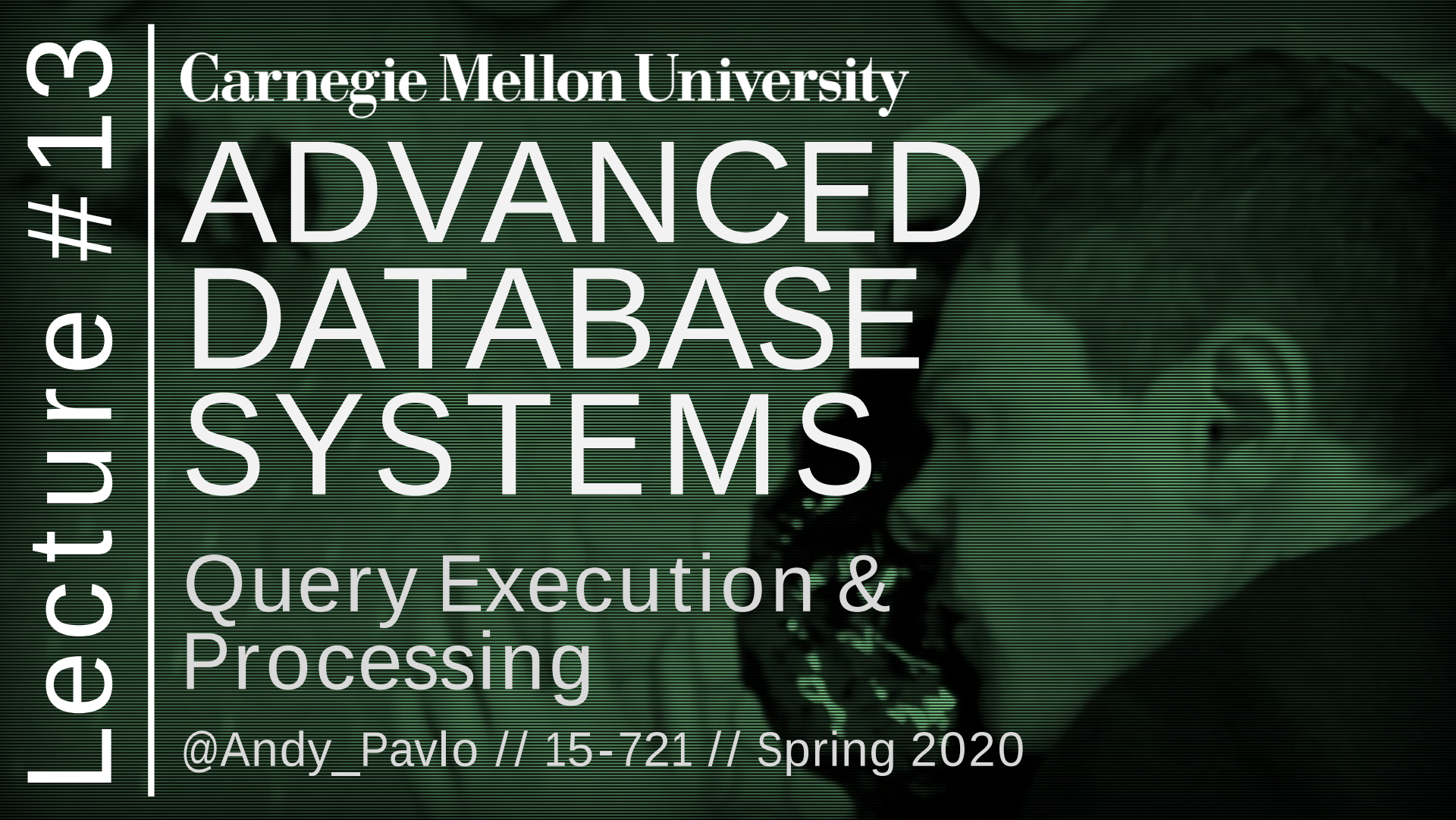


Lecture #13

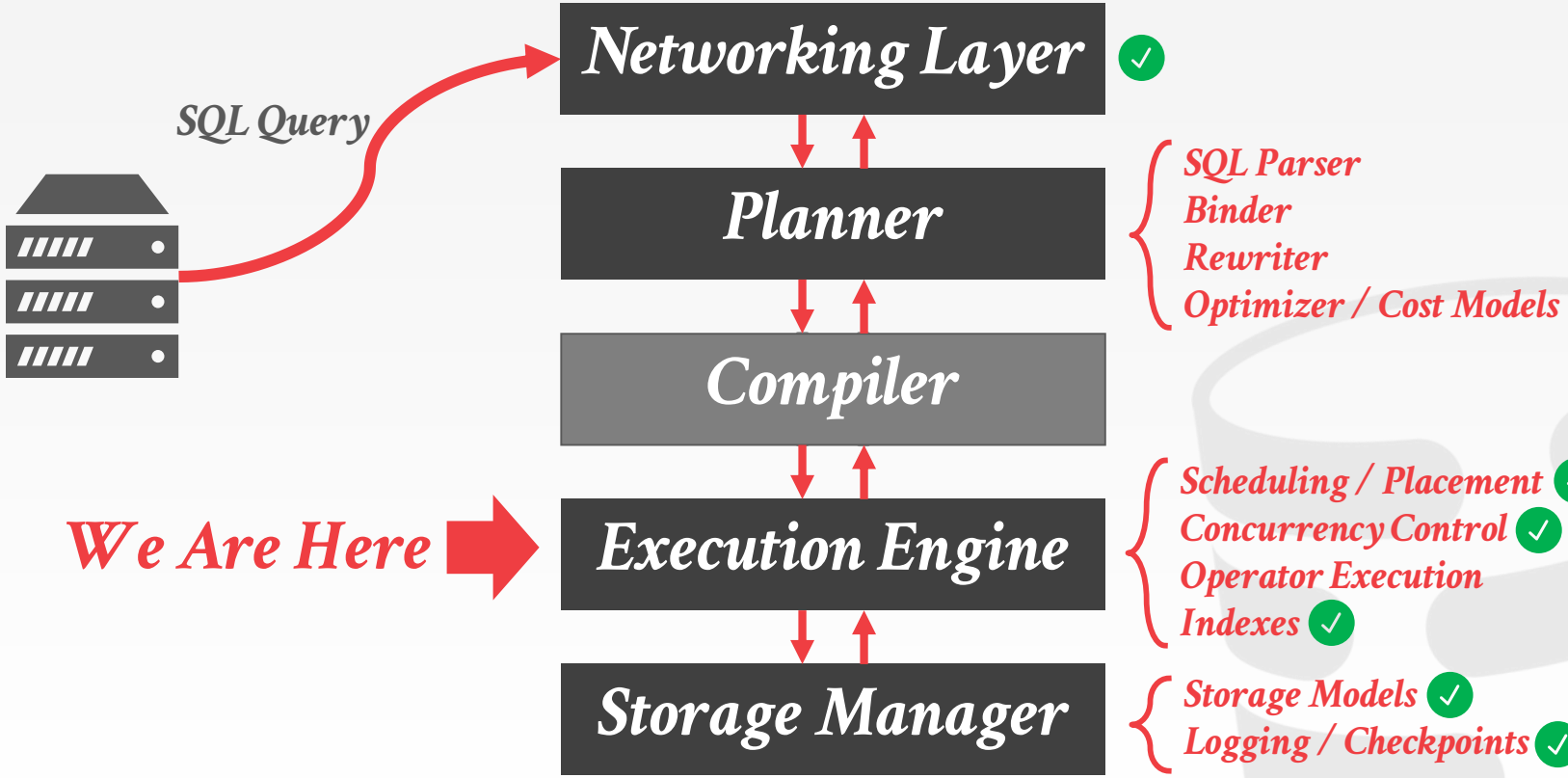
Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Query Execution & Processing

@Andy_Pavlo // 15-721 // Spring 2020

ARCHITECTURE OVERVIEW



EXECUTION OPTIMIZATION

We are now going to start discussing ways to improve the DBMS's query execution performance for data sets that fit entirely in memory.

There are other bottlenecks to target when we remove the disk.

OPTIMIZATION GOALS

Approach #1: Reduce Instruction Count
→ Use fewer instructions to do the same amount of work.

Approach #2: Reduce Cycles per Instruction
→ Execute more CPU instructions in fewer cycles.

Approach #3: Parallelize Execution
→ Use multiple threads to compute each query in parallel.

ACCESS PATH SELECTION

One major decision in query planning is whether to perform a sequential scan or index scan to retrieve data from table.

This decision depends on the selectivity of predicates as well as hardware performance and concurrency.

ACCESS PATH SELECTION IN MAIN-MEMORY OPTIMIZED DATA SYSTEMS: SHOULD I SCAN OR SHOULD I PROBE?
SIGMOD 2017

OPERATOR EXECUTION

Query Plan Processing
Scan Sharing
Materialized Views
Query Compilation
Vectorized Operators
Parallel Algorithms
Application Logic Execution (UDFs)

TODAY'S AGENDA

MonetDB/X100 Analysis
Processing Models
Parallel Execution

MONETDB/X100 (2005)

Low-level analysis of execution bottlenecks for in-memory DBMSs on OLAP workloads.

