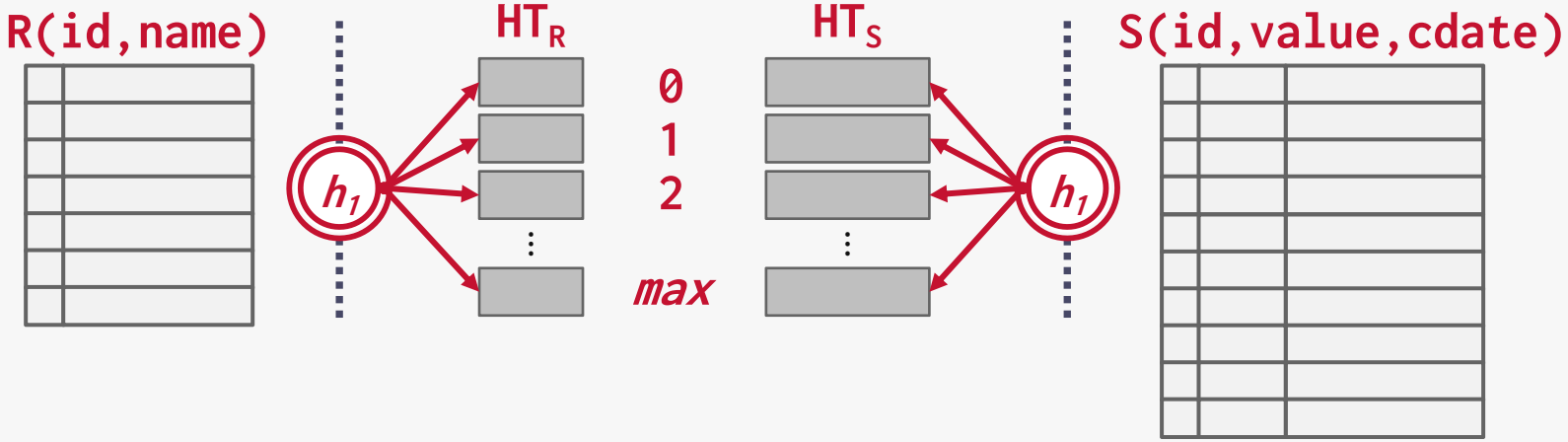
These techniques are not mutually exclusive.

There are parallel versions of every operator.

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

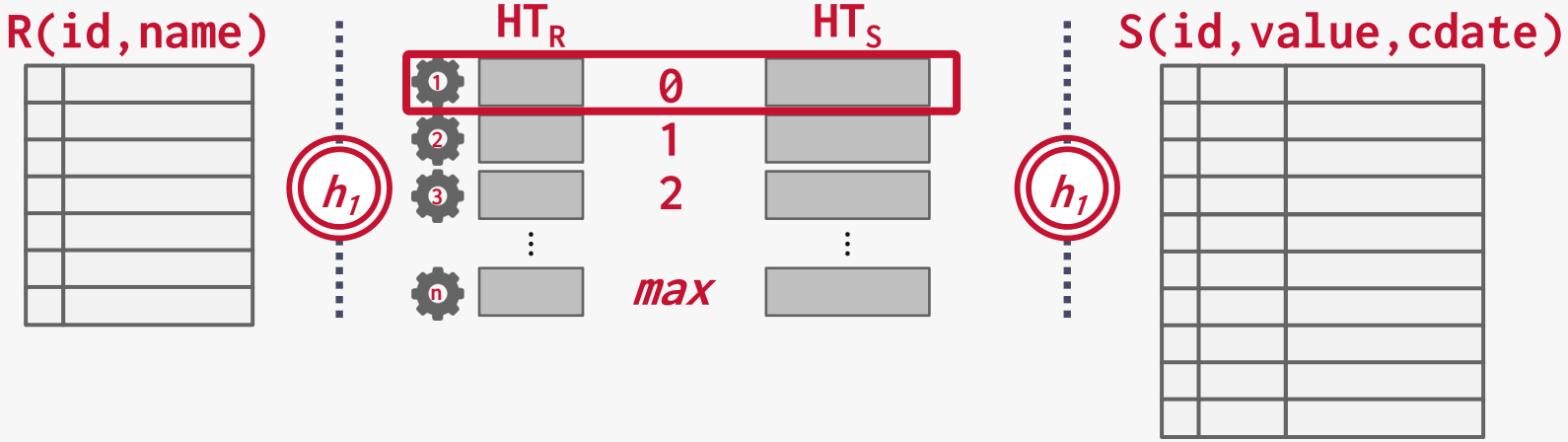
PARALLEL GRACE HASH JOIN

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)

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

Approach #3: **Bushy**

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.

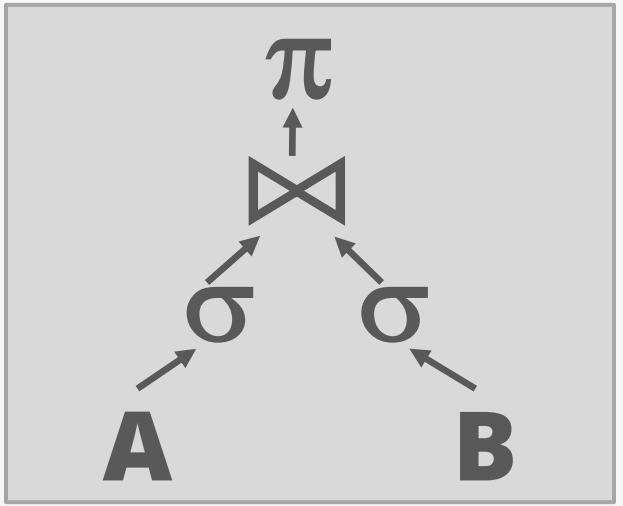
The DBMS inserts an **exchange** operator into the query plan to coalesce/split results from multiple children/parent operators.

→ PostgreSQL calls these Gather operators.

