Lecture #15

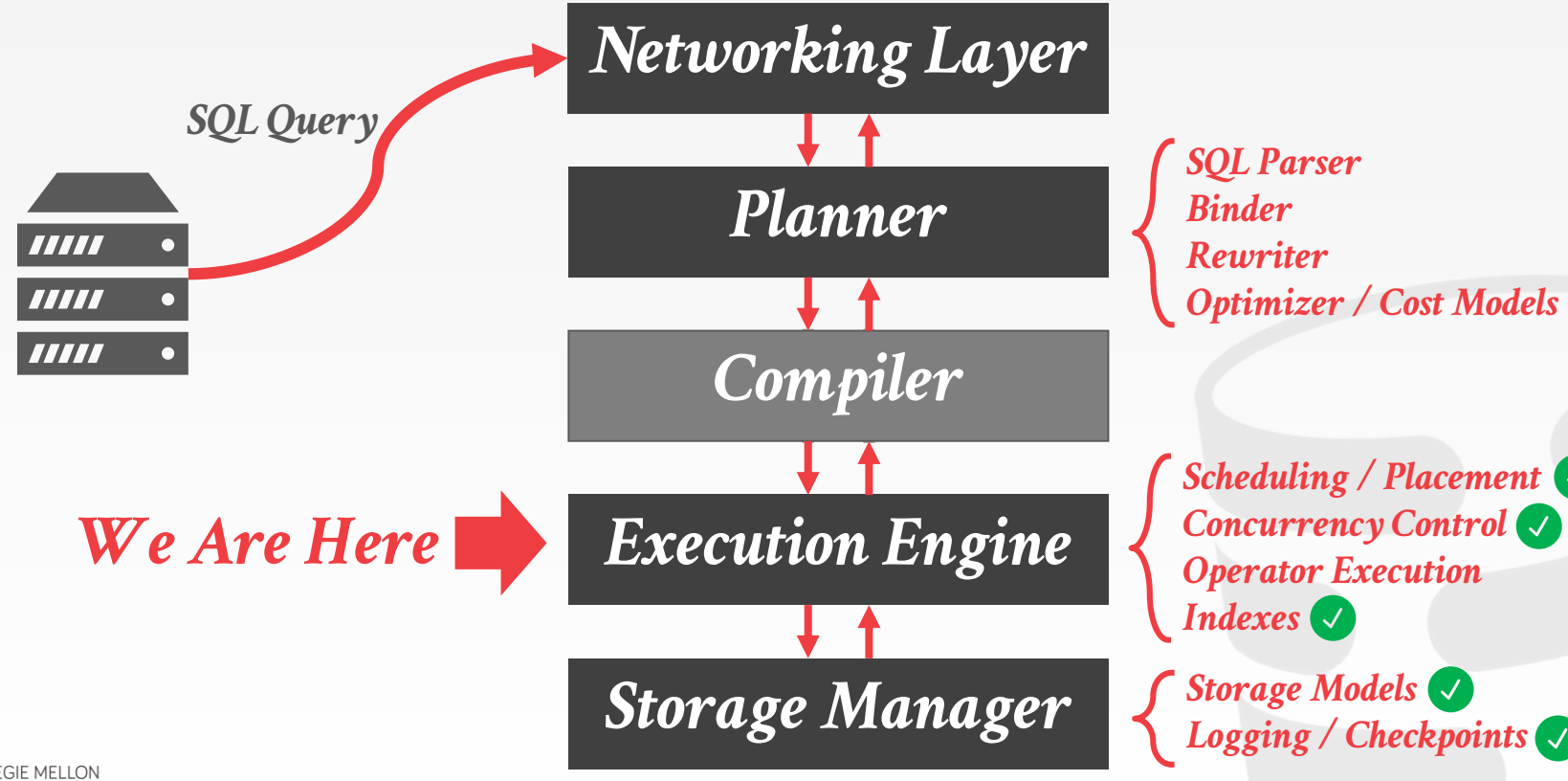
Carnegie Mellon University

ADVANCED DATABASE SYSTEMS

Query Execution & Processing

@Andy_Pavlo // 15-721 // Spring 2019

ARCHITECTURE OVERVIEW



OPERATOR EXECUTION

Query Plan Processing
Application Logic Execution (UDFs)
Parallel Join Algorithms
Vectorized Operators
Query Compilation

QUERY EXECUTION

A query plan is comprised of **operators**.

An **operator instance** is an invocation of an operator on some segment of data.

A **task** is the execution of a sequence of one or more operator instances.

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.
→ This means reducing cache misses and stalls due to memory load/stores.

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

TODAY'S AGENDA

MonetDB/X100 Analysis
Processing Models
Parallel Execution

MONETDB/X100

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.