→ Show how DBMS are designed incorrectly for modern CPU architectures.

Based on these findings, they proposed a new DBMS called MonetDB/X100.

→ Renamed to Vectorwise and acquired by Actian in 2010.

→ Rebranded as Vector and Avalanche.

MONETDB/X100: HYPER-PIPELINING QUERY EXECUTION
CIDR 2005

CPU OVERVIEW

CPUs organize instructions into **pipeline stages**.

The goal is to keep all parts of the processor busy at each cycle by masking delays from instructions that cannot complete in a single cycle.

Super-scalar CPUs support multiple pipelines.

→ Execute multiple instructions in parallel in a single cycle if they are independent (**out-of-order** execution).

Everything is fast until there is a mistake…

DBMS / CPU PROBLEMS

Problem #1: Dependencies

→ If one instruction depends on another instruction, then it cannot be pushed immediately into the same pipeline.

Problem #2: Branch Prediction

→ The CPU tries to predict what branch the program will take and fill in the pipeline with its instructions.

→ If it gets it wrong, it must throw away any speculative work and flush the pipeline.

BRANCH MISPREDICTION

Because of long pipelines, CPUs will speculatively execute branches. This potentially hides the long stalls between dependent instructions.

The most executed branching code in a DBMS is the filter operation during a sequential scan.
But this is (nearly) impossible to predict correctly.

BRAN

Because of lo
execute bran
stalls betwee

The most exe
the filter ope
But this is (n

cppreference.com

Create account Search

Page Discussion View Edit History

C++ C++ language Declarations Attributes

C++ attribute: likely, unlikely (since C++20)

Allow the compiler to optimize for the case where paths of execution including that statement are more or less likely than any alternative path of execution that does not include such a statement

Syntax

`[[likely]]`	(1)
`[[unlikely]]`	(2)

Explanation

These attributes may be applied to labels and statements (other than declaration-statements). They may not be simultaneously applied to the same label or statement.

1) Applies to a statement to allow the compiler to optimize for the case where paths of execution including that statement are more likely than any alternative path of execution that does not include such a statement.
2) Applies to a statement to allow the compiler to optimize for the case where paths of execution including that statement are less likely than any alternative path of execution that does not include such a statement.

A path of execution is deemed to include a label if and only if it contains a jump to that label:

```
int f(int i) {
    switch(i) {
    case 1: [[fallthrough]];
    [[likely]] case 2: return 1;
    }
    return 2;
}
```

`i == 2` is considered more likely than any other value of `i`, but the `[[likely]]` has no effect on the `i == 1` case even though it falls through the `case 2:` label.

Example

This section is incomplete
Reason: no example

SELECTION SCANS

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

Source: Bogdan Raducanu

SELECTION SCANS

Scalar (Branching)

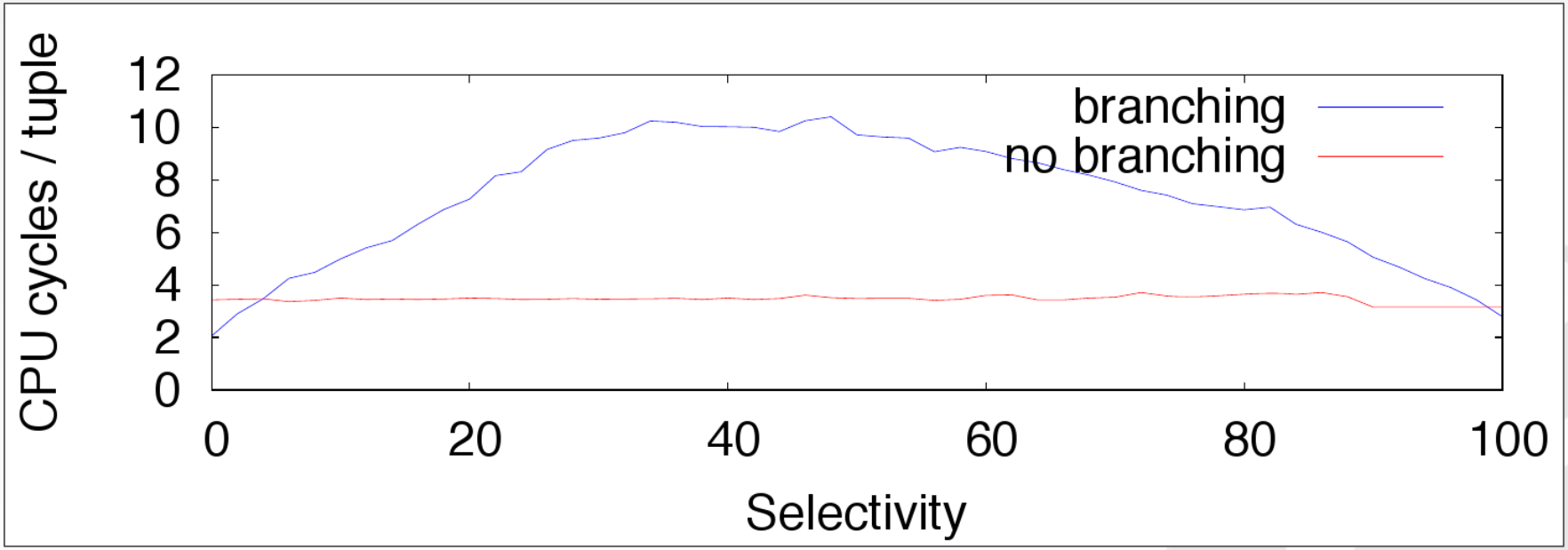
```
i = 0
for t in table:
  key = t.key
  if (key≥low) && (key≤high):
    copy(t, output[i])
    i = i + 1
```

Scalar (Branchless)

```
i = 0
for t in table:
  copy(t, output[i])
  key = t.key
  m = (key≥low ? 1 : 0) &&
    ↳(key≤high ? 1 : 0)
  i = i + m
```

Source: Bogdan Raducanu

SELECTION SCANS



Source: Bogdan Raducanu

EXCESSIVE INSTRUCTIONS

The DBMS needs to support different data types, so it must check a values type before it performs any operation on that value.

→ This is usually implemented as giant switch statements.
→ Also creates more branches that can be difficult for the CPU to predict reliably.

Example: Postgres' addition for `NUMERIC` types.

EXCESSIVE

The DBMS needs to
so it must check a v
any operation on th
→ This is usually imple
→ Also creates more b
CPU to predict relia

Example: Postgres'

```
/* ----------
 * add_var() -
 *
 *  Full version of add functionality on variable level (handling signs).
 *  result might point to one of the operands too without danger.
 * ----------
 */
int
PGTYPESnumeric_add(numeric *var1, numeric *var2, numeric *result)
{
    /*
     * Decide on the signs of the two variables what to do
     */
    if (var1->sign == NUMERIC_POS)
    {
        if (var2->sign == NUMERIC_POS)
        {
            /*
             * Both are positive result = +(ABS(var1) + ABS(var2))
             */
            if (add_abs(var1, var2, result) != 0)
                return -1;
            result->sign = NUMERIC_POS;
        }
        else
        {
            /*
             * var1 is positive, var2 is negative Must compare absolute values
             */
            switch (cmp_abs(var1, var2))
            {
                case 0:
                    /* ----------
                     * ABS(var1) == ABS(var2)
                     * result = ZERO
                     * ----------
                     */
                    zero_var(result);
                    result->rscale = Max(var1->rscale, var2->rscale);
                    result->dscale = Max(var1->dscale, var2->dscale);
                    break;

                case 1:
                    /* ----------
                     * ABS(var1) > ABS(var2)
                     * result = +(ABS(var1) - ABS(var2))
                     * ----------
                     */
                    if (sub_abs(var1, var2, result) != 0)
                        return -1;
                    result->sign = NUMERIC_POS;
                    break;

                case -1:
                    /* ----------
```

PROCESSING MODEL

A DBMS's **processing model** defines how the system executes a query plan.
→ Different trade-offs for workloads (OLTP vs. OLAP).