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

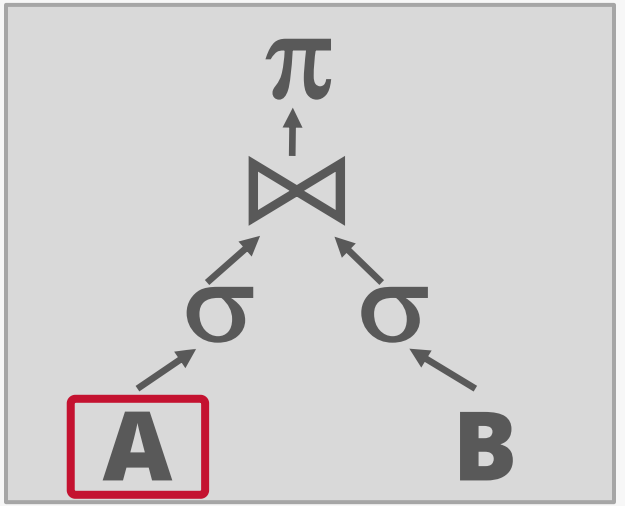
INTRA-OPERATOR PARALLELISM

```
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

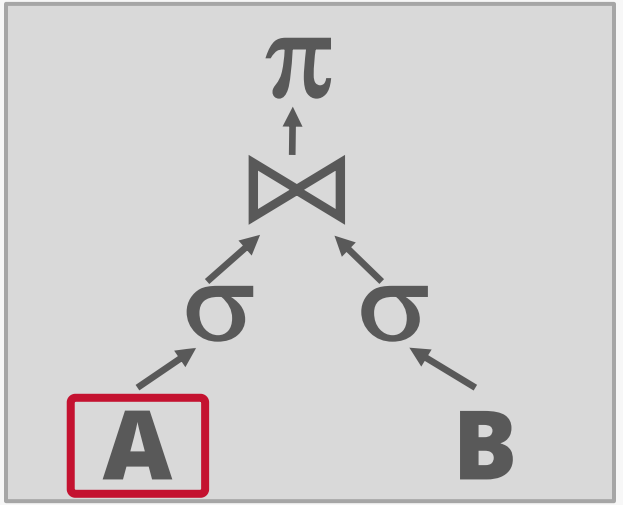
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



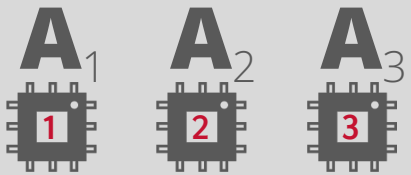
INTRA-OPERATOR PARALLELISM

A_1 A_2 A_3

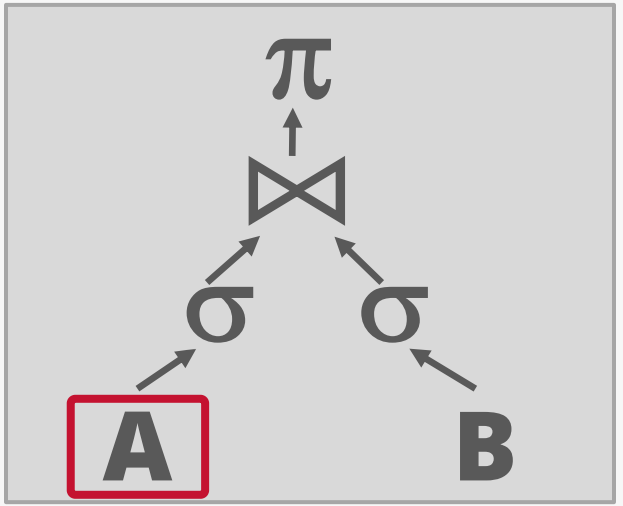
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



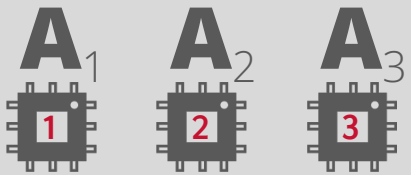
INTRA-OPERATOR PARALLELISM



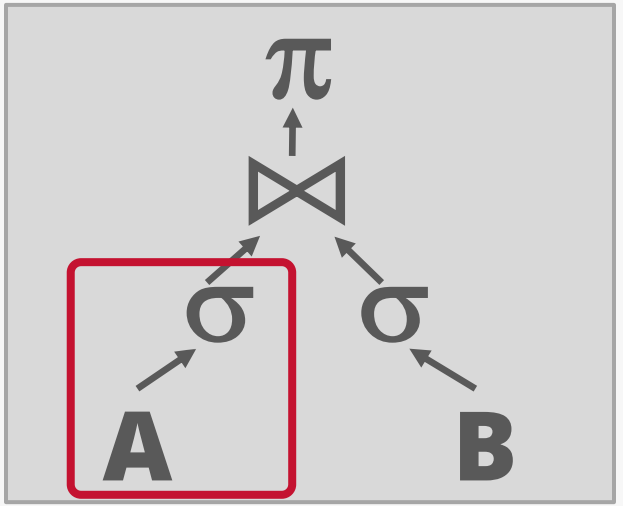

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



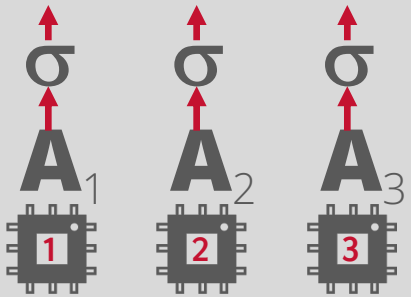
INTRA-OPERATOR PARALLELISM



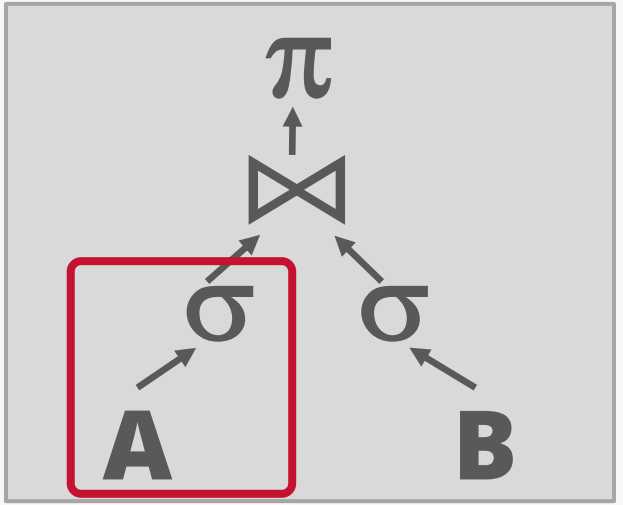

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



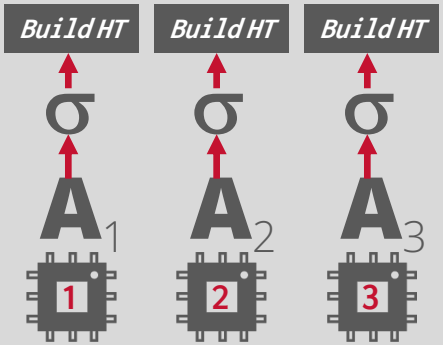
INTRA-OPERATOR PARALLELISM



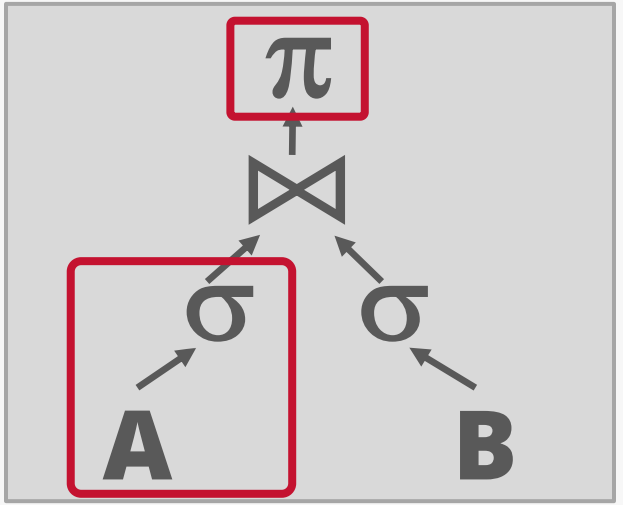

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



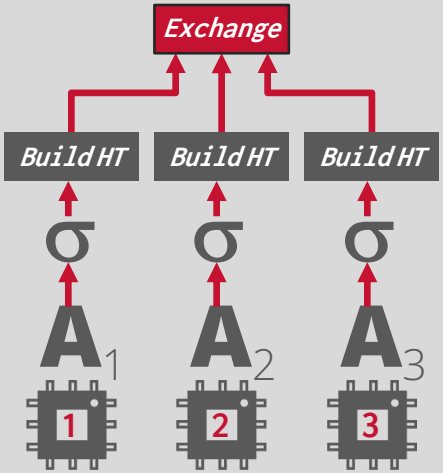
INTRA-OPERATOR PARALLELISM



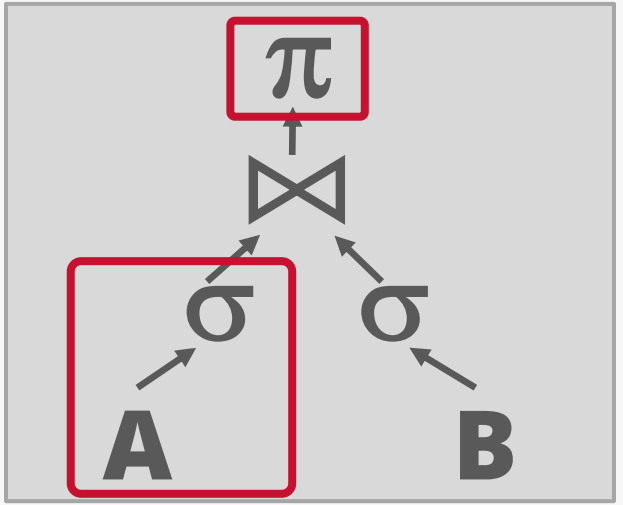

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



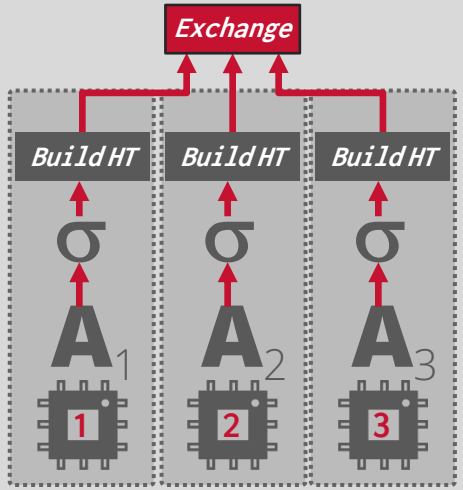
INTRA-OPERATOR PARALLELISM



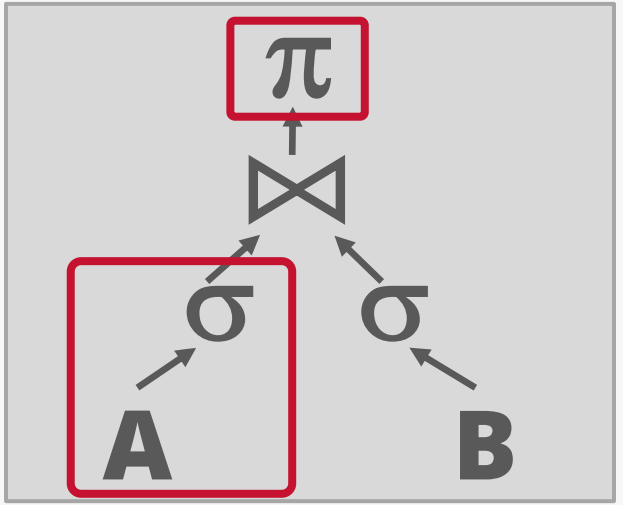

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



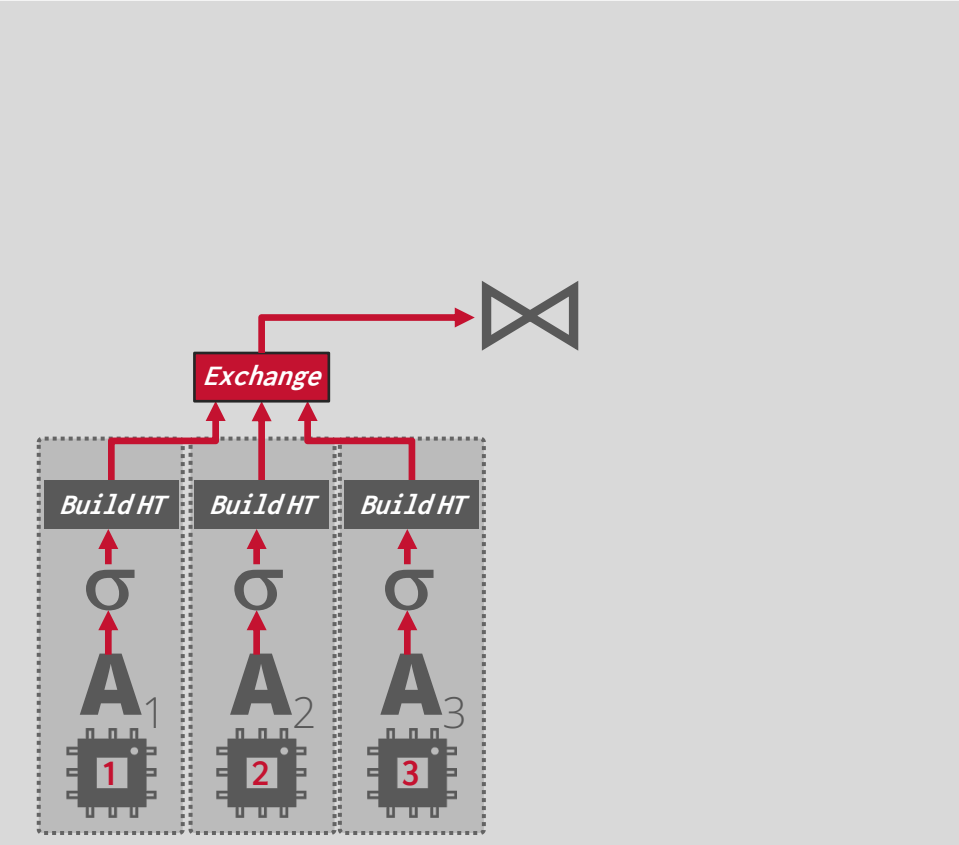