Approach #1: Iterator Model

Approach #2: Materialization Model

Approach #3: Vectorized / Batch Model

ITERATOR MODEL

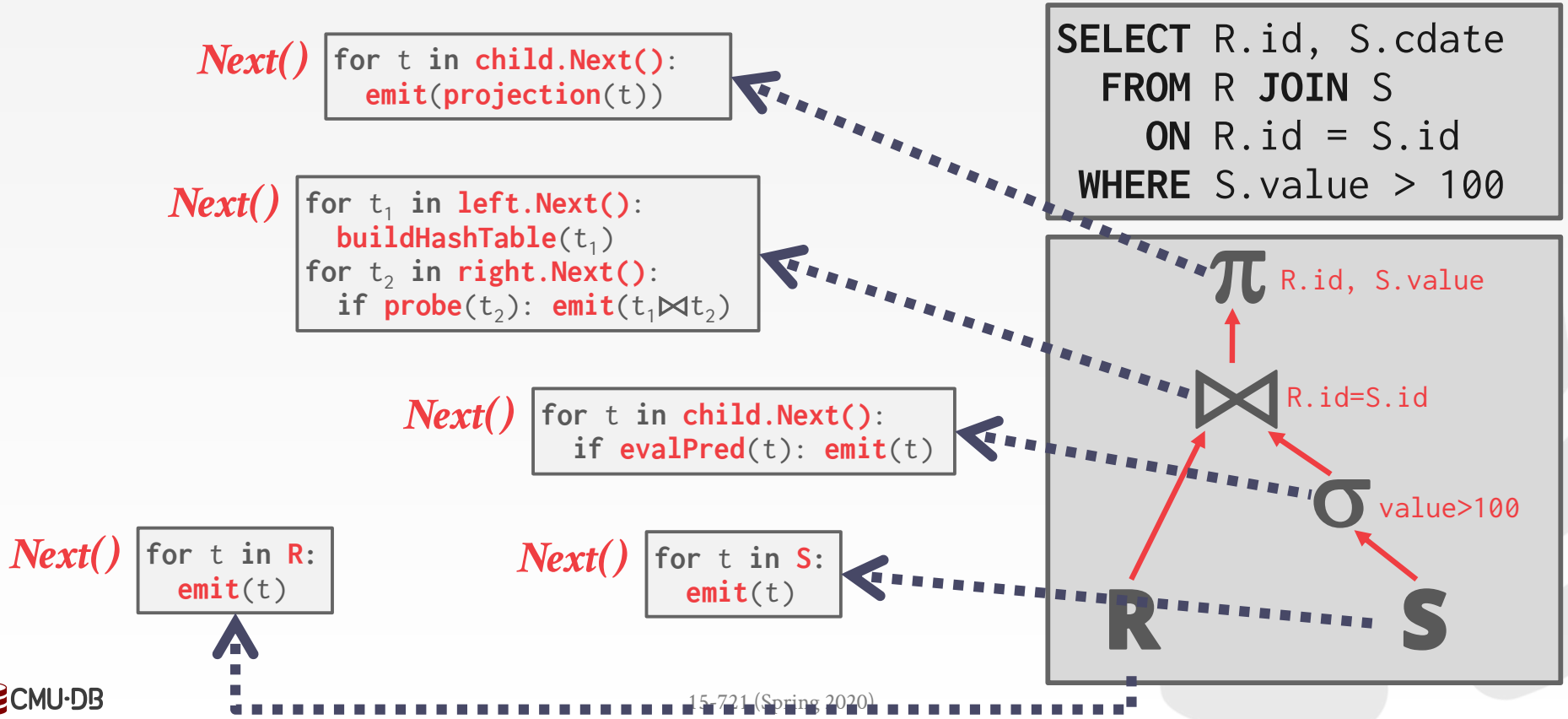
Each query plan operator implements a **next** function.

→ On each invocation, the operator returns either a single tuple or a null marker if there are no more tuples.

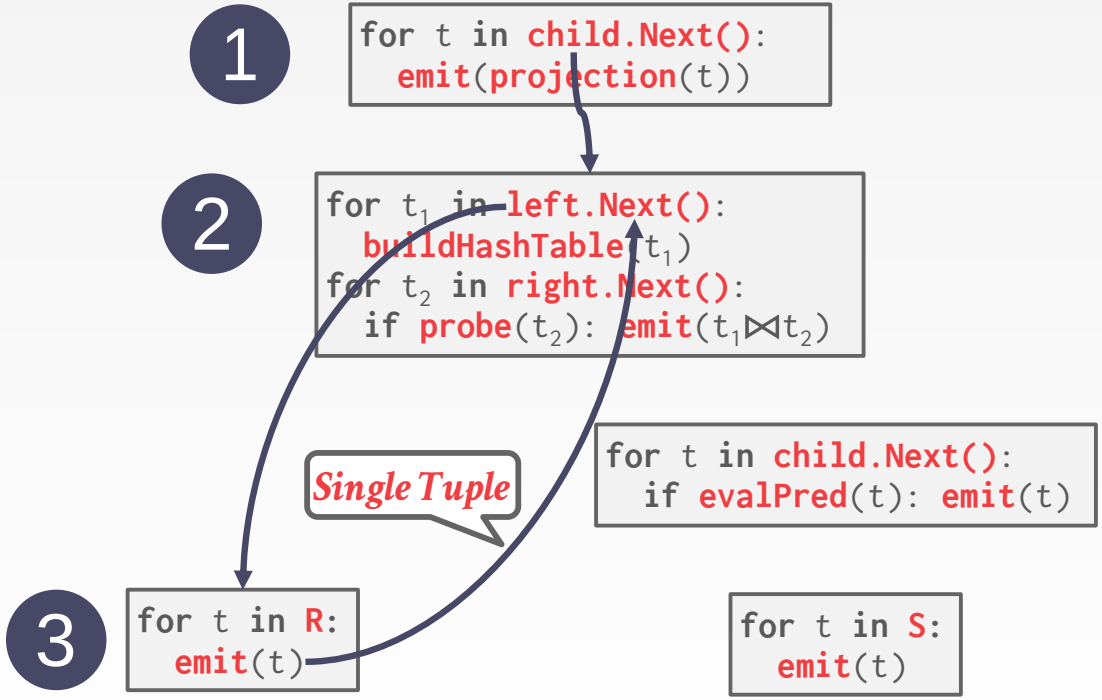
→ The operator implements a loop that calls next on its children to retrieve their tuples and then process them.

Also called **Volcano** or **Pipeline** Model.

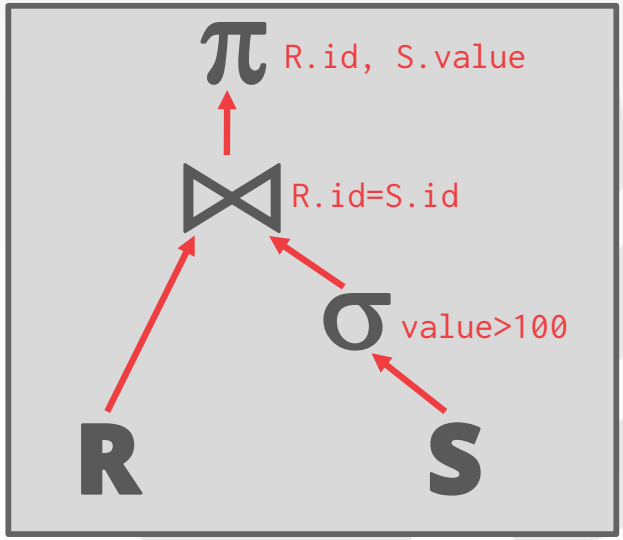
ITERATOR MODEL



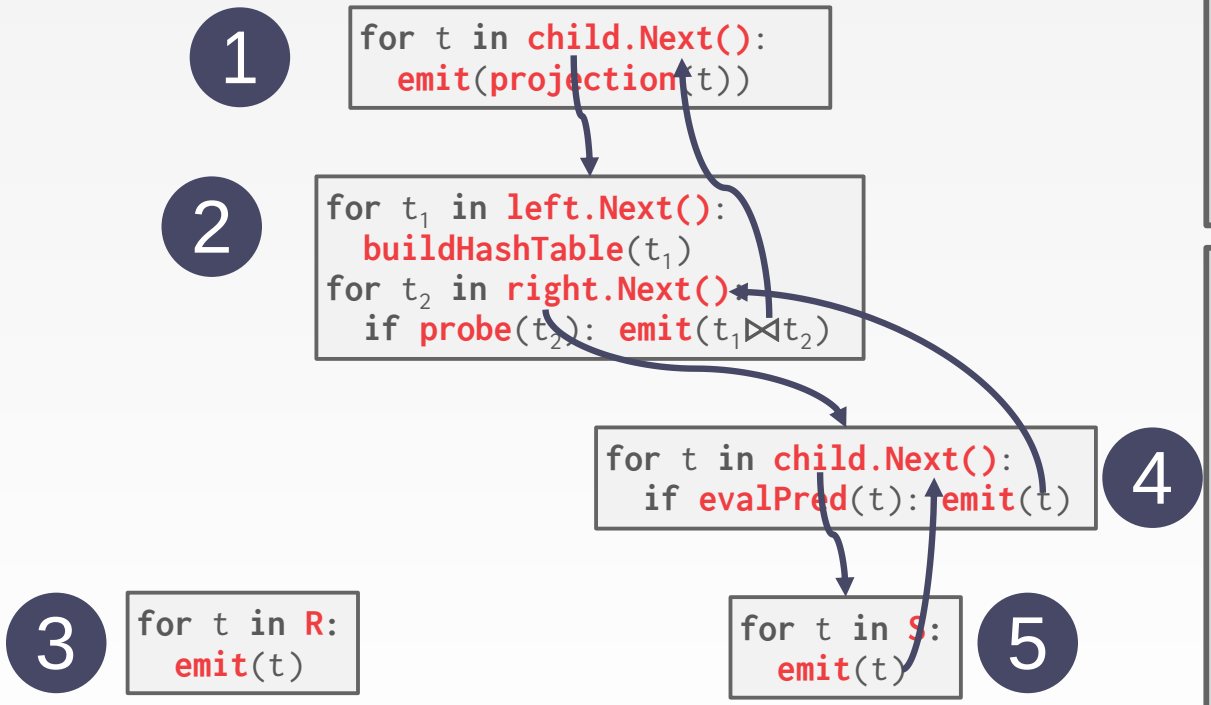
ITERATOR MODEL



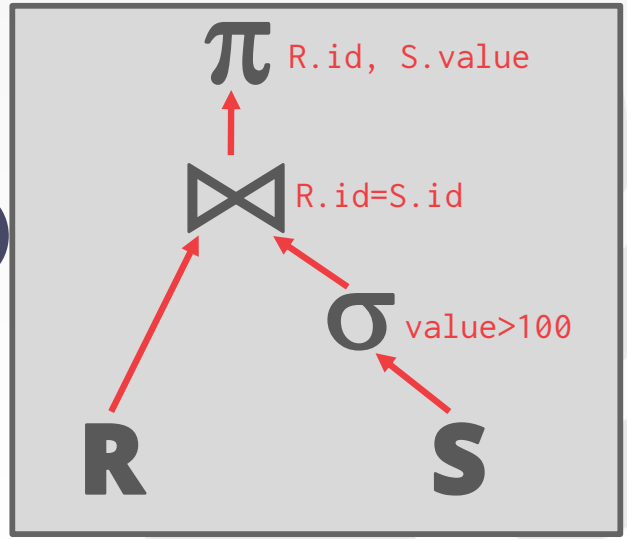

```
SELECT R.id, S.cdate
  FROM R JOIN S
    ON R.id = S.id
 WHERE S.value > 100
```



ITERATOR MODEL




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

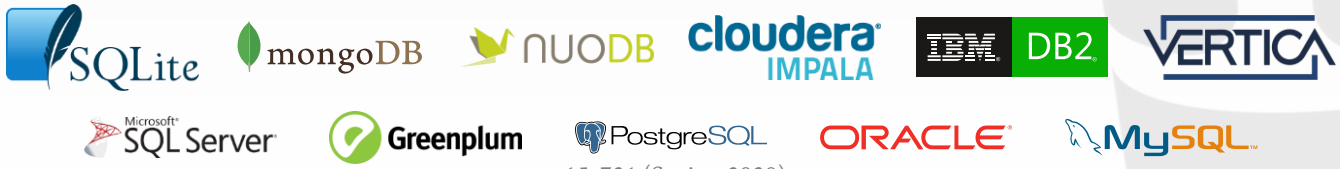


ITERATOR MODEL

This is used in almost every DBMS. Allows for tuple **pipelining**.

Some operators must block until their children emit all their tuples.
→ Joins, Subqueries, Order By

Output control works easily with this approach.



MATERIALIZATION MODEL

Each operator processes its input all at once and then emits its output all at once.

→ The operator "materializes" it output as a single result.
→ The DBMS can push down hints into to avoid scanning too many tuples.
→ Can send either a materialized row or a single column.

The output can be either whole tuples (NSM) or subsets of columns (DSM)

MATERIALIZATION MODEL

1

```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

2

```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

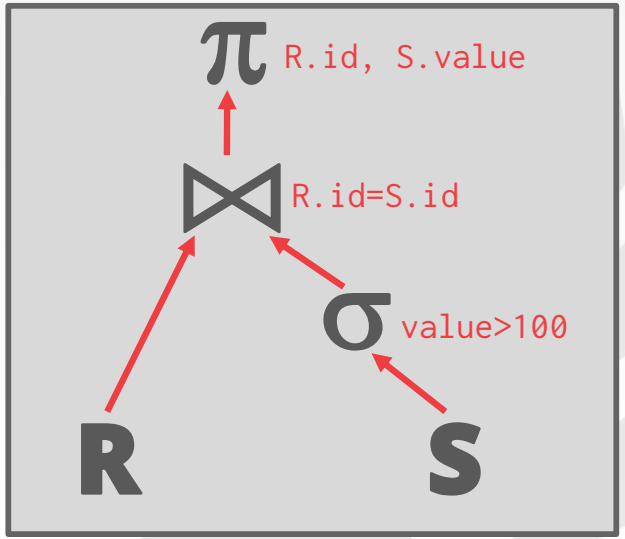
All Tuples

3

```
out = [ ]
for t in R:
  out.add(t)
return out
```

```
out = [ ]
for t in S:
  out.add(t)
return out
```

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



MATERIALIZATION MODEL

```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

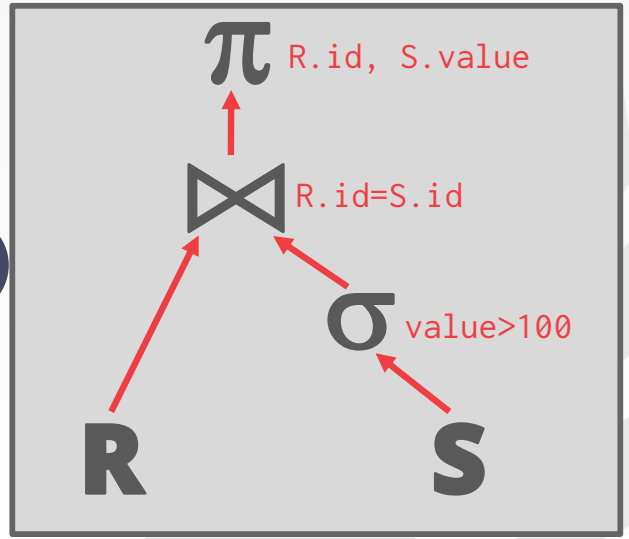
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
out = [ ]
for t in R:
  out.add(t)
return out
```

```
out = [ ]
for t in S:
  out.add(t)
return out
```

```
out = [ ]
for t in child.Output():
  if evalPred(t): out.add(t)
return out
```

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



MATERIALIZATION MODEL

1

```
out = [ ]
for t in child.Output():
  out.add(projection(t))
return out
```

2

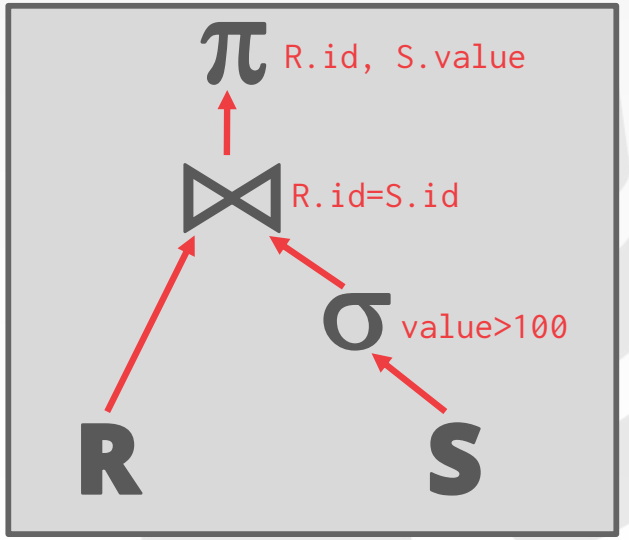
```
out = [ ]
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

```
out = [ ]
for t in S:
  if evalPred(t): out.add(t)
return out
```

3

```
out = [ ]
for t in R:
  out.add(t)
return out
```

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



MATERIALIZATION MODEL

Better for OLTP workloads because queries only access a small number of tuples at a time.
→ Lower execution / coordination overhead.
→ Fewer function calls.

Not good for OLAP queries with large intermediate results.

VECTORIZATION MODEL

Like the Iterator Model where each operator implements a **`next`** function.

But each operator emits a **batch** of tuples instead of a single tuple.

→ The operator's internal loop processes multiple tuples at a time.

→ The size of the batch can vary based on hardware or query properties.

VECTORIZATION MODEL

```
out = [ ]
for t in child.Next():
  out.add(projection(t))
  if |out|>n: emit(out)
```

1

2

```
out = [ ]
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

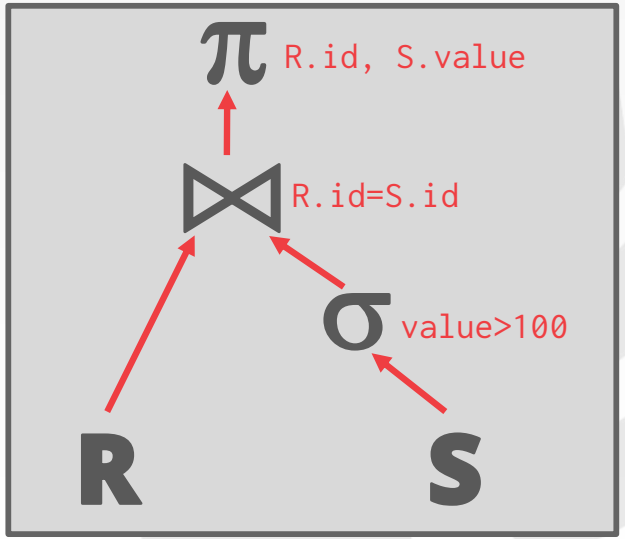
3

```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

Tuple Batch

```
out = [ ]
for t in S:
  out.add(t)
  if |out|>n: emit(out)
```

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



VECTORIZATION MODEL

```
out = [ ]
for t in child.Next():
  out.add(projection(t))
  if |out|>n: emit(out)
```

1