INTRA-OPERATOR PARALLELISM



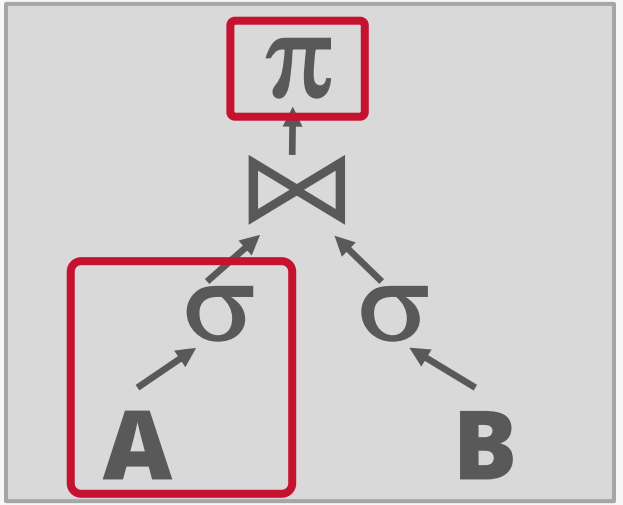

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



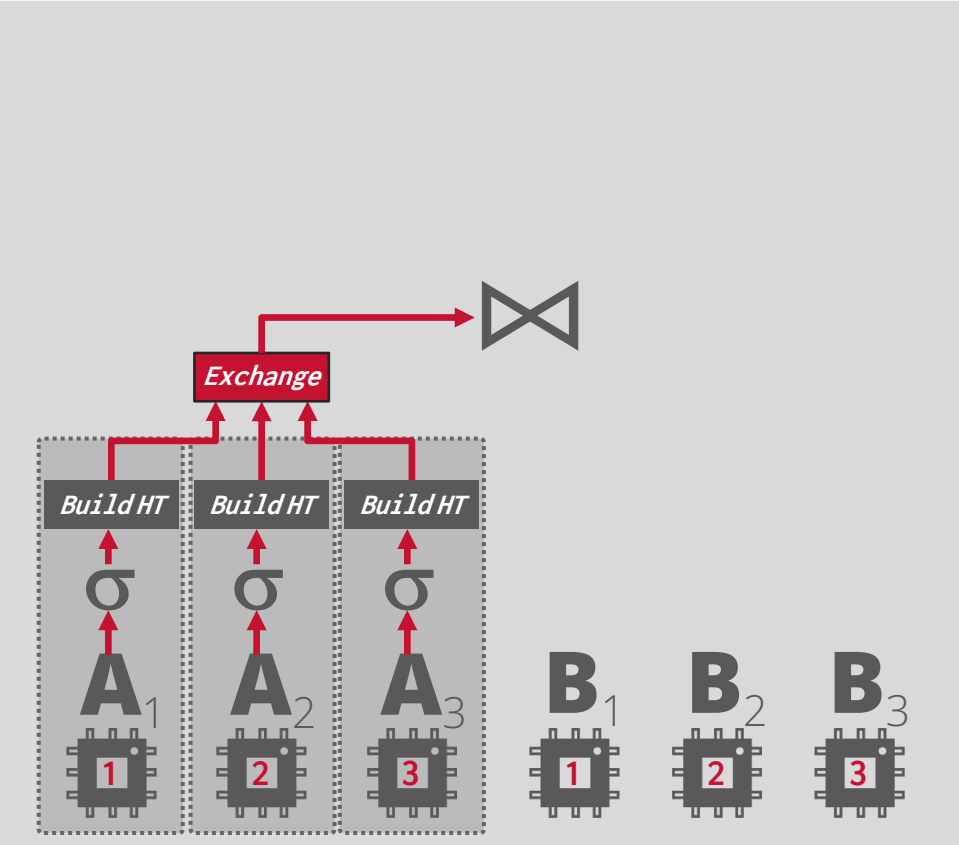
INTRA-OPERATOR PARALLELISM



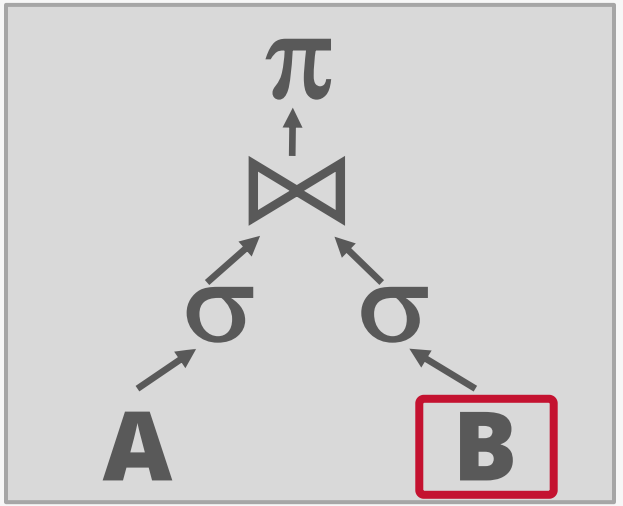

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



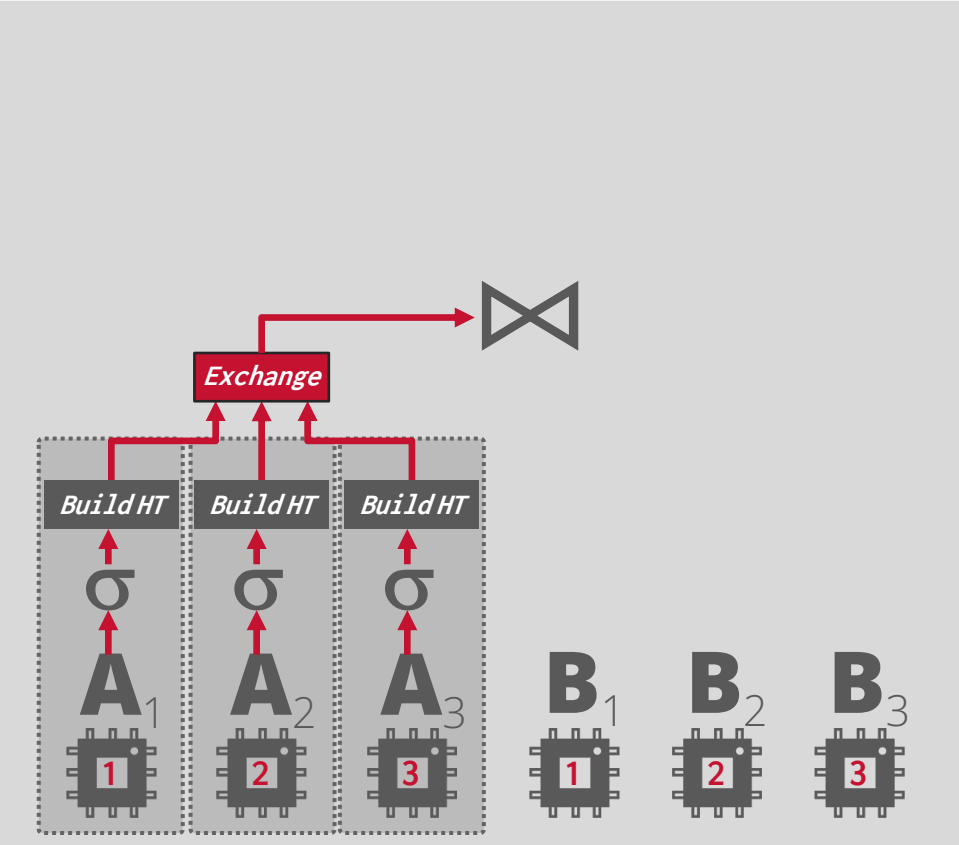
INTRA-OPERATOR PARALLELISM



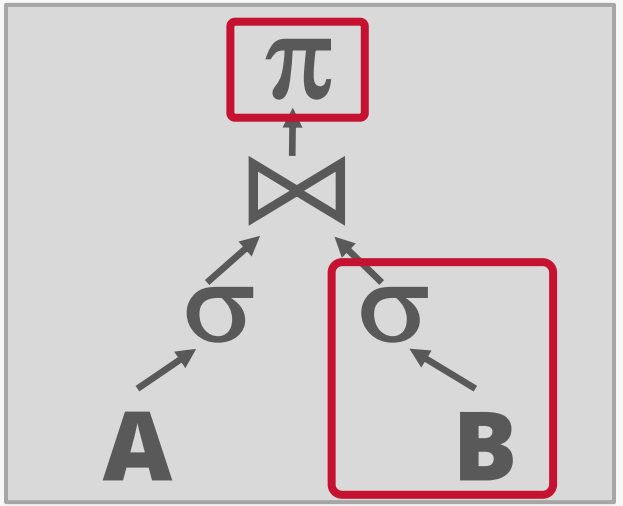

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



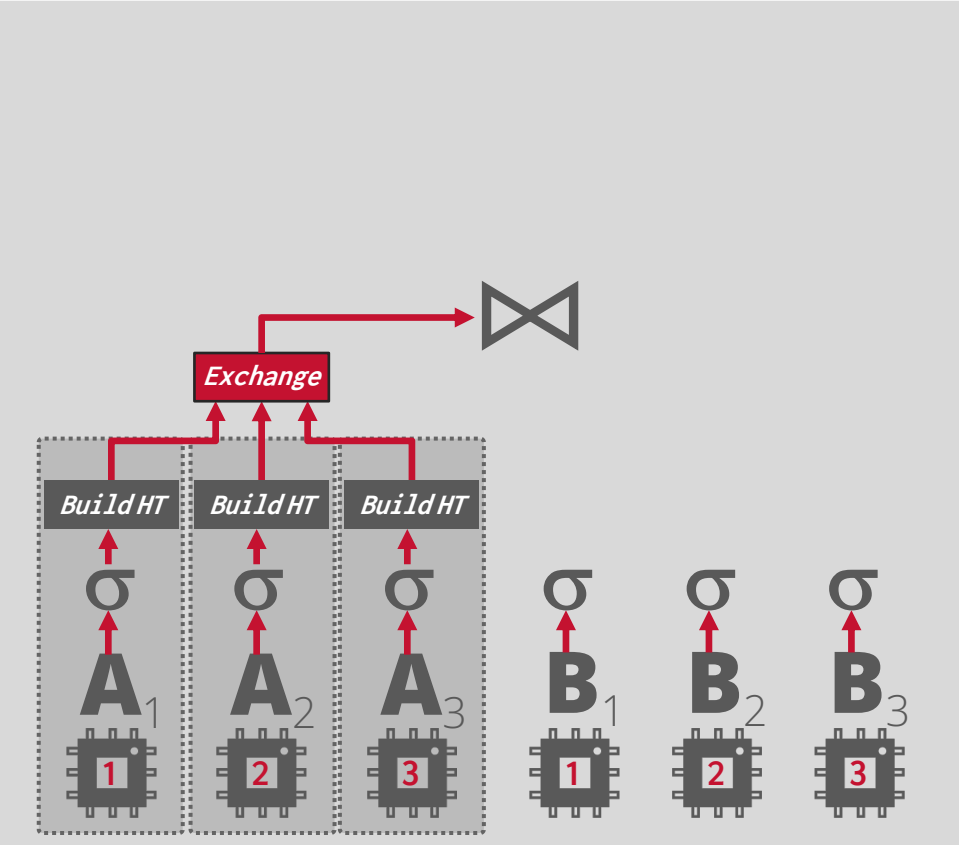
INTRA-OPERATOR PARALLELISM



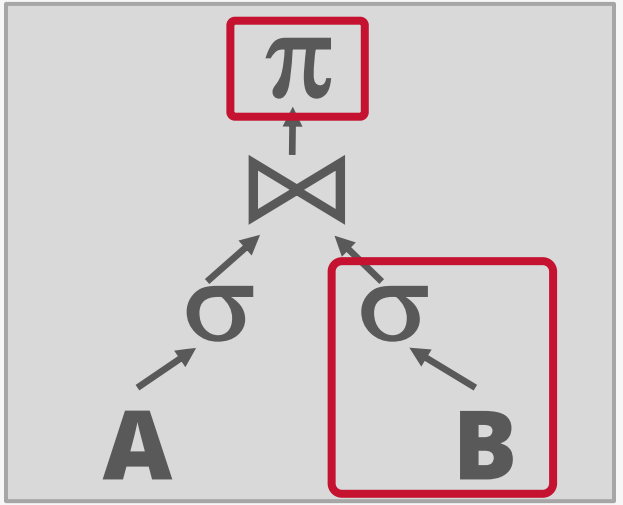

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



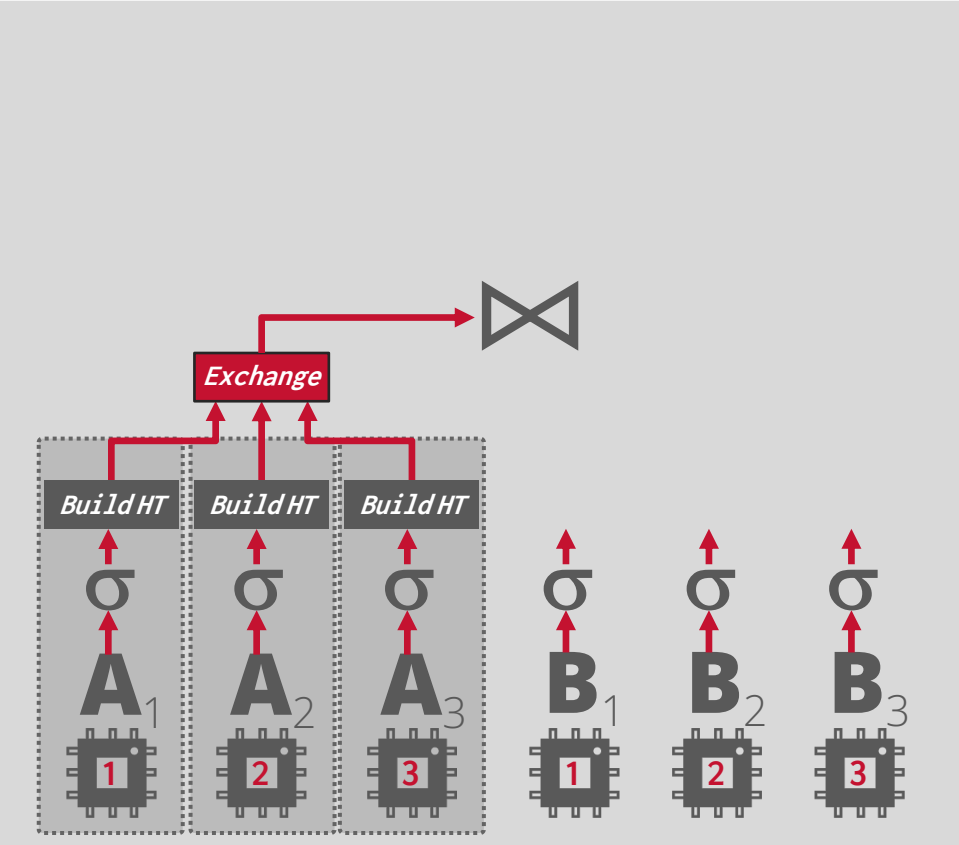
INTRA-OPERATOR PARALLELISM



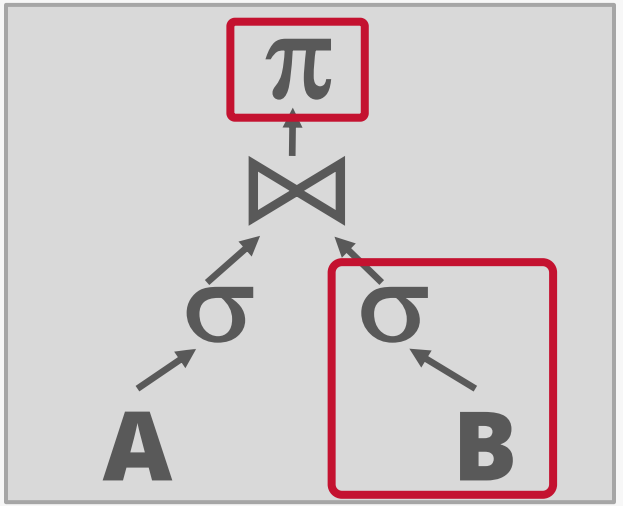

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



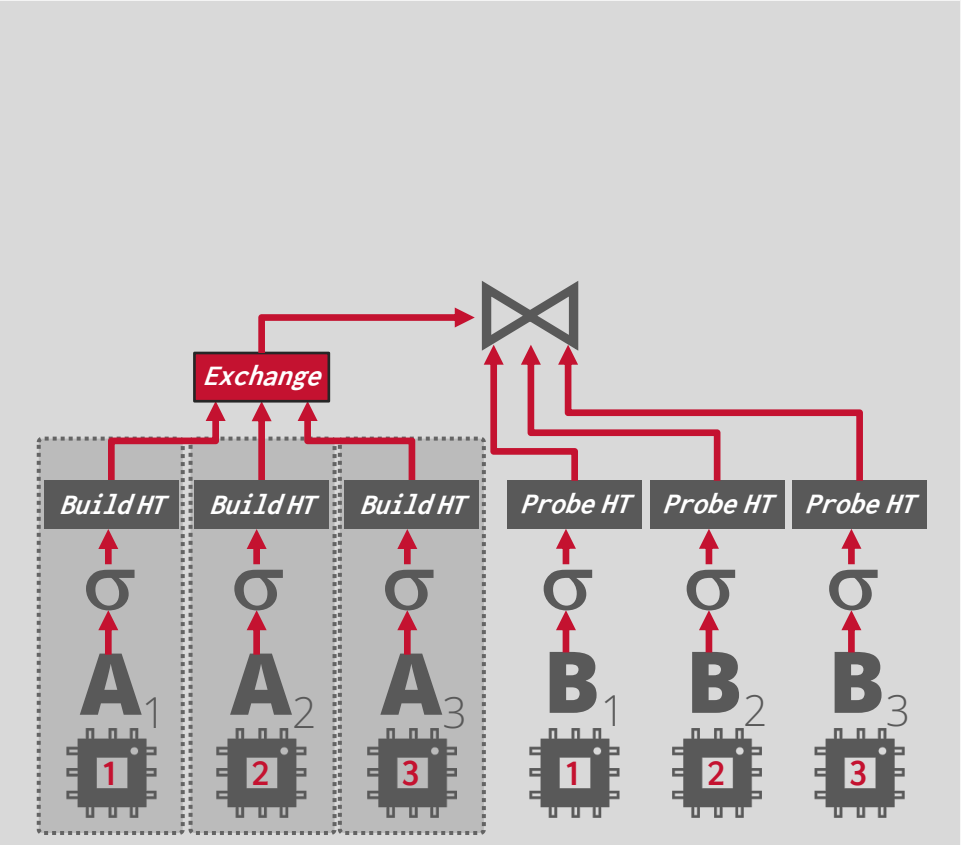
INTRA-OPERATOR PARALLELISM



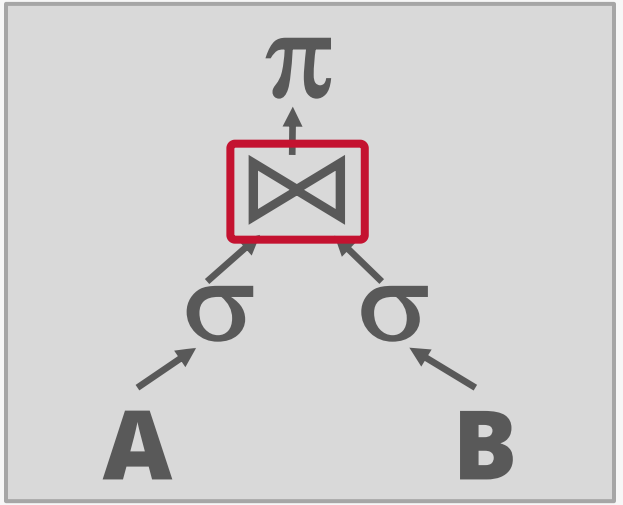

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



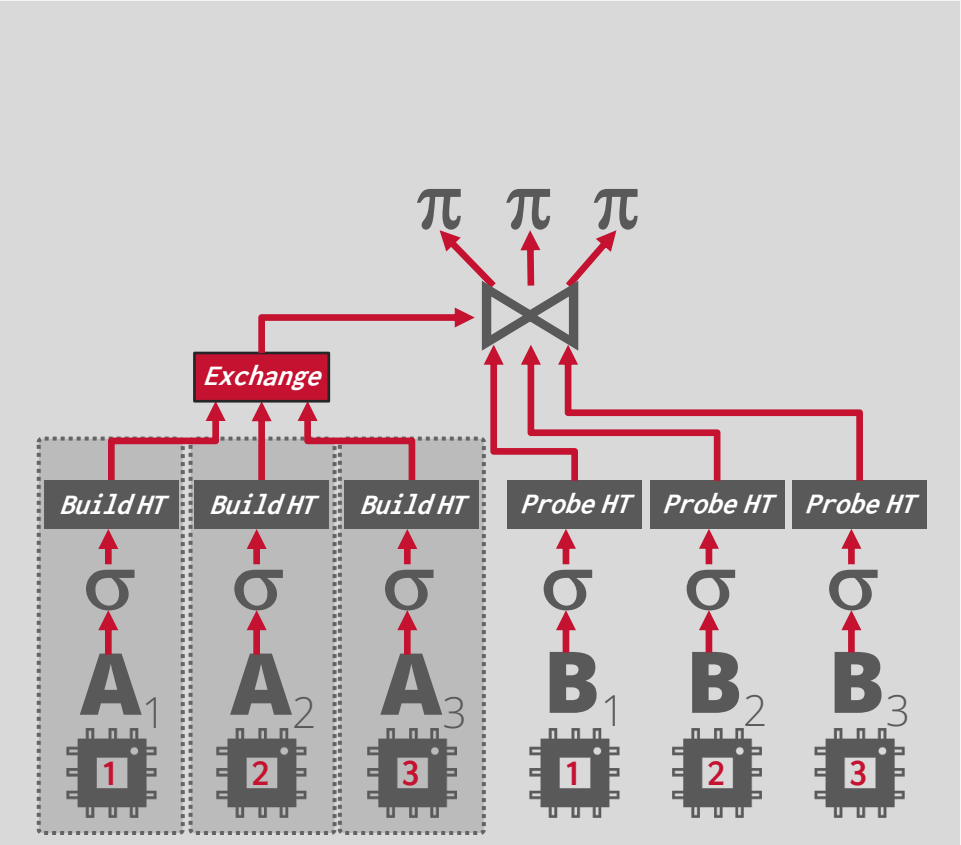
INTRA-OPERATOR PARALLELISM



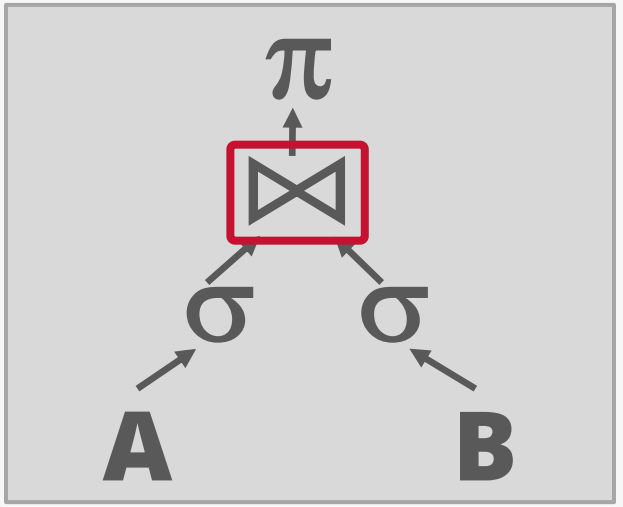

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



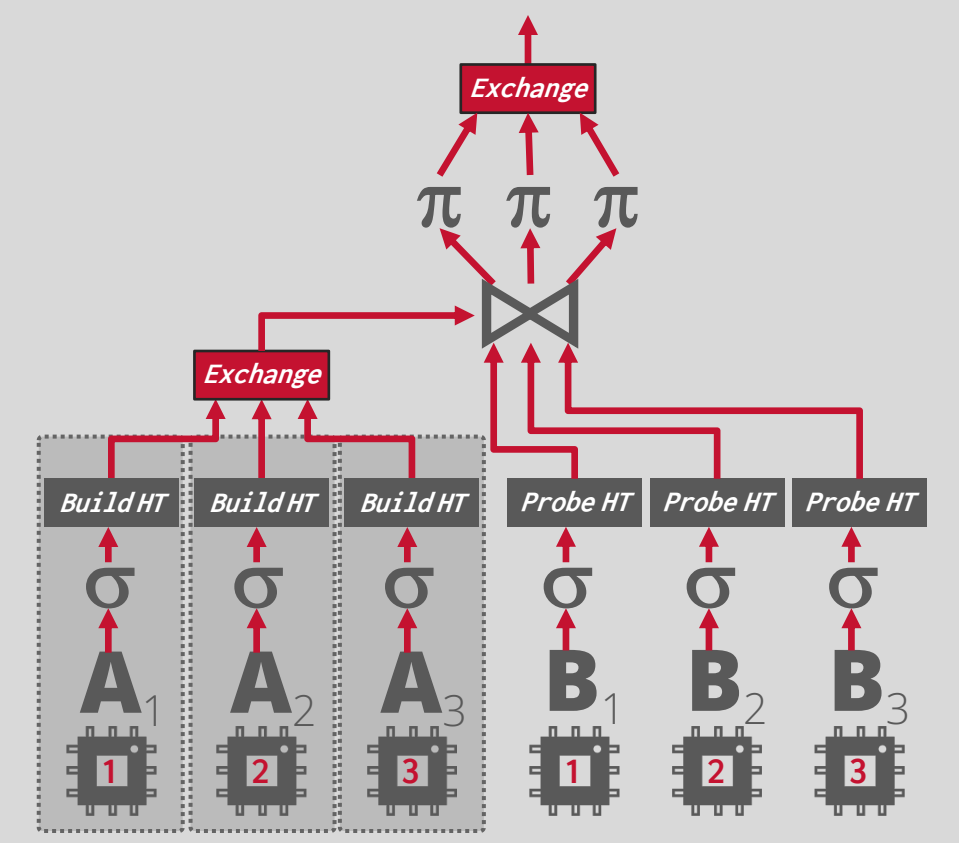
INTRA-OPERATOR PARALLELISM



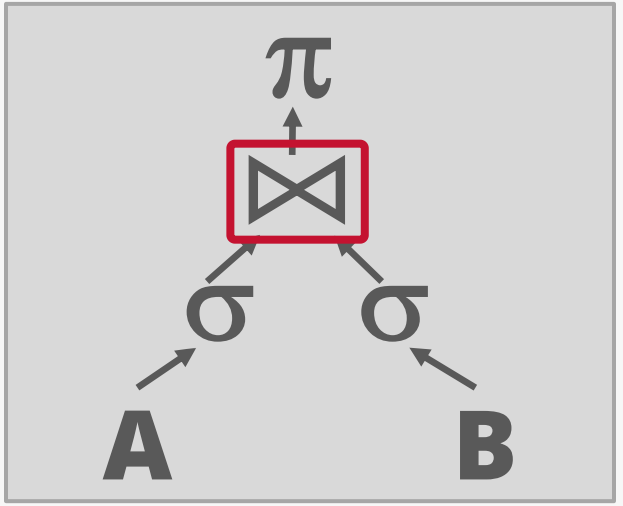

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



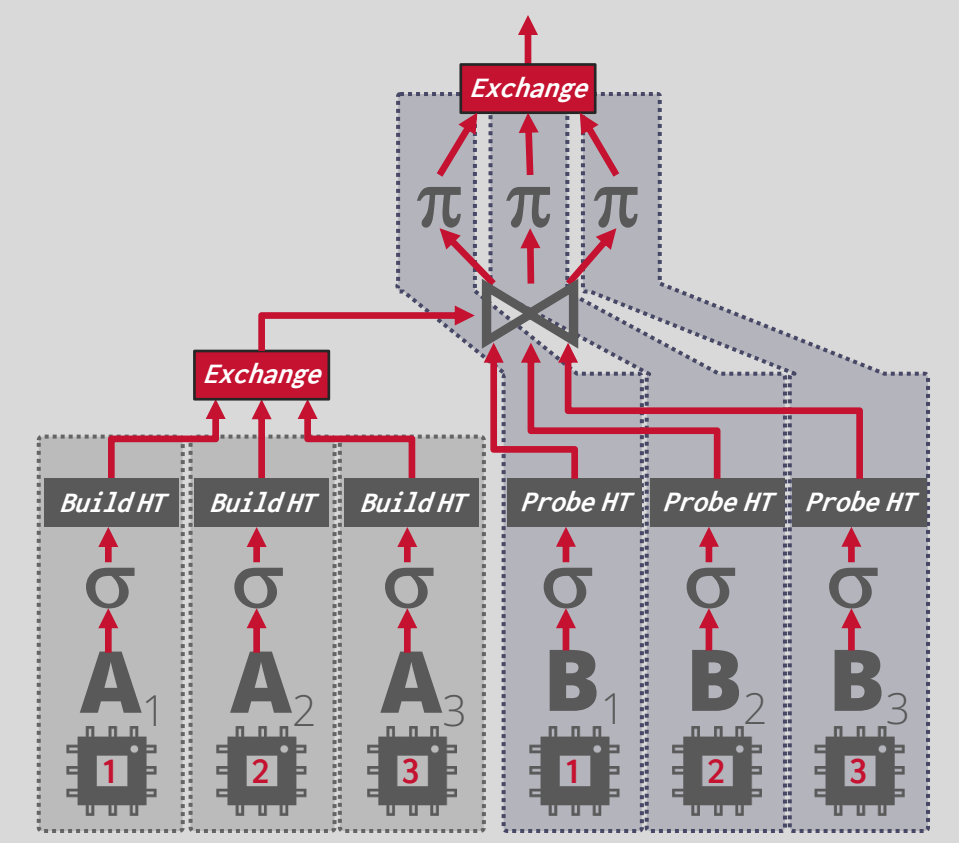
INTRA-OPERATOR PARALLELISM



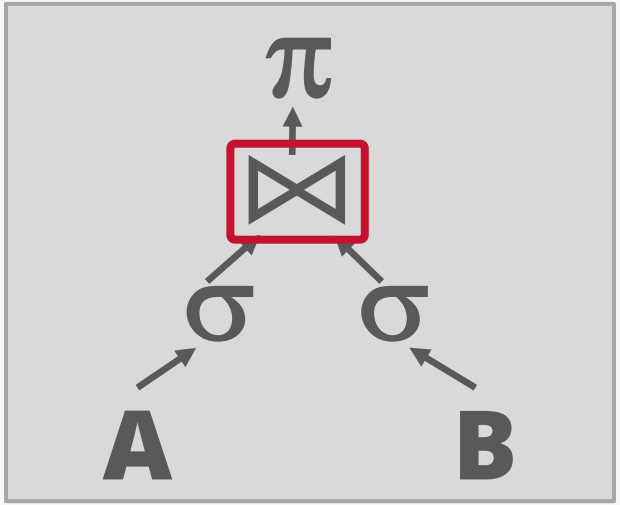

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



INTRA-OPERATOR PARALLELISM



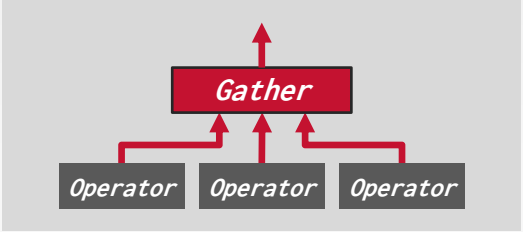

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



EXCHANGE OPERATOR

Exchange Type #1 – Gather

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

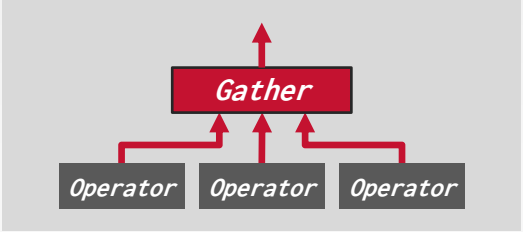


Source: Craig Freedman

EXCHANGE OPERATOR

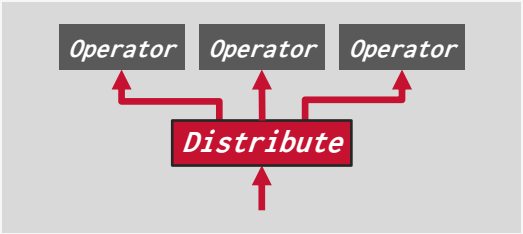
Exchange Type #1 – Gather

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



Exchange Type #2 – Distribute

→ Split a single input stream into multiple output streams.

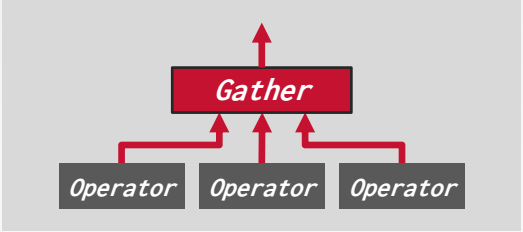


Source: Craig Freedman

EXCHANGE OPERATOR

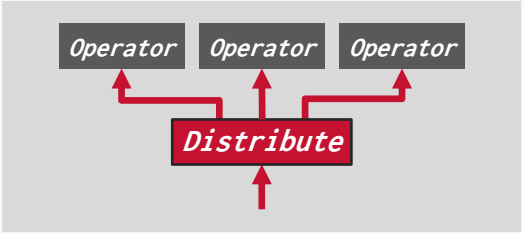
Exchange Type #1 – Gather

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



Exchange Type #2 – Distribute

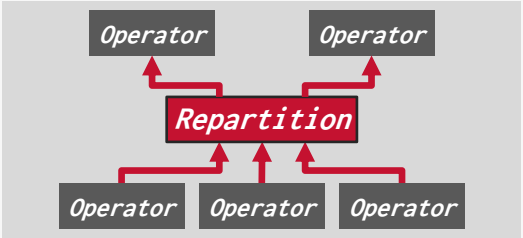
→ Split a single input stream into multiple output streams.



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**.







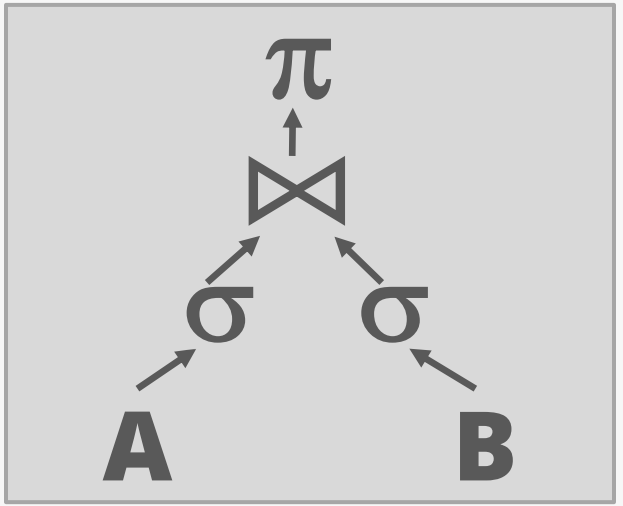






INTER-OPERATOR PARALLELISM

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

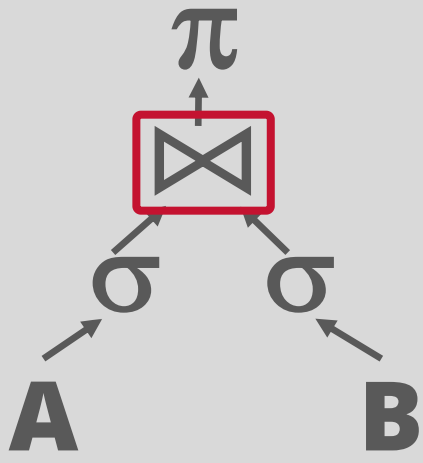


INTER-OPERATOR PARALLELISM

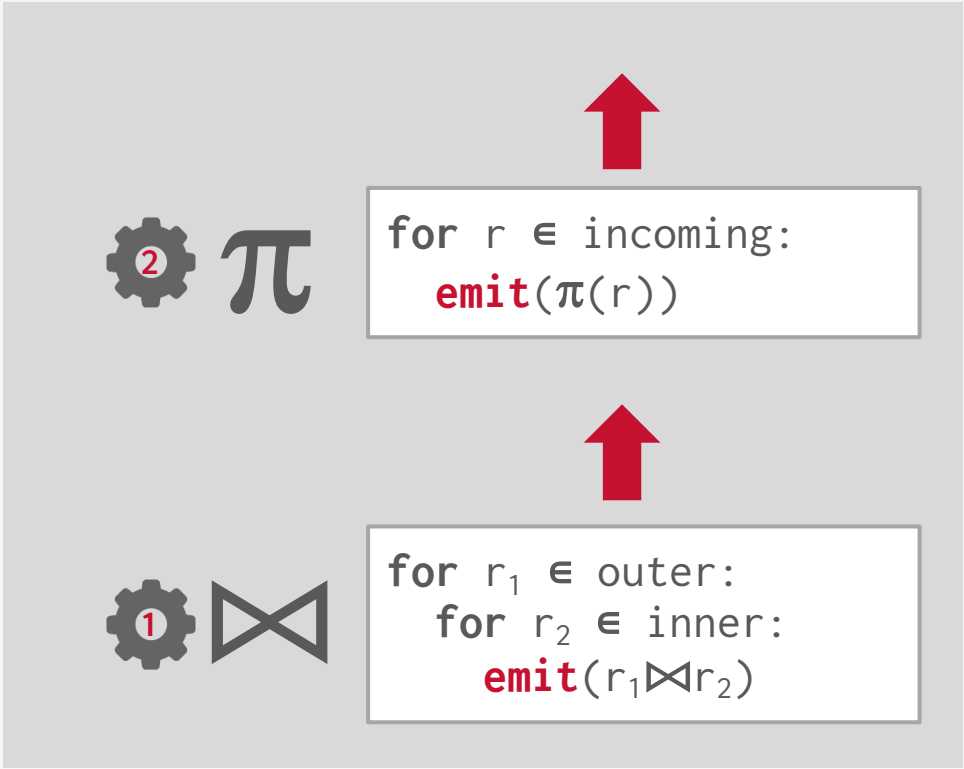