```
out = [ ]
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

2

```
out = [ ]
for t in R:
  out.add(t)
  if |out|>n: emit(out)
```

3

Tuple Batch

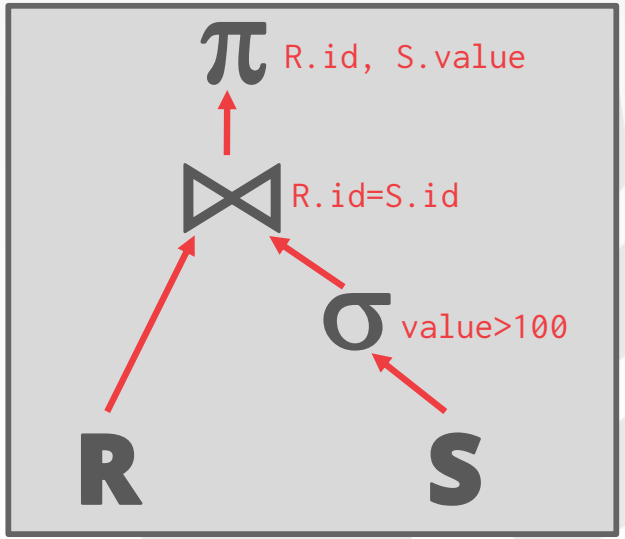
```
out = [ ]
for t in child.Next():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

4