```
for r1 ∈ outer:
  for r2 ∈ inner:
    emit(r1⋈r2)
```

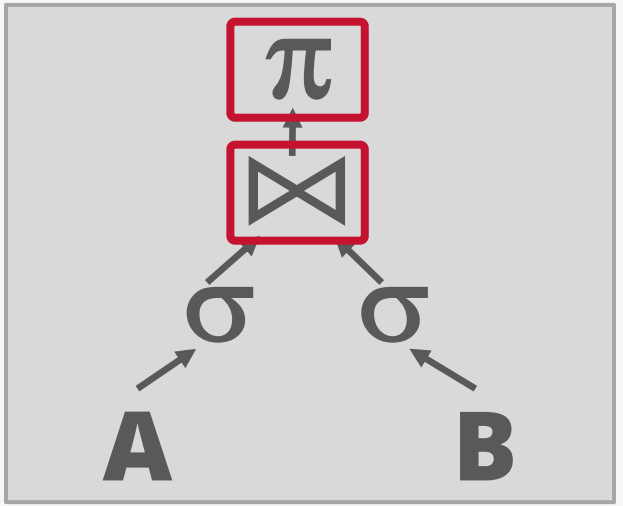
```
SELECT SLOW_UDF(B.value)
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



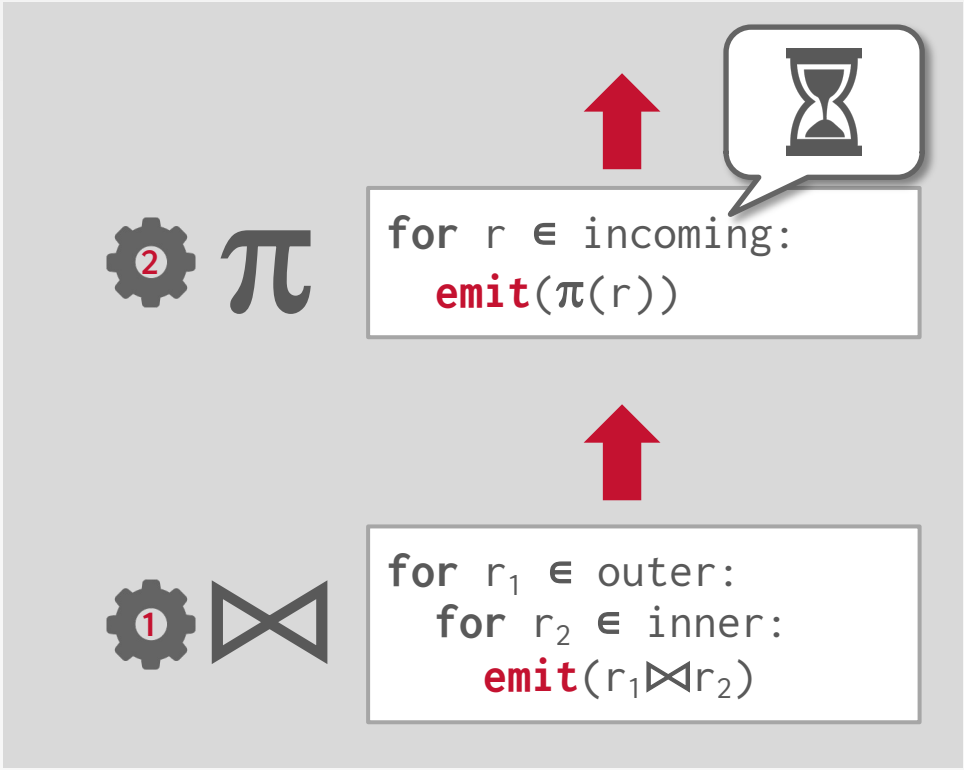
INTER-OPERATOR PARALLELISM



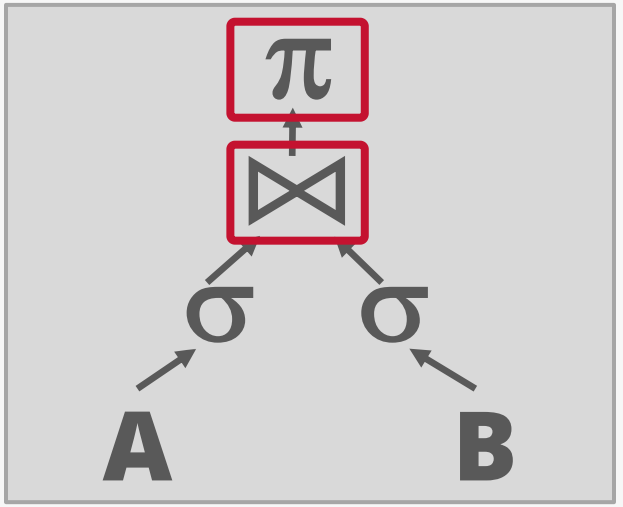

```
SELECT SLOW_UDF(B.value)
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



INTER-OPERATOR PARALLELISM




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



BUSHY PARALLELISM

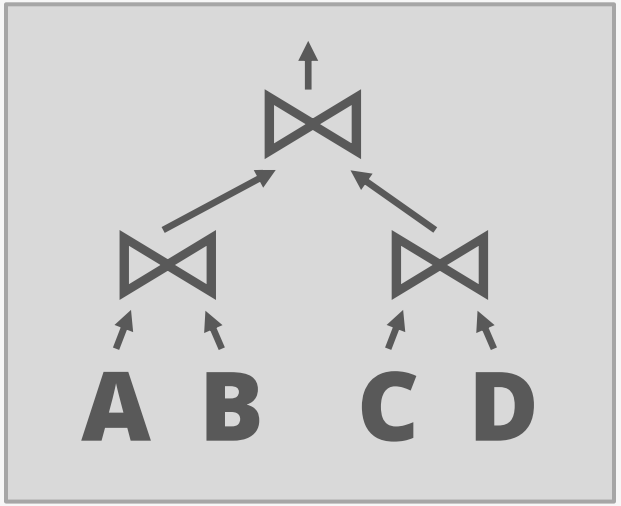
Approach #3: Bushy Parallelism

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

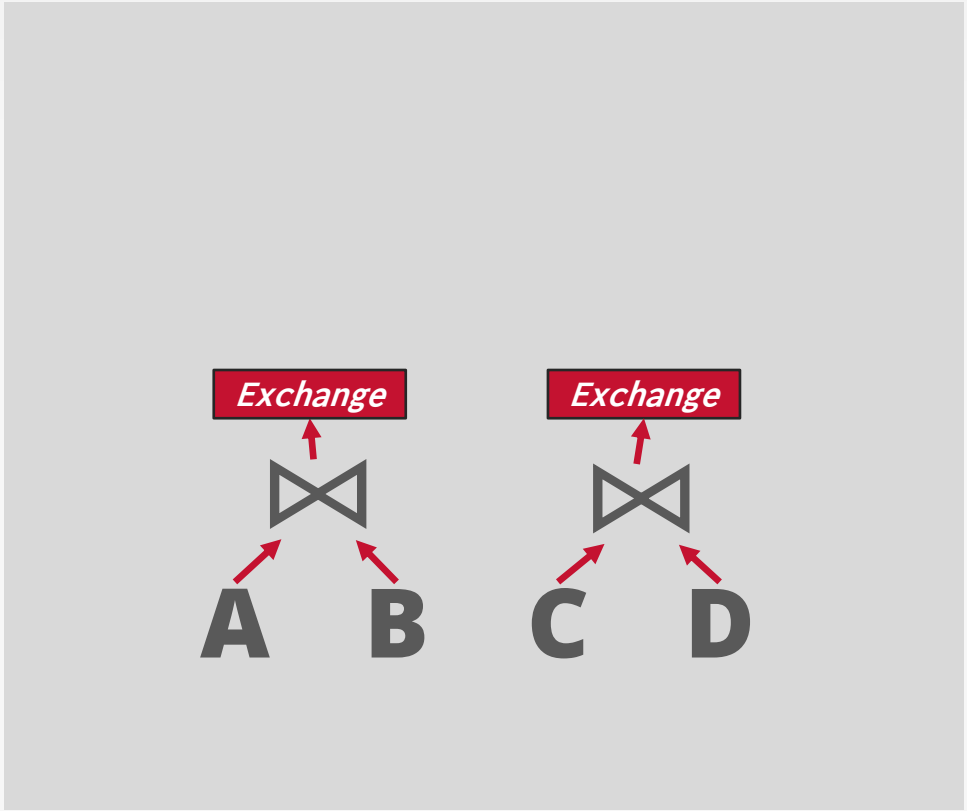
→ Still need exchange operators to combine intermediate results from segments.

BUSHY PARALLELISM

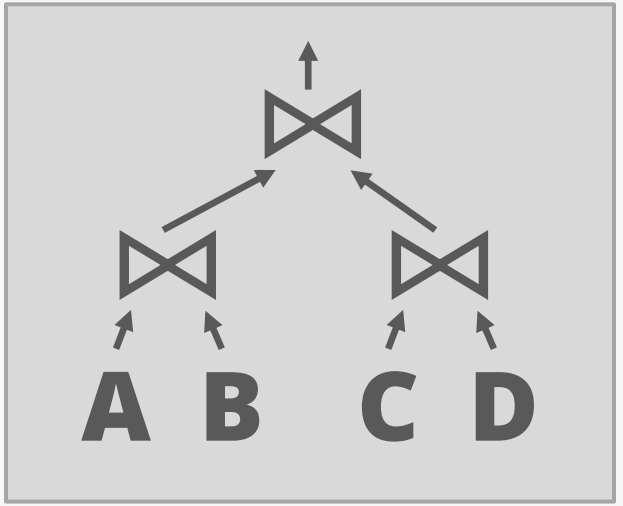
```
SELECT *
  FROM A
  JOIN B
  JOIN C
  JOIN D
```



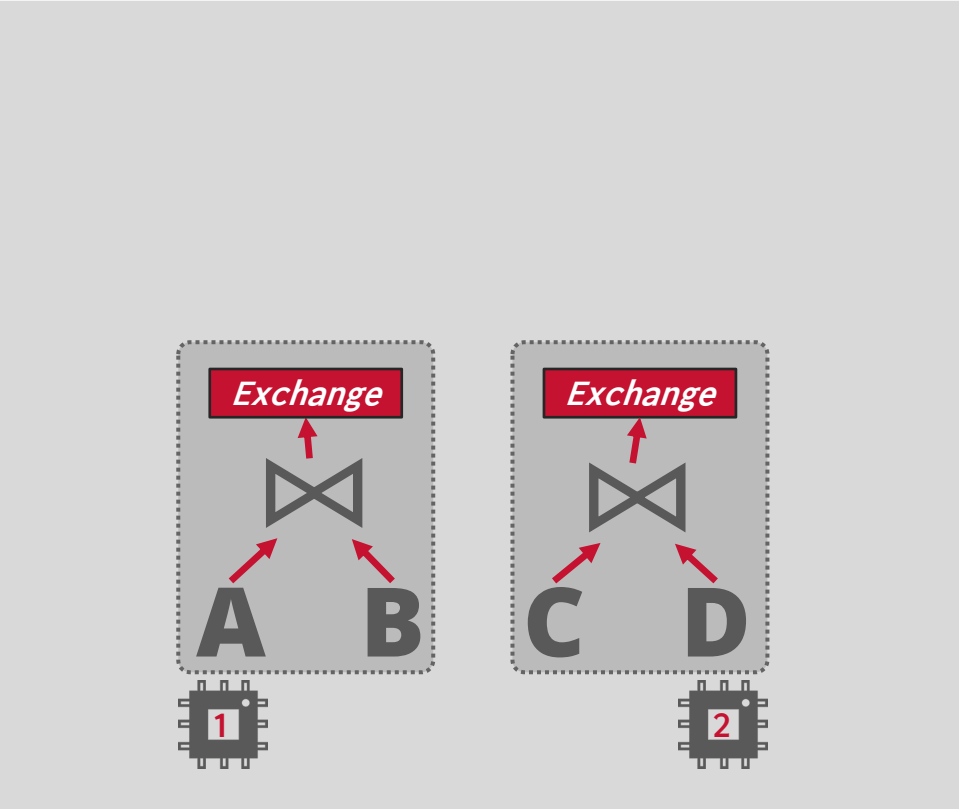
BUSHY PARALLELISM



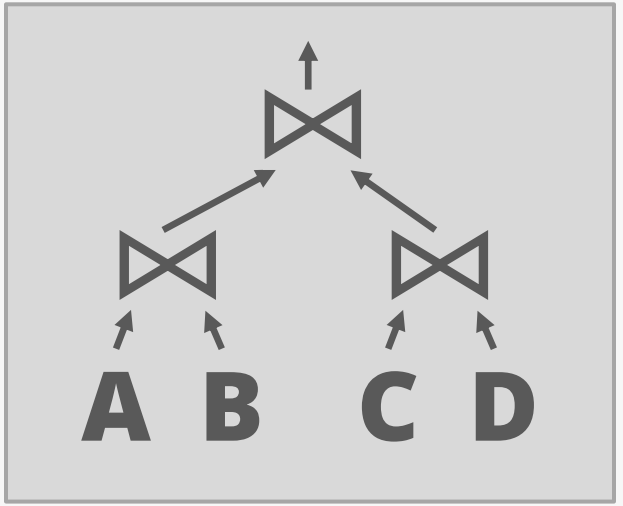

```
SELECT *
  FROM A
  JOIN B
  JOIN C
  JOIN D
```



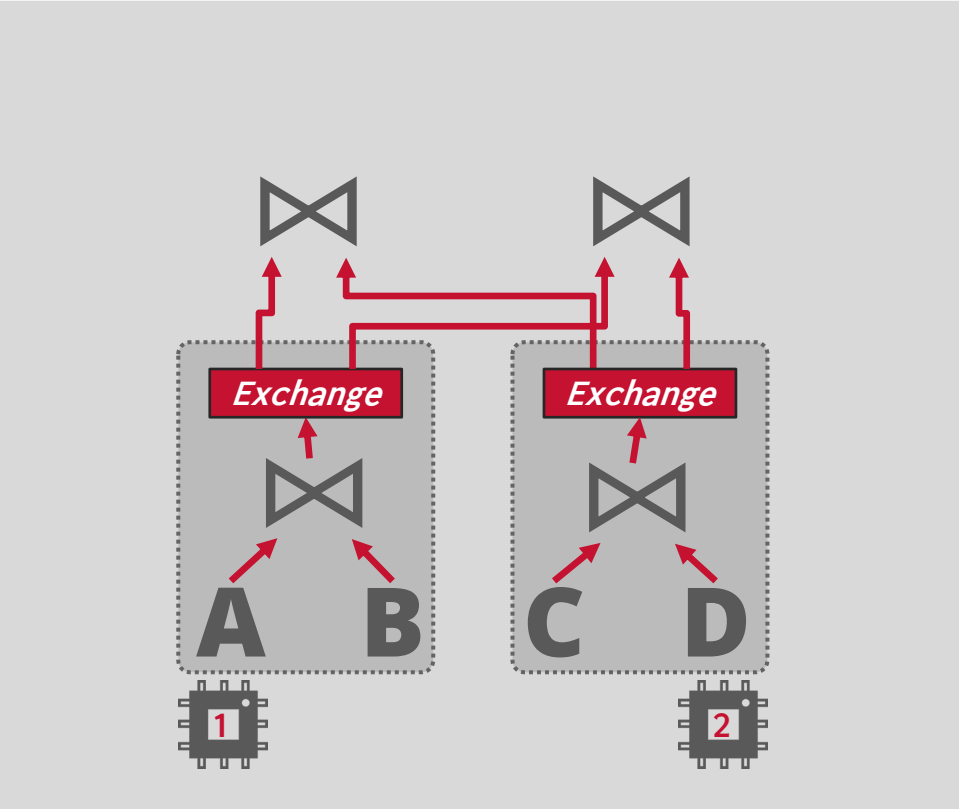
BUSHY PARALLELISM



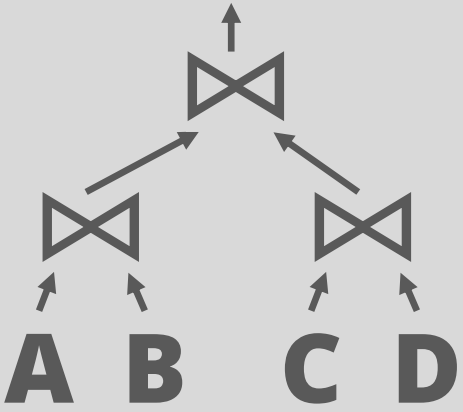

```
SELECT *
  FROM A
  JOIN B
  JOIN C
  JOIN D
```



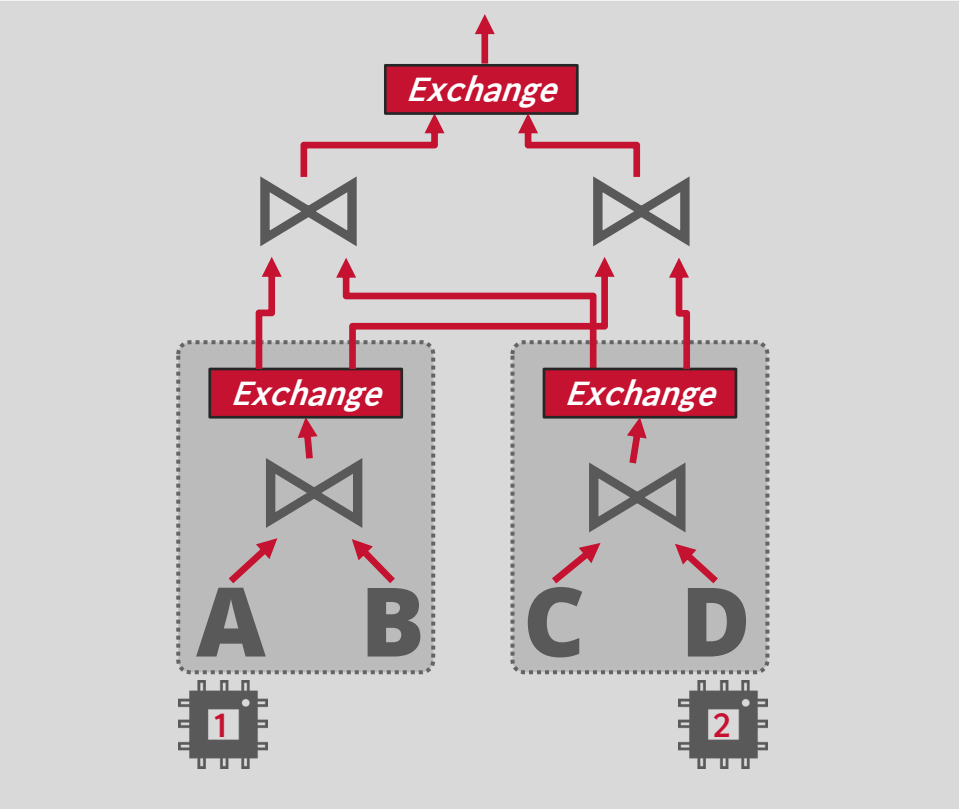
BUSHY PARALLELISM



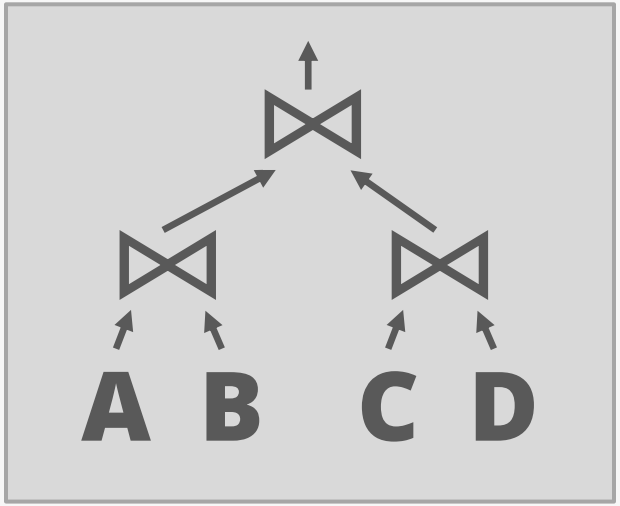

```
SELECT *
  FROM A
  JOIN B
  JOIN C
  JOIN D
```



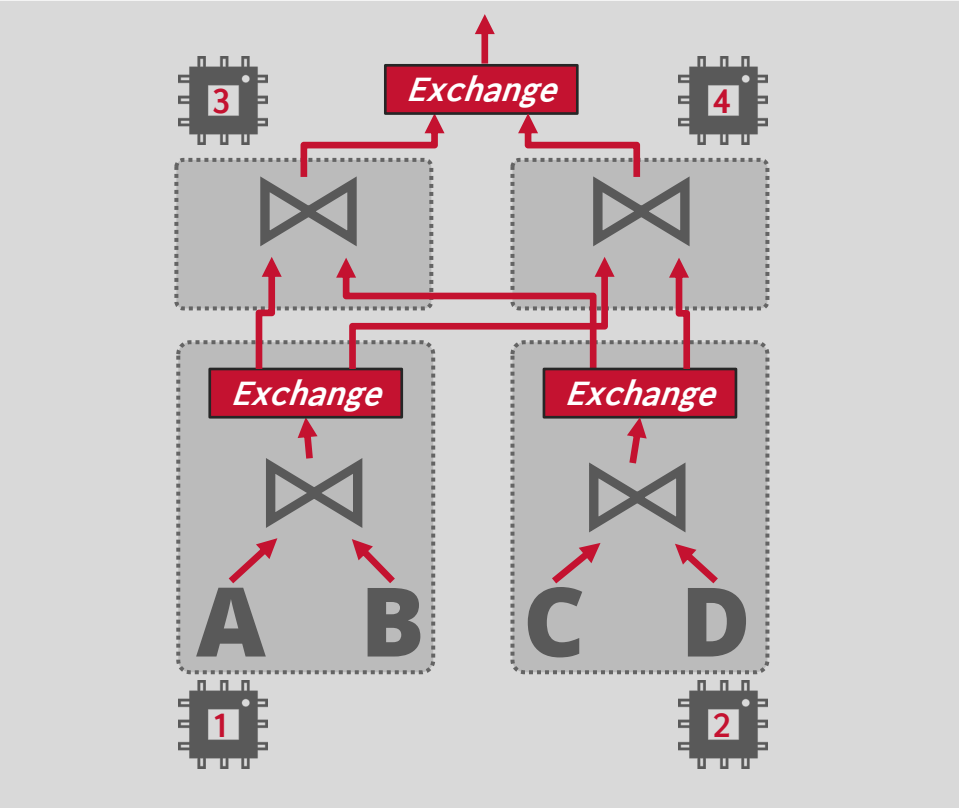
BUSHY PARALLELISM



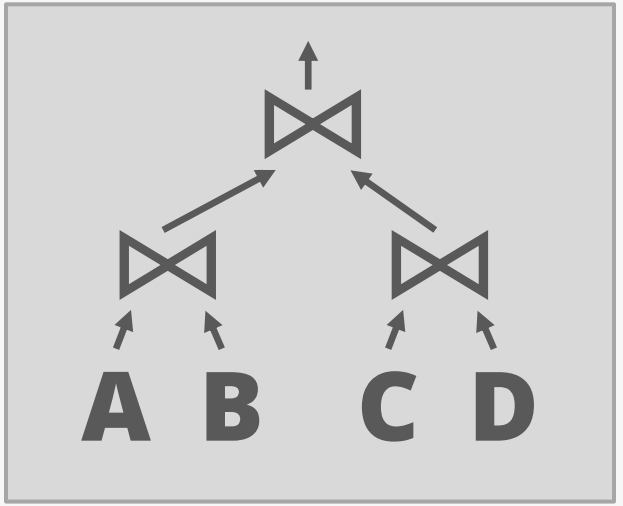

```
SELECT *
  FROM A
  JOIN B
  JOIN C
  JOIN D
```



BUSHY PARALLELISM




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



SEQUENTIAL SCAN: OPTIMIZATIONS

Lecture #05 Data Encoding / Compression

Lecture #06 Prefetching / Scan Sharing / Buffer Bypass

Lecture #14 Task Parallelization / Multi-threading

Lecture #08 Clustering / Sorting

Lecture #12 Late Materialization

Materialized Views / Result Caching

Lecture #13 Data Skipping

Data Parallelization / Vectorization

Lecture #13 Code Specialization / Compilation

SEQUENTIAL SCAN: OPTIMIZATIONS

Lecture #05 Data Encoding / Compression

Lecture #06 Prefetching / Scan Sharing / Buffer Bypass

Lecture #14 Task Parallelization / Multi-threading

Lecture #08 Clustering / Sorting

Lecture #12 Late Materialization

Materialized Views / Result Caching

Lecture #13 Data Skipping

Data Parallelization / Vectorization

Lecture #13 Code Specialization / Compilation

DATA PARALLELISM

Modern CPUs provide vector registers that allow a single instruction to operate on multiple values (SIMD) packed into a register.

Convert a DBMS's scalar operator implementations that processes a single pair of operands at a time, to vectorized implementations that processes one operation on multiple pairs of operands at once.

SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```

X

8
7
6
5
4
3
2
1

Y

1
1
1
1
1
1
1
1

Z

SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```



SISD

+

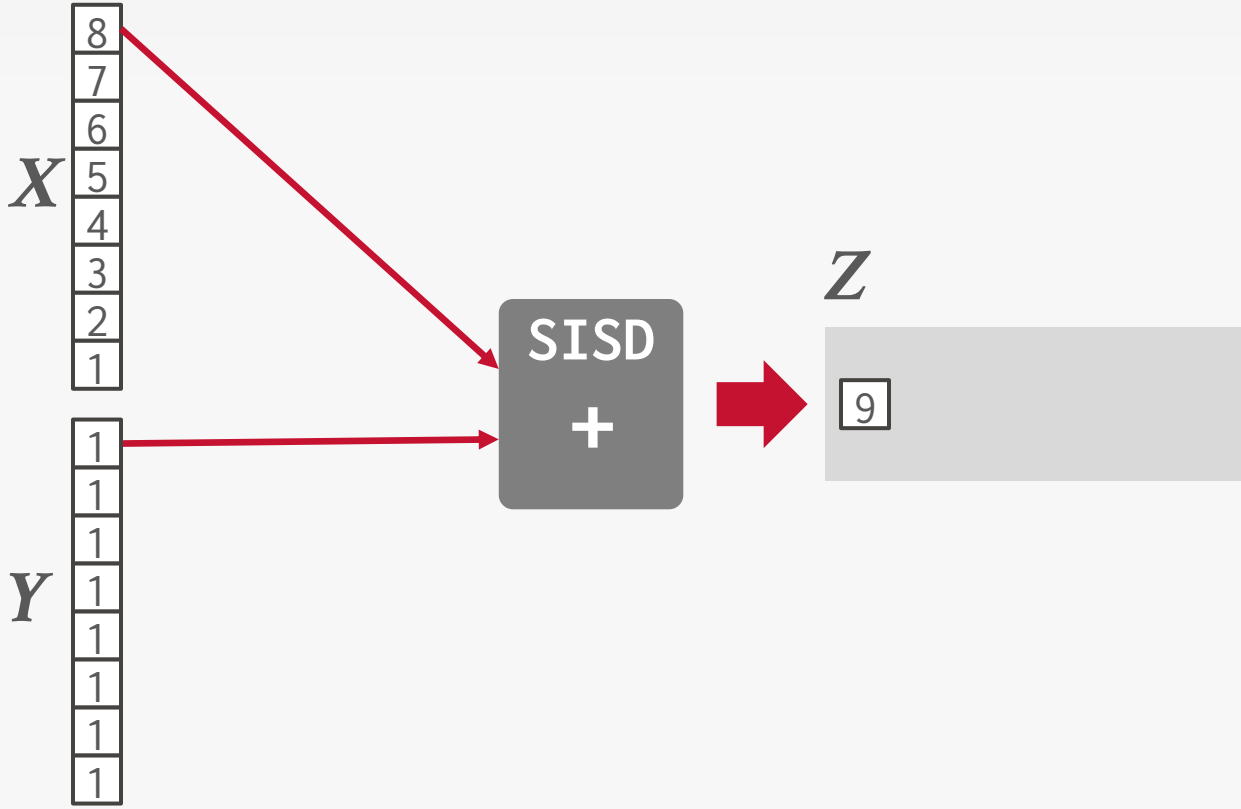
Z

SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```



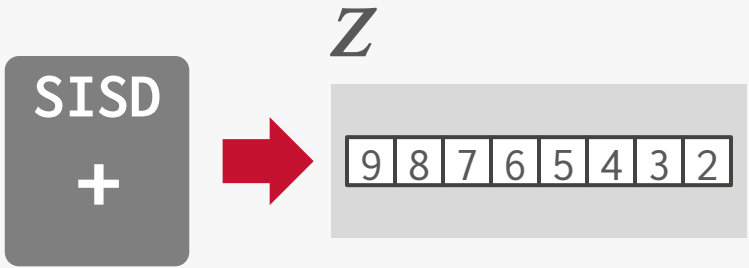
SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```





SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```

X

8
7
6
5
4
3
2
1

Y

1
1
1
1
1
1
1
1

Z

SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```



SIMD +

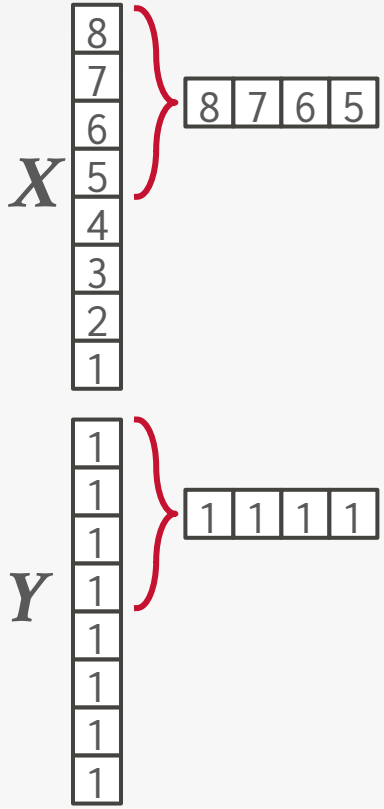
Z

SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```



SIMD

+

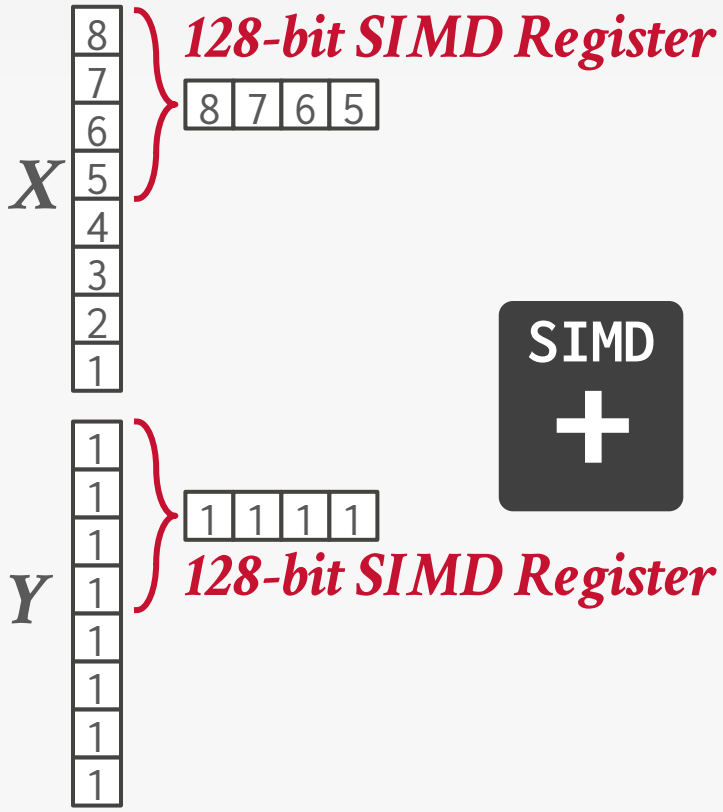
Z

SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```



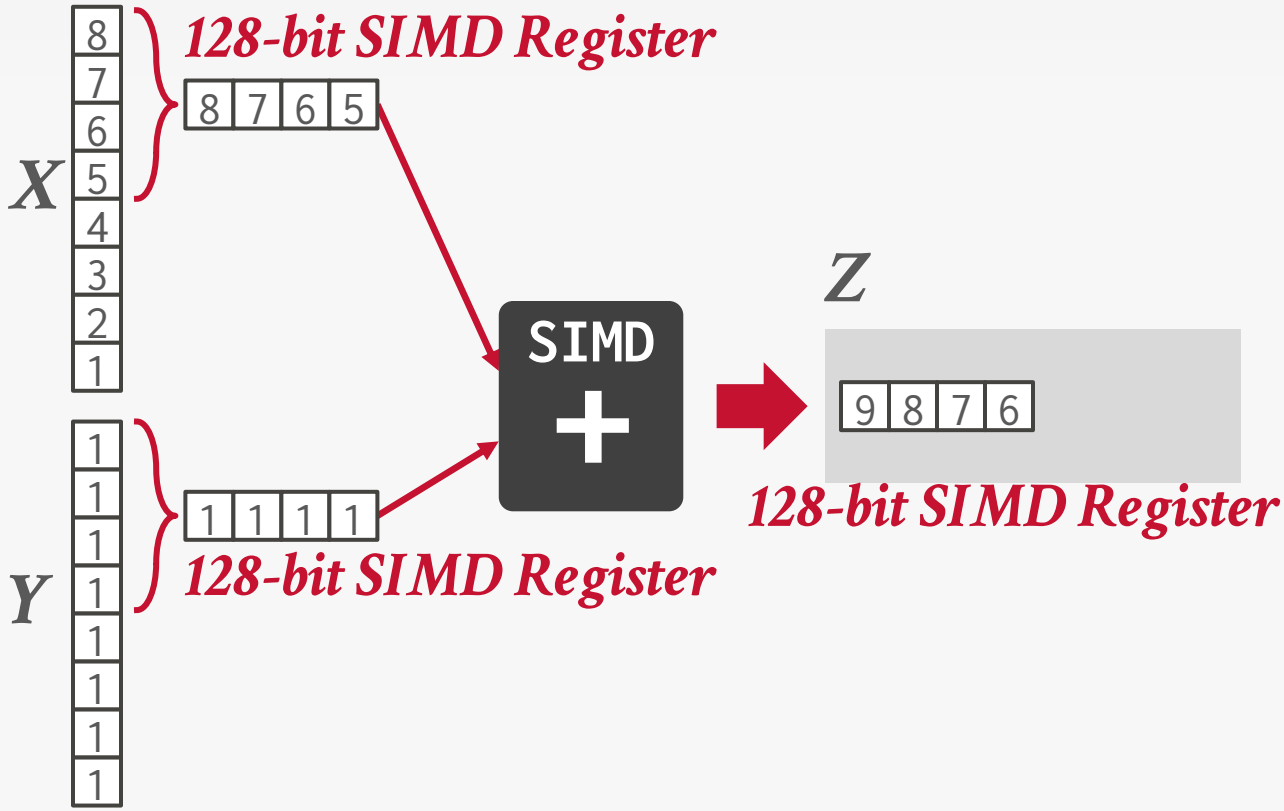
Z

SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```

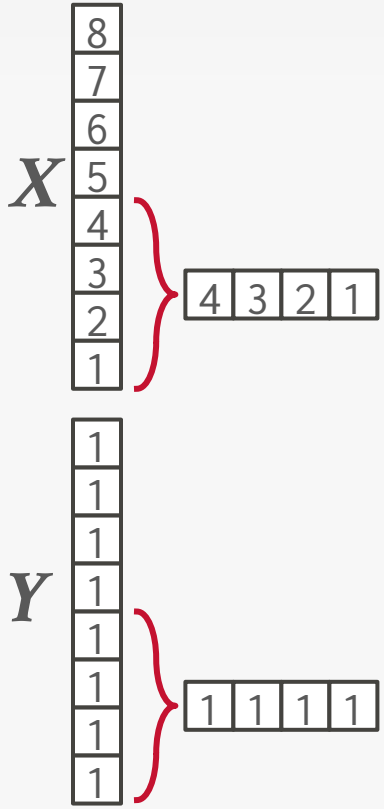


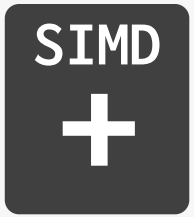
SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```





Z

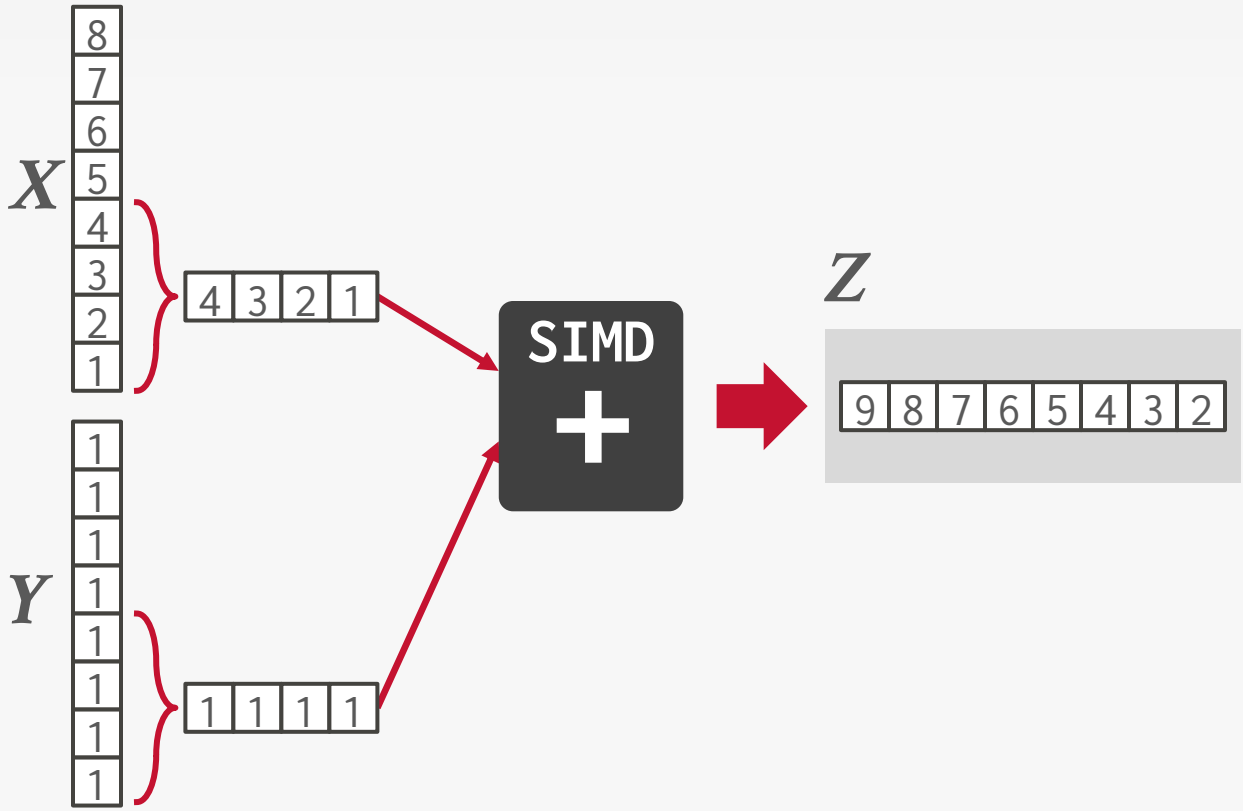
9 8 7 6

SIMD EXAMPLE

$$X + Y = Z$$

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} + \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} x_1+y_1 \\ x_2+y_2 \\ \vdots \\ x_n+y_n \end{bmatrix}$$