```
out = [ ]
for t in S:
  out.add(t)
  if |out|>n: emit(out)
```

5

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



VECTORIZATION MODEL

Ideal for OLAP queries because it greatly reduces the number of invocations per operator.

Allows for operators to use vectorized (SIMD) instructions to process batches of tuples.

PLAN PROCESSING DIRECTION

Approach #1: Top-to-Bottom

→ Start with the root and "pull" data up from its children.
→ Tuples are always passed with function calls.

Approach #2: Bottom-to-Top

→ Start with leaf nodes and "push" data to their parents.
→ Allows for tighter control of caches/registers in pipelines.
→ We will see this later in HyPer and Peloton ROF.

INTER-QUERY PARALLELISM

Improve overall performance by allowing multiple queries to execute simultaneously.

→ Provide the illusion of isolation through concurrency control scheme.

The difficulty of implementing a concurrency control scheme is not significantly affected by the DBMS's process model.

INTRA-QUERY PARALLELISM

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

Approach #1: Intra-Operator (Horizontal)

Approach #2: Inter-Operator (Vertical)

These techniques are not mutually exclusive.

There are parallel algorithms for every relational operator.

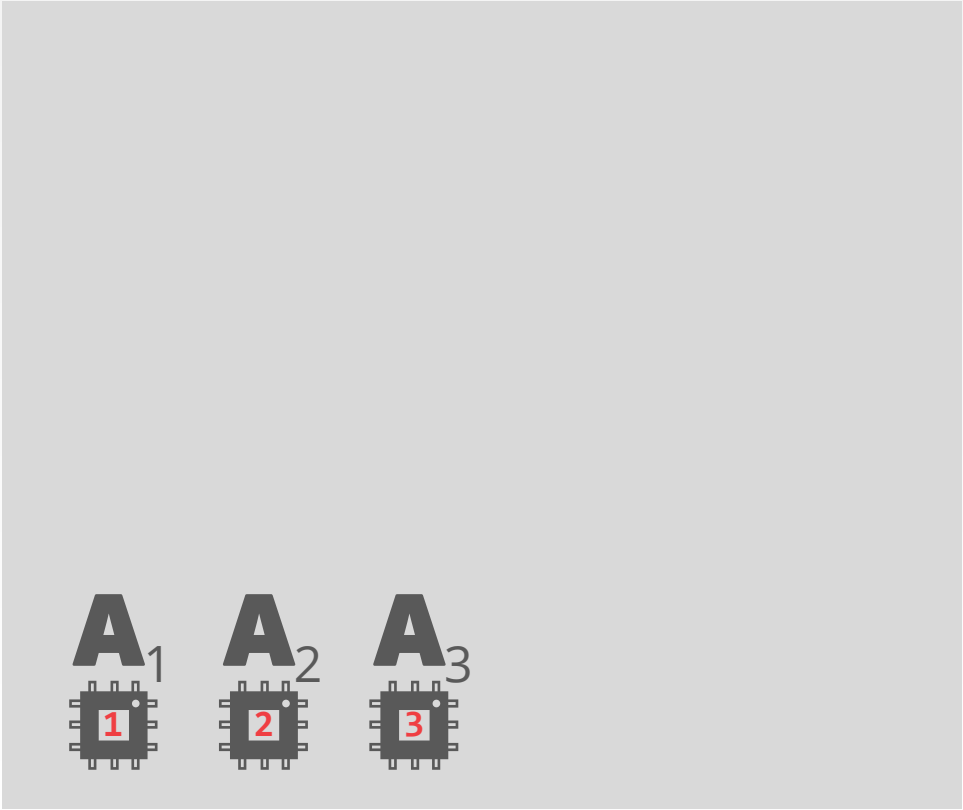
INTRA-OPERATOR PARALLELISM

Approach #1: Intra-Operator (Horizontal)

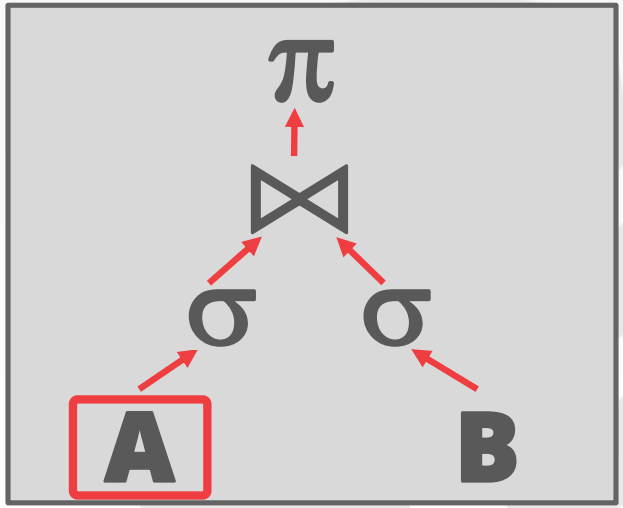
→ Operators are decomposed into independent instances that perform the same function on different subsets of data.

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

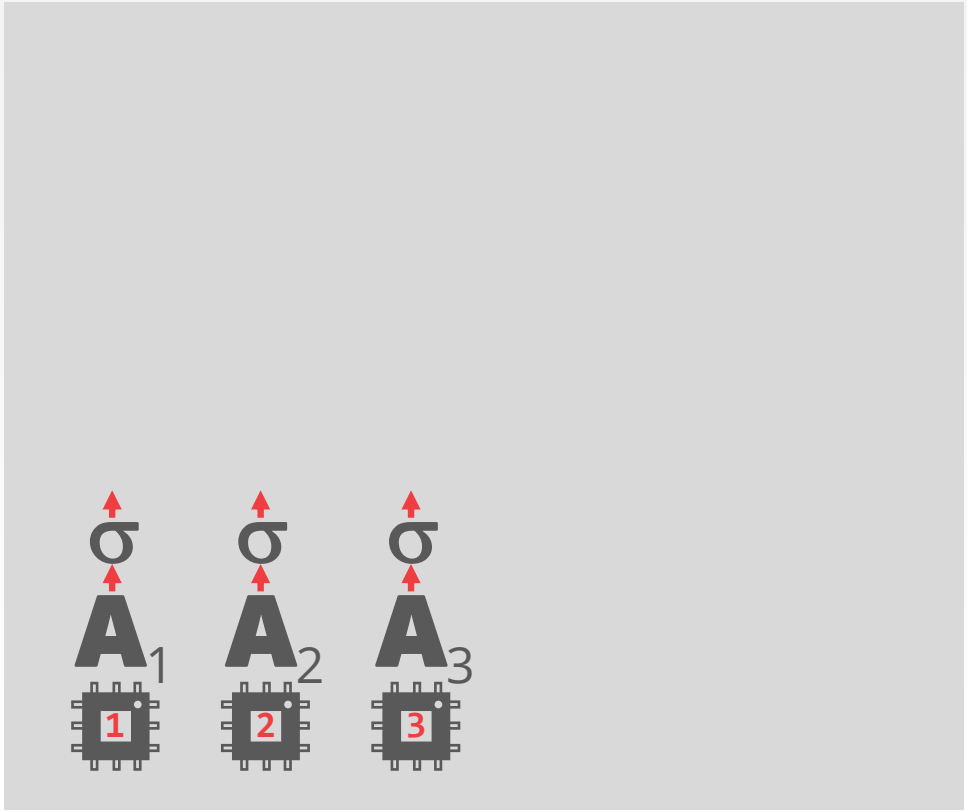
INTRA-OPERATOR PARALLELISM



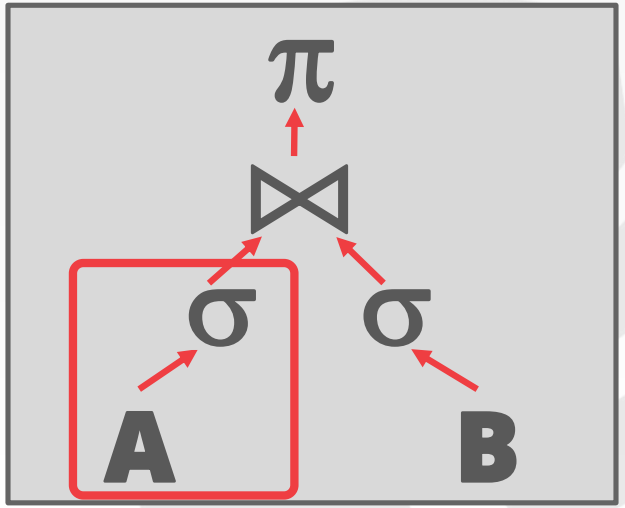

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



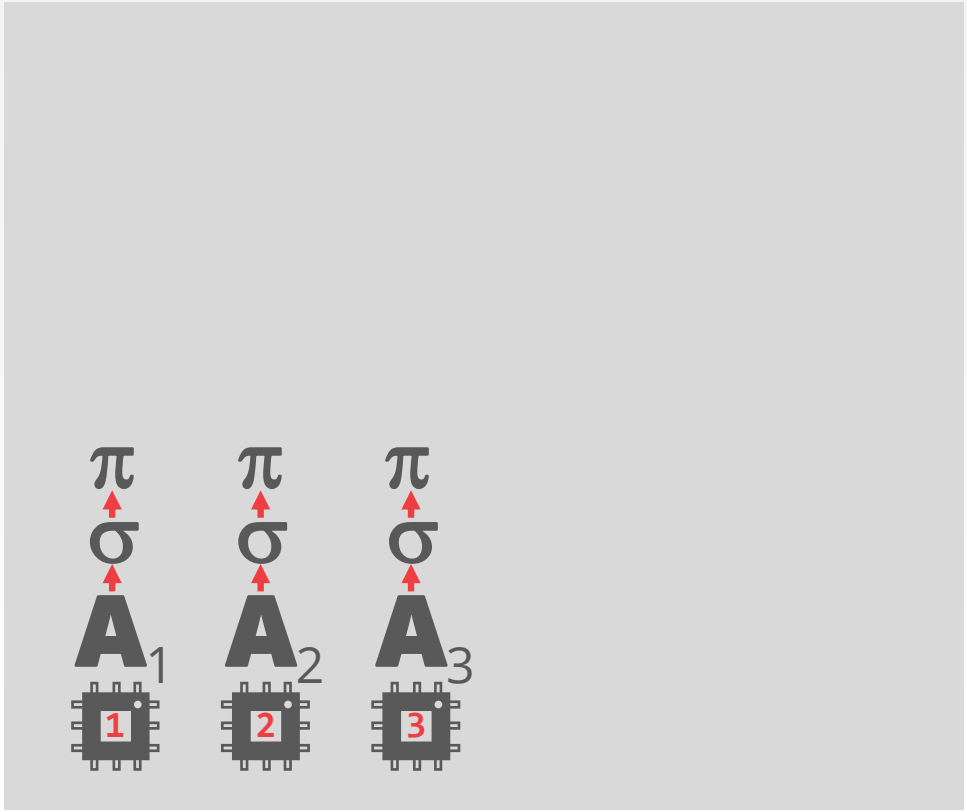
INTRA-OPERATOR PARALLELISM



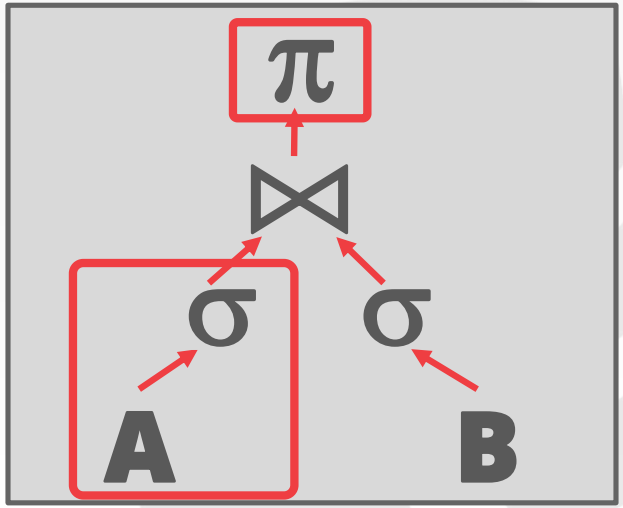

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



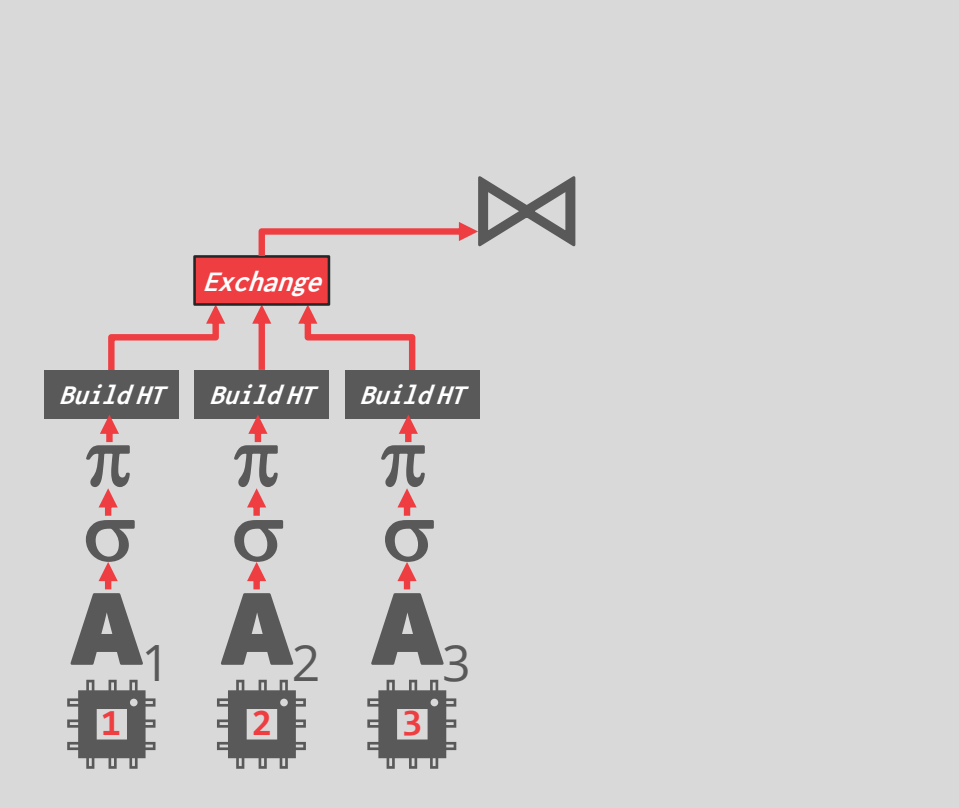
INTRA-OPERATOR PARALLELISM



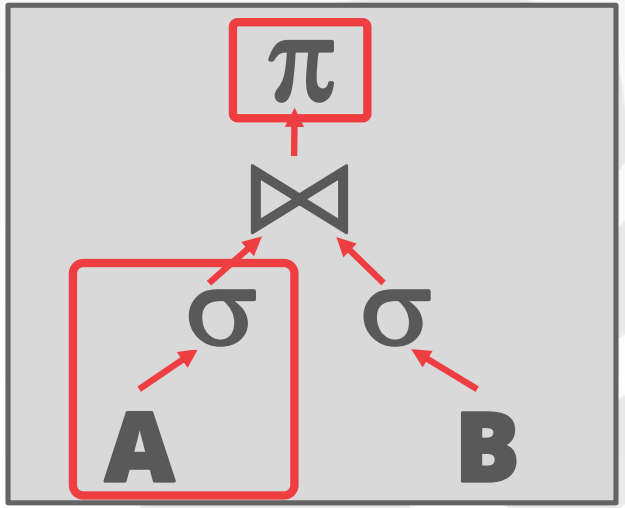

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



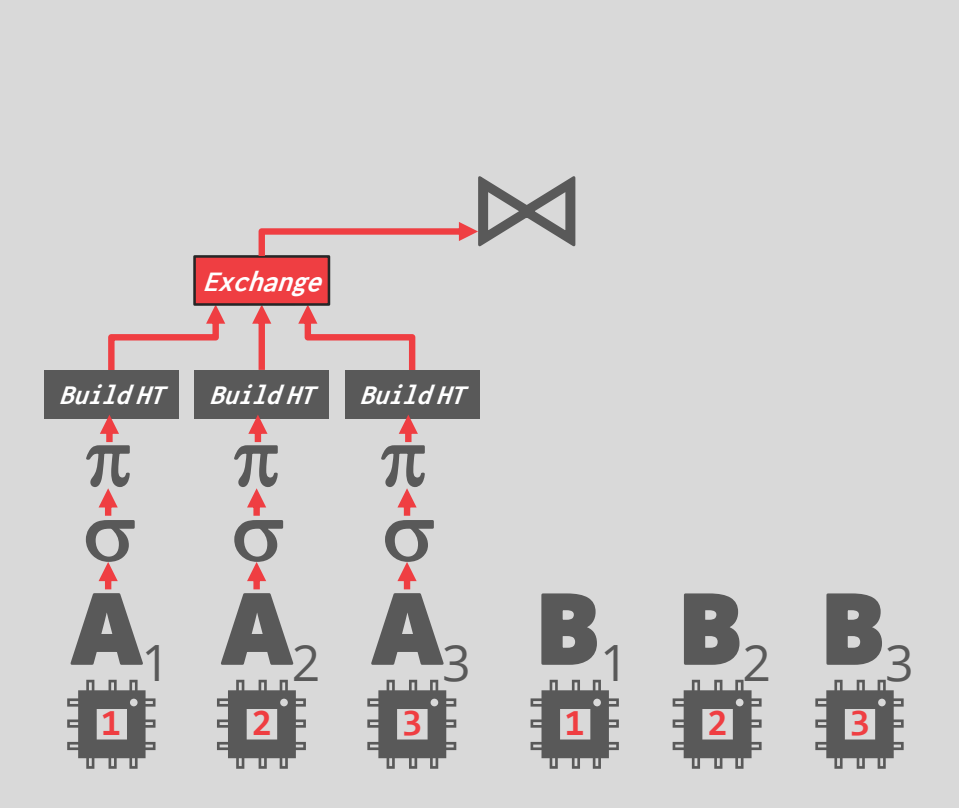
INTRA-OPERATOR PARALLELISM



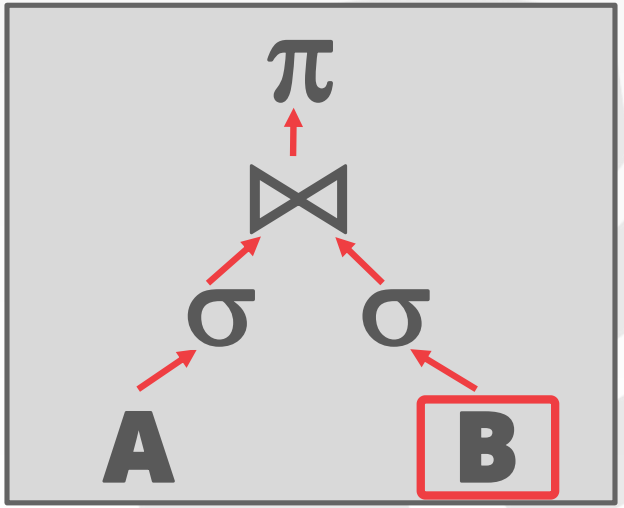

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



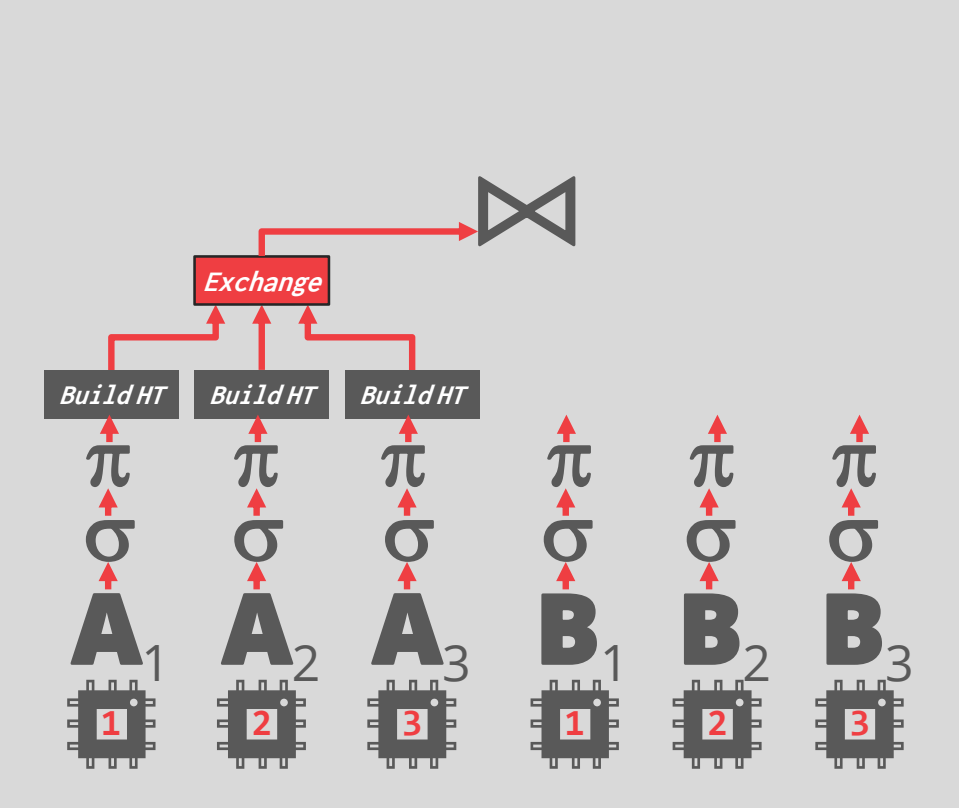
INTRA-OPERATOR PARALLELISM



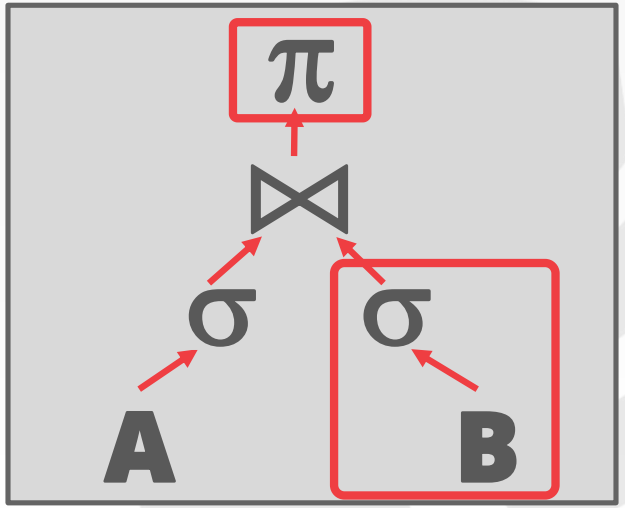

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



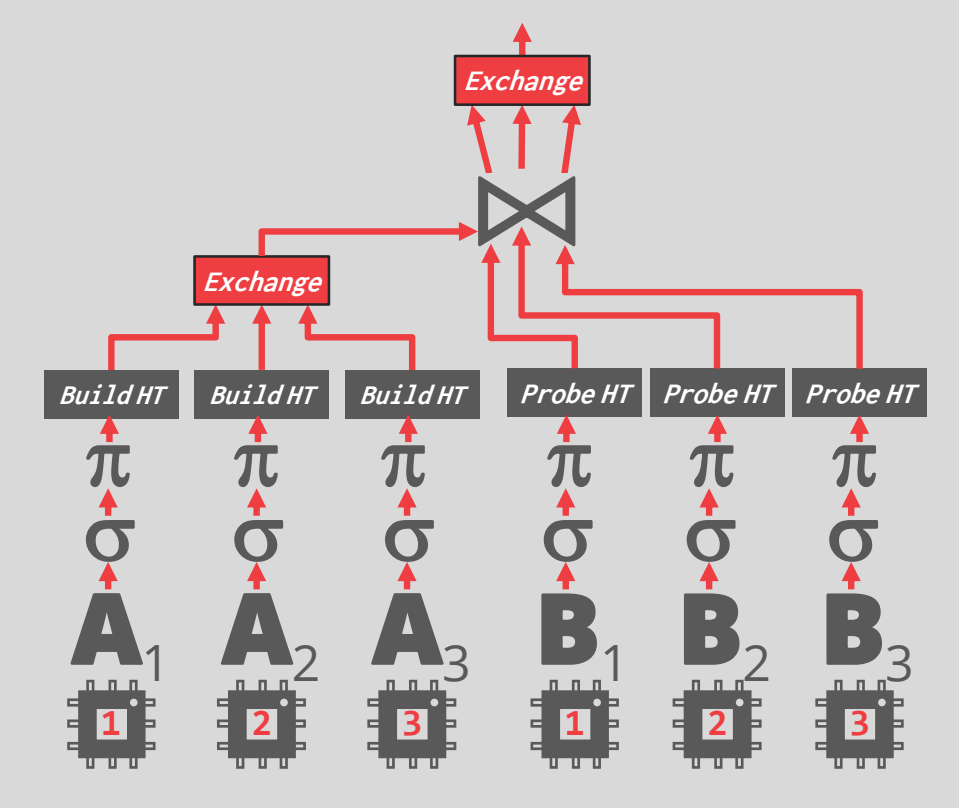
INTRA-OPERATOR PARALLELISM



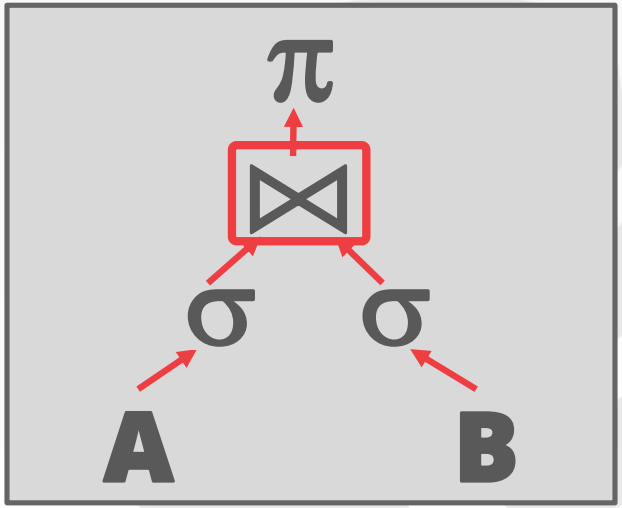