```
for (i=0; i<n; i++) {
  Z[i] = X[i] + Y[i];
}
```



SIMD SELECTION SCANS

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Scalar Scan

```
out = []
for t in table:
  key = t.key
  if (key>low) && (key<high):
    out.add(t)
  if |out|>n: emit(out)
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       (vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &
        (v_k≤high ? 1 : 0)
  simdStore(v_t, v_m, output[i])
  i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &
        (v_k≤high ? 1 : 0)
  simdStore(v_t, v_m, output[i])
  i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &
        (v_k≤high ? 1 : 0)
  simdStore(v_t, v_m, output[i])
  i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &
        (v_k≤high ? 1 : 0)
  simdStore(v_t, v_m, output[i])
  i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       (vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= $low AND key <= $high
```

SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       (vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'T'
```

SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       (vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

tid	key
100	F
101	O
102	C
103	U
104	S
105	I
106	H
107	G

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'T'
```

SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       (vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'T'
```

tid	key
100	F
101	O
102	C
103	U
104	S
105	I
106	H
107	G

Key Vector

F	O	C	U	S	I	N	G

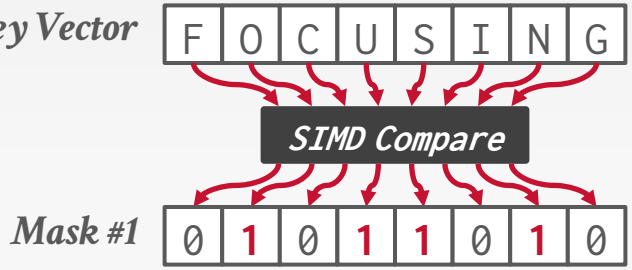
SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       (vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'T'
```

tid	key
100	F
101	O
102	C
103	U
104	S
105	I
106	H
107	G



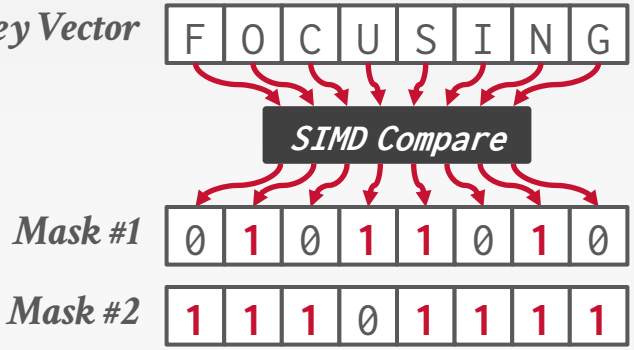
SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       (vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'T'
```

tid	key
100	F
101	O
102	C
103	U
104	S
105	I
106	H
107	G



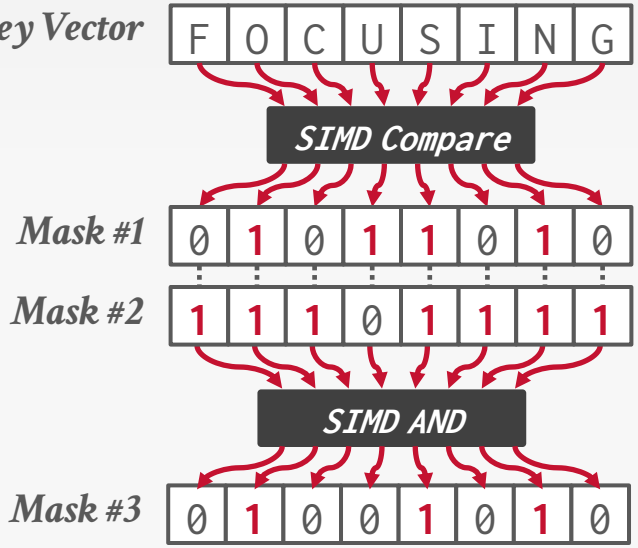
SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       (vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'T'
```

tid	key
100	F
101	O
102	C
103	U
104	S
105	I
106	H
107	G



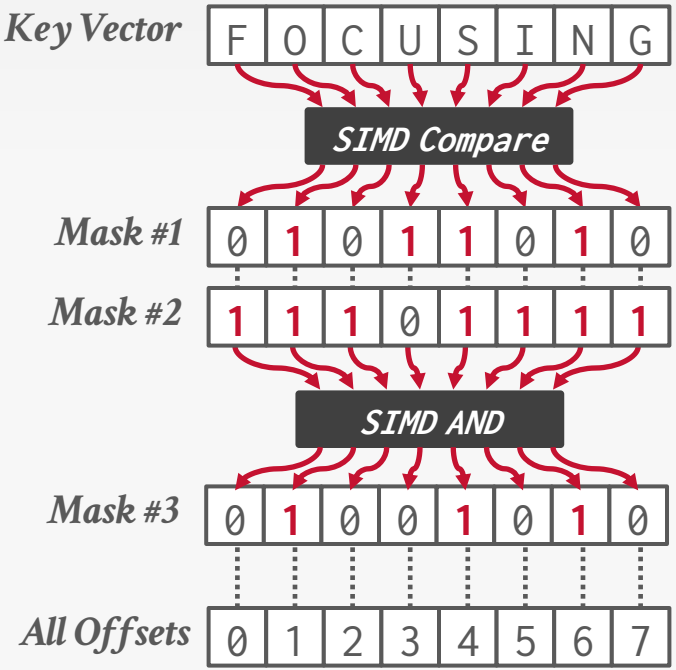
SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &
        (v_k≤high ? 1 : 0)
  simdStore(v_t, v_m, output[i])
  i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'T'
```

tid	key
100	F
101	O
102	C
103	U
104	S
105	I
106	H
107	G



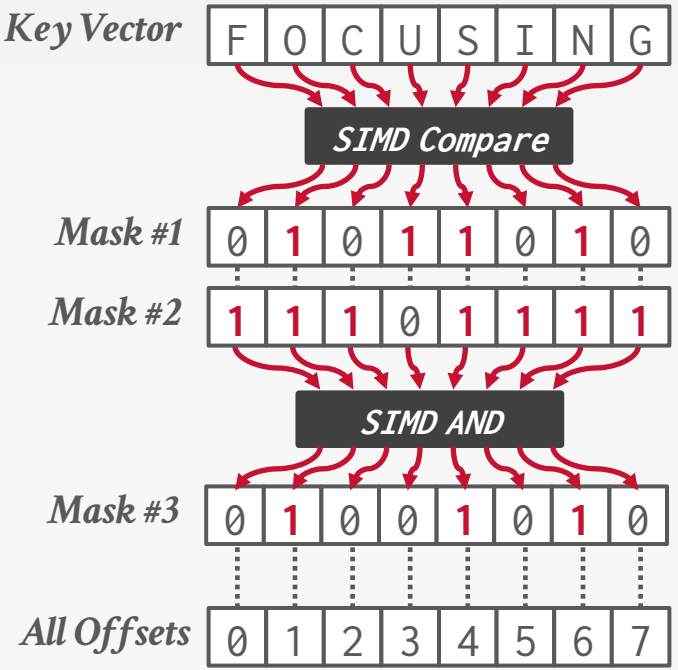
SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for v_t in table:
  simdLoad(v_t.key, v_k)
  v_m = (v_k≥low ? 1 : 0) &
        (v_k≤high ? 1 : 0)
  simdStore(v_t, v_m, output[i])
  i = i + |v_m≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'T'
```

Offset	tid	key
0	100	F
1	101	O
2	102	C
3	103	U
4	104	S
5	105	I
6	106	H
7	107	G



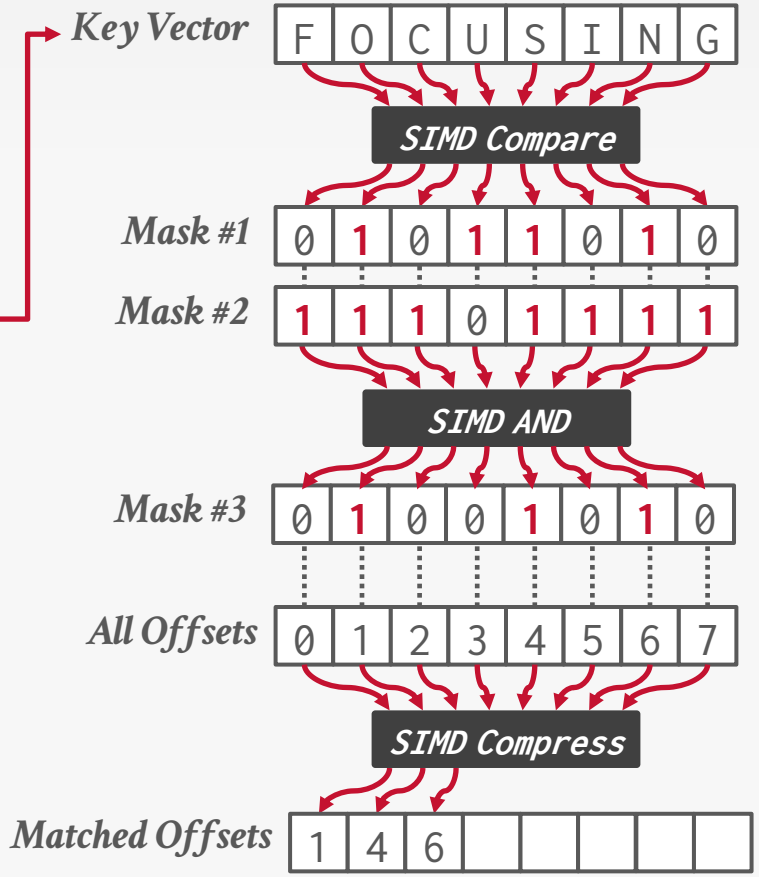
SIMD SELECTION SCANS

Vectorized Scan

```
i = 0
for vt in table:
  simdLoad(vt.key, vk)
  vm = (vk≥low ? 1 : 0) &
       (vk≤high ? 1 : 0)
  simdStore(vt, vm, output[i])
  i = i + |vm≠false|
```

```
SELECT * FROM table
 WHERE key >= 'N' AND key <= 'T'
```

Offset	tid	key
0	100	F
1	101	O
2	102	C
3	103	U
4	104	S
5	105	I
6	106	H
7	107	G



FILTER REPRESENTATION

```
WHERE col0 IS NULL OR col1 LIKE 'b%'
```

Approach #1: Selection Vectors

→ Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.

→ Pre-allocate selection vector as the max-size of the input vector.

FILTER REPRESENTATION

Approach #1: Selection Vectors

→ Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.

→ Pre-allocate selection vector as the max-size of the input vector.

```
WHERE col0 IS NULL OR col1 LIKE 'b%'
```

col0: int32

data	null?
55	0
66	0
77	0
-	1
88	0

col1: varchar

data	null?
aa	0
bb	0
-	1
cc	0
bbb	0

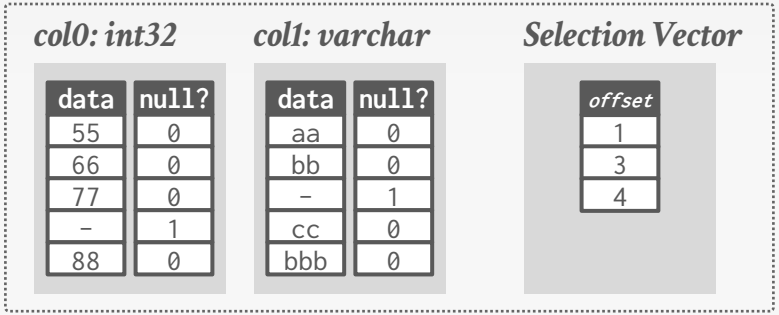
FILTER REPRESENTATION

Approach #1: Selection Vectors

→ Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.

→ Pre-allocate selection vector as the max-size of the input vector.

```
WHERE col0 IS NULL OR col1 LIKE 'b%'
```



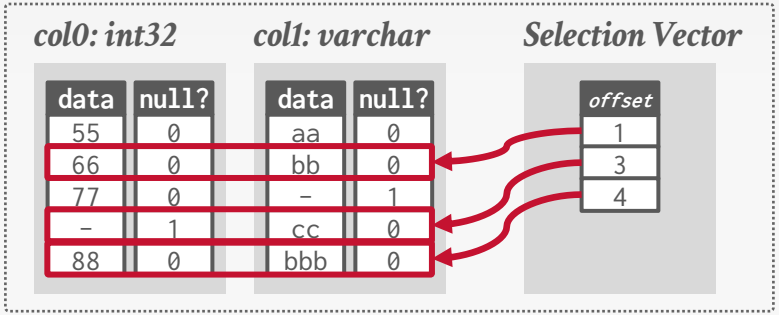
FILTER REPRESENTATION

Approach #1: Selection Vectors

→ Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.

→ Pre-allocate selection vector as the max-size of the input vector.

```
WHERE col0 IS NULL OR col1 LIKE 'b%'
```

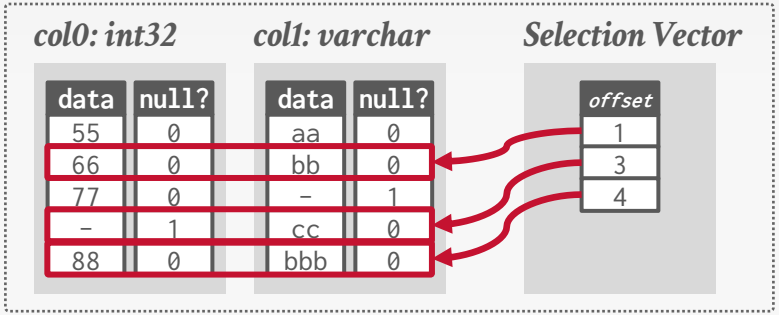


FILTER REPRESENTATION

Approach #1: Selection Vectors

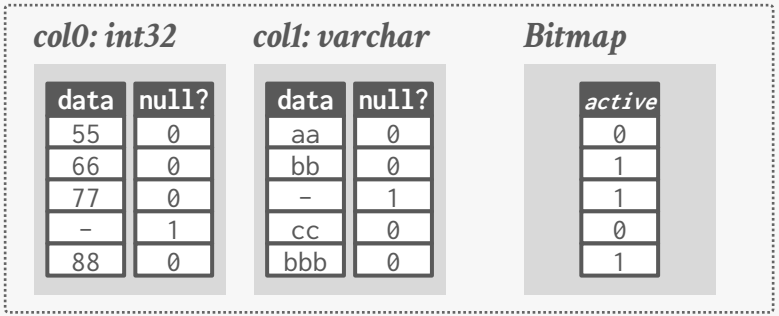
→ Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.
→ Pre-allocate selection vector as the max-size of the input vector.

```
WHERE col0 IS NULL OR col1 LIKE 'b%'
```



Approach #2: Bitmaps

→ Positionally-aligned bitmap that indicates whether a tuple is valid at an offset.
→ Some SIMD instructions natively use these bitmaps as input masks.

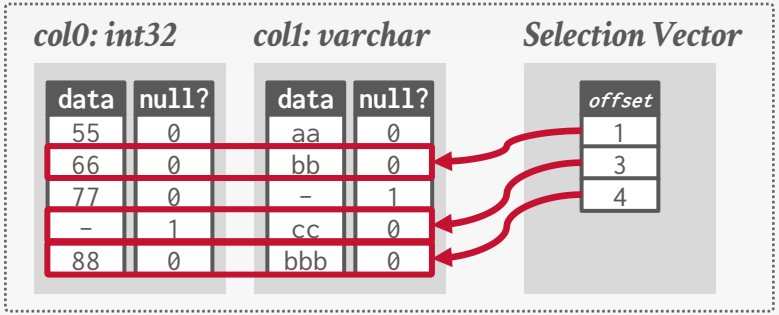


FILTER REPRESENTATION

Approach #1: Selection Vectors

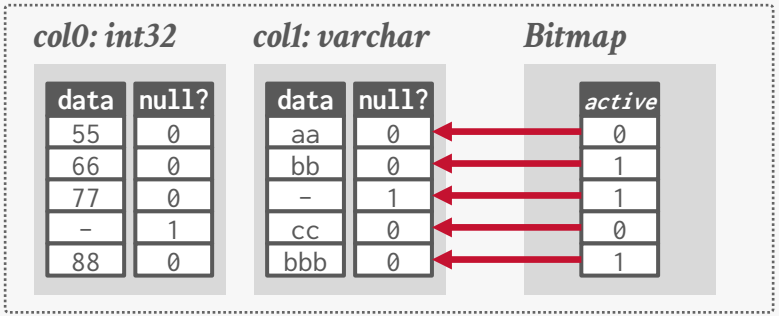
→ Dense sorted list of tuple identifiers that indicate which tuples in a batch are valid.
→ Pre-allocate selection vector as the max-size of the input vector.

```
WHERE col0 IS NULL OR col1 LIKE 'b%'
```



Approach #2: Bitmaps

→ Positionally-aligned bitmap that indicates whether a tuple is valid at an offset.
→ Some SIMD instructions natively use these bitmaps as input masks.



DATA PARALLELISM

Suppose the DBMS can parallelize some algorithm over 32 cores (one worker per core).

Assume each core has a 4-wide SIMD registers.

Potential Speed-up: 32x × 4x = 128x

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.

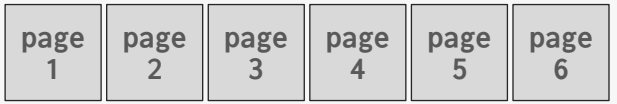
MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

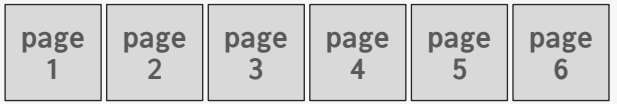
File of 6 pages (logical view):

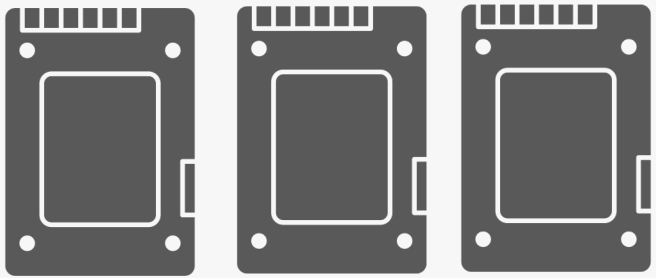


MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

File of 6 pages (logical view):



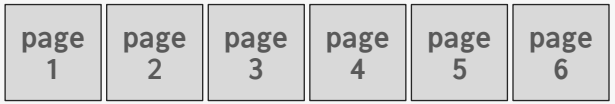


Physical layout of pages across disks

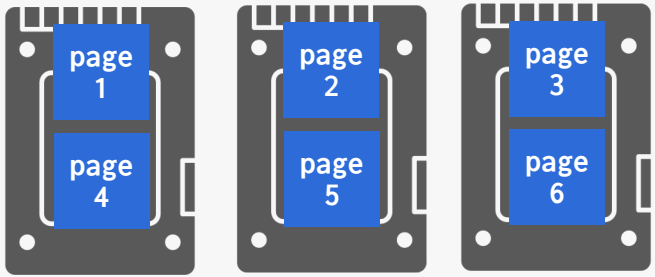
MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

File of 6 pages (logical view):



Striping (RAID 0)

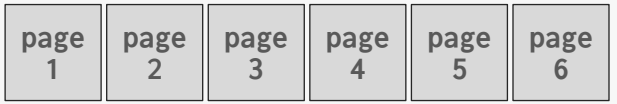


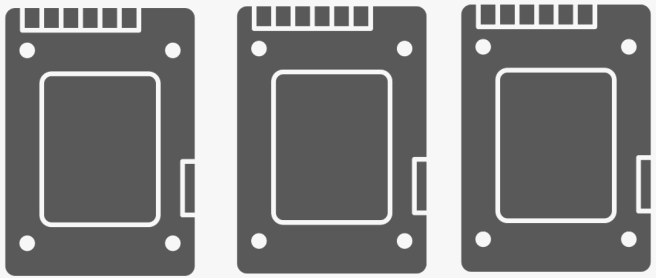
Physical layout of pages across disks

MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

File of 6 pages (logical view):



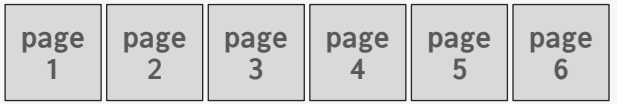


Physical layout of pages across disks

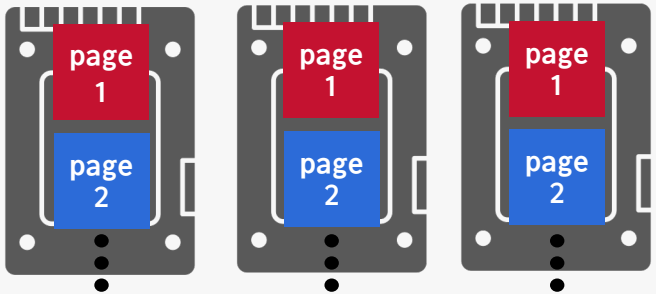
MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

File of 6 pages (logical view):



Mirroring (RAID 1)



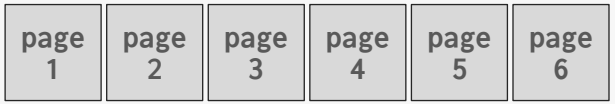
Physical layout of pages across disks

MULTI-DISK PARALLELISM

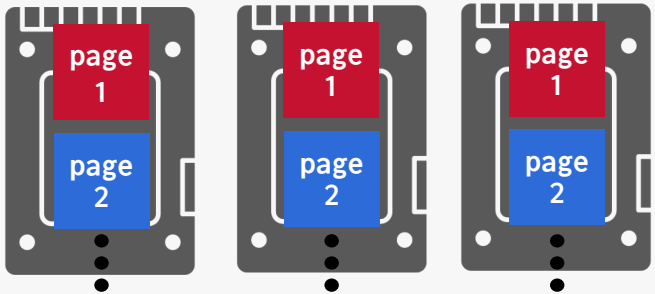
Store data across multiple disks to improve performance + durability.

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

File of 6 pages (logical view):



Mirroring (RAID 1)



Physical layout of pages across disks

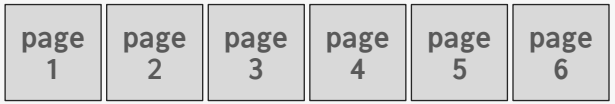
MULTI-DISK PARALLELISM

Store data across multiple disks to improve performance + durability.

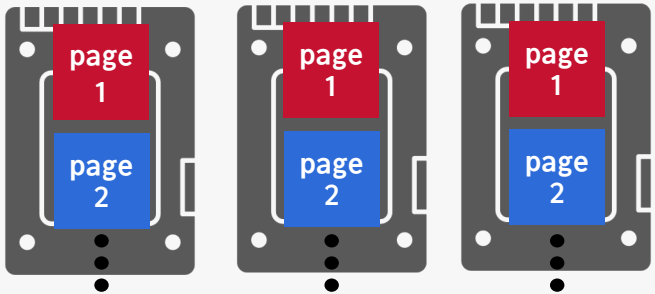
Hardware-based: I/O controller makes multiple physical devices appear as single logical device.
→ Transparent to DBMS (e.g., RAID).

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

File of 6 pages (logical view):



Mirroring (RAID 1)



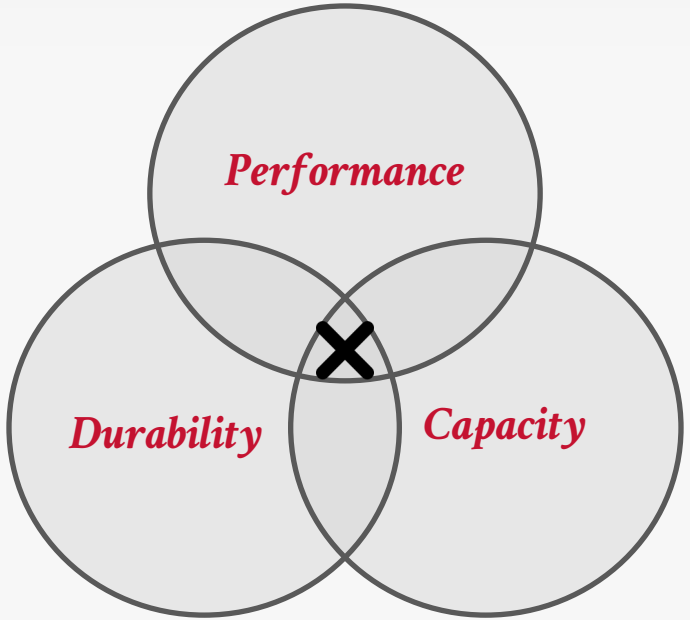
Physical layout of pages across disks

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).

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



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

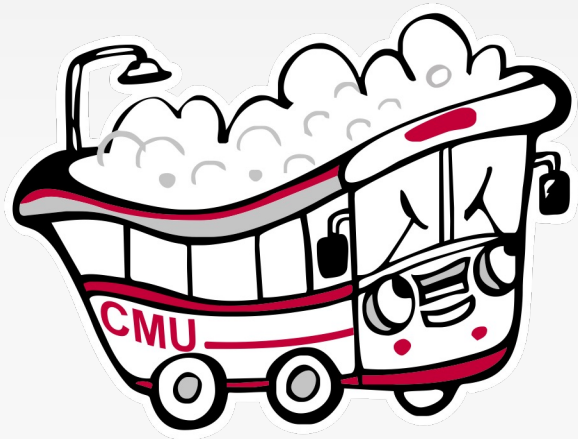
NEXT CLASS

Query Optimization
→ Logical vs Physical Plans
→ Search Space of Plans

PROJECT #3: QUERY EXECUTION

You will implement query operators and extend the optimizer to support query execution in BusTub.

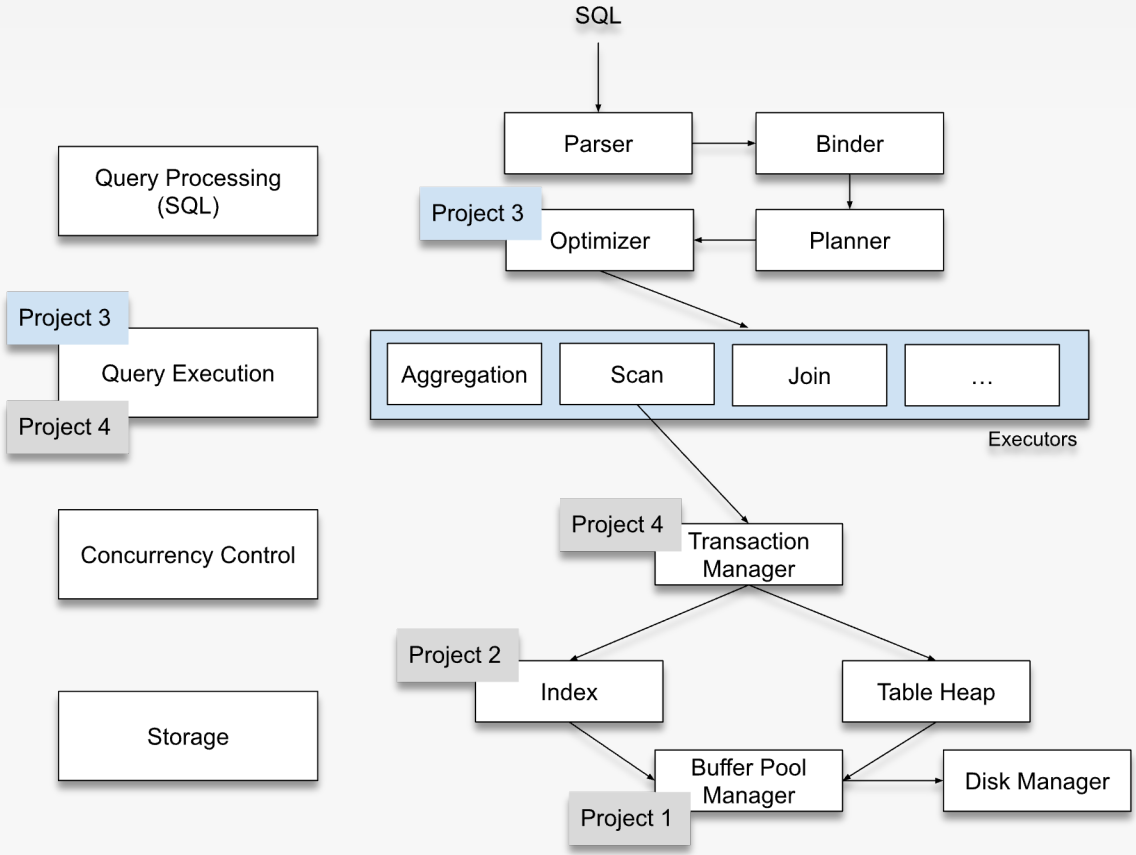
BusTub supports (basic) SQL with a rule-based optimizer for converting AST into physical plans.



Due Date:
Sunday November 15th @ 11:59pm

https://15445.courses.cs.cmu.edu/fall2026/project3

PROJECT #3: QUERY EXECUTION



PROJECT #3: TASKS

Plan Node Executors

→ Access Methods: Sequential Scan, Index Scan
→ Modifications: Insert, Delete, Update
→ Joins: Nest Loop Join, Hash Join
→ Miscellaneous: Window Aggregation, Aggregation, Limit, Sort.

Optimizer Rules:

→ Convert Nested Loops to Hash Join
→ Convert Sequential Scan to Index Scan

PROJECT #3: LEADERBOARD

The leaderboard requires you to add additional rules to the optimizer to generate query plans.

→ It will be impossible to get a top ranking by just having the fastest implementations in Project #1 + Project #2.

Tasks:

→ Column Pruning
→ More Aggressive Predicate Pushdown
→ Bloom Filter for Hash Join

DEVELOPMENT HINTS

Implement the **Insert** and **Sequential Scan** executors first so that you can populate tables and read from it.

Follow the Project Road Map rather than the order of the writeup.

You do **not** need to worry about transactions.

The aggregation hash table does **not** need to be backed by your buffer pool (i.e., use STL)

Gradescope is for meant for grading, **not** debugging. Write your own local tests.

THINGS TO NOTE

Do **not** change any file other than the ones that you submit to Gradescope.

Make sure you pull in the latest changes from the BusTub main branch.

Post your questions on Piazza or come to TA office hours.

Compare against our solution in your browser!



PLAGIARISM WARNING



The homework and projects must be your own original work. They are not group assignments.

→ You may not copy source code from other people or the web.

→ You *may* use generative AI tools for P3 and P4.

Plagiarism is not tolerated. You will get lit up.

→ Please ask instructors (not TAs!) if you are unsure.

See CMU's Policy on Academic Integrity for additional information.