```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



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
```



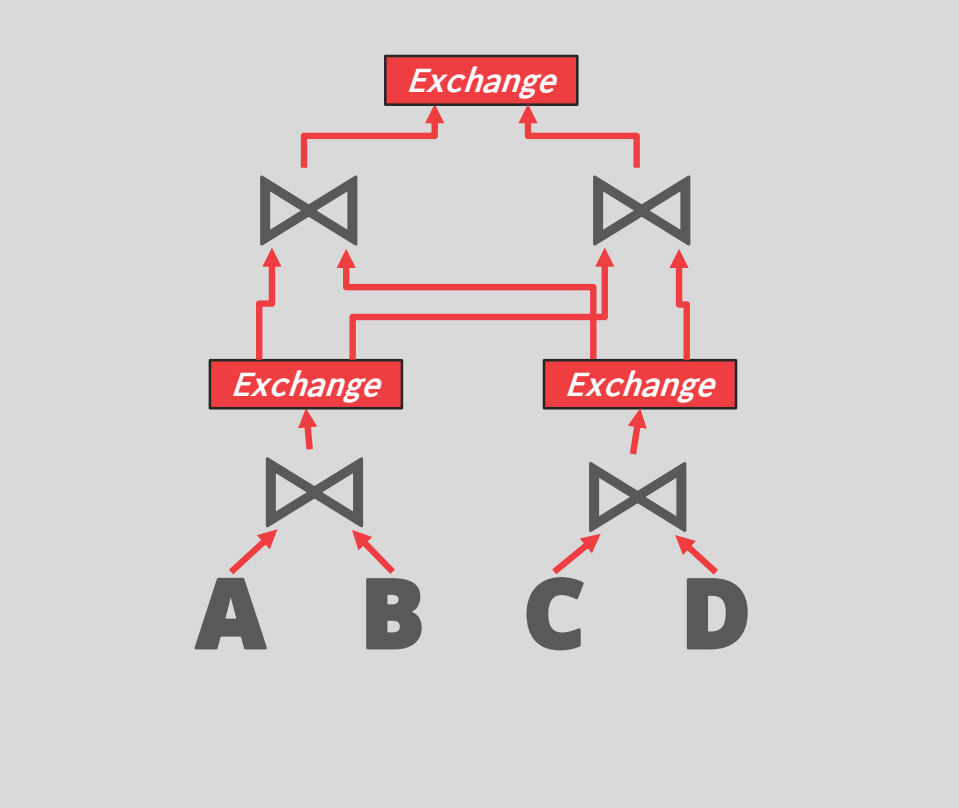
INTER-OPERATOR PARALLELISM

Approach #2: Inter-Operator (Vertical)

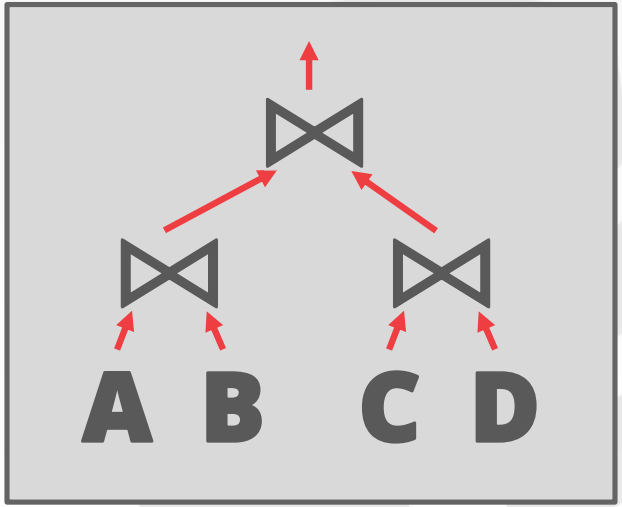
→ Operations are overlapped in order to pipeline data from one stage to the next without materialization.
→ Workers execute multiple operators from different segments of a query plan at the same time.
→ Still need exchange operators to combine intermediate results from segments.

Also called **pipelined parallelism**.

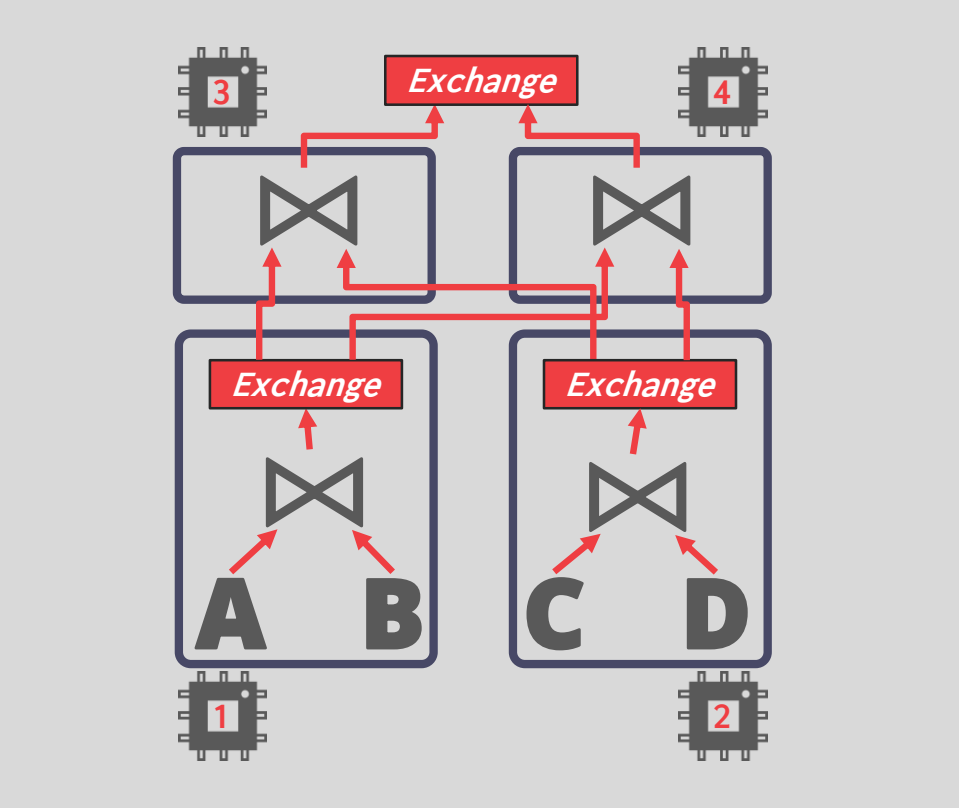
INTRA-OPERATOR PARALLELISM



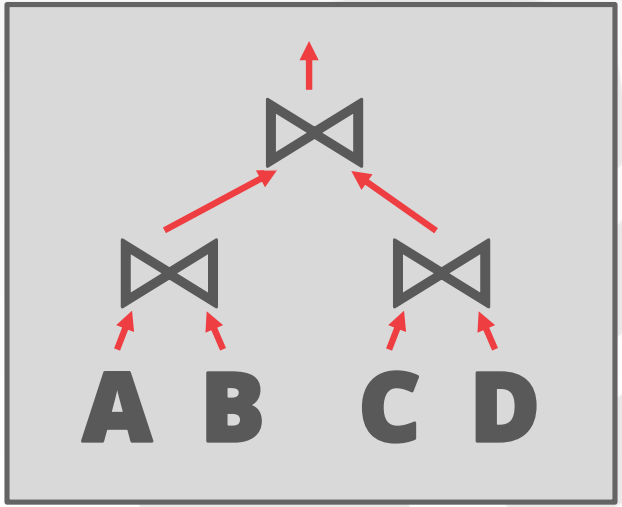

```
SELECT *
  FROM A
  JOIN B
  JOIN C
  JOIN D
```



INTRA-OPERATOR PARALLELISM




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



OBSERVATION

Determining the right number of workers to use for a query plan depends on the number of CPU cores, the size of the data, and functionality of the operators.

WORKER ALLOCATION

Approach #1: One Worker per Core

→ Each core is assigned one thread that is pinned to that core in the OS.
→ See `sched_setaffinity`

Approach #2: Multiple Workers per Core

→ Use a pool of workers per core (or per socket).
→ Allows CPU cores to be fully utilized in case one worker at a core blocks.

TASK ASSIGNMENT

Approach #1: Push

→ A centralized dispatcher assigns tasks to workers and monitors their progress.

→ When the worker notifies the dispatcher that it is finished, it is given a new task.

Approach #1: Pull

→ Workers pull the next task from a queue, process it, and then return to get the next task.

PARTING THOUGHTS

The easiest way to implement something is not going to always produce the most efficient execution strategy for modern CPUs.

We will see that vectorized / bottom-up execution will be the better way to execute OLAP queries.

NEXT CLASS

Query Compilation