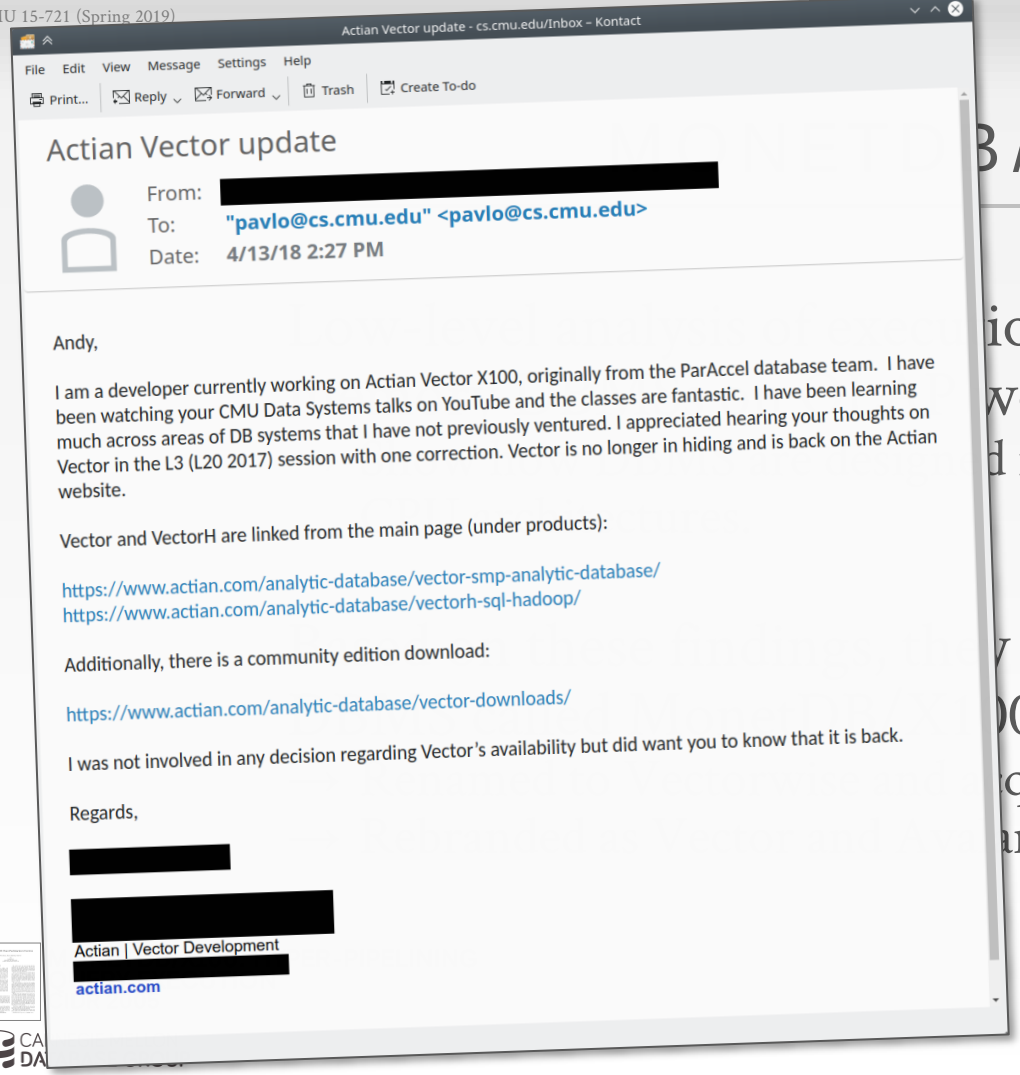
→ Rebranded as Vector and Avalance

MONETDB/X100: HYPER-PIPELINING QUERY EXECUTION
CIDR 2005

3/X100

ion bottlenecks for in-
workloads.
d incorrectly for modern

y proposed a new
00.
quired by Actian in 2010.
ance


Actian Vector update - cs.cmu.edu/Inbox – Kontact

File Edit View Message Settings Help

Print... Reply Forward Trash Create To-do

Actian Vector update

From: [redacted]
To: "pavlo@cs.cmu.edu" <pavlo@cs.cmu.edu>
Date: 4/13/18 2:27 PM

Andy,

I am a developer currently working on Actian Vector X100, originally from the ParAccel database team. I have been watching your CMU Data Systems talks on YouTube and the classes are fantastic. I have been learning much across areas of DB systems that I have not previously ventured. I appreciated hearing your thoughts on Vector in the L3 (L20 2017) session with one correction. Vector is no longer in hiding and is back on the Actian website.

Vector and VectorH are linked from the main page (under products):

https://www.actian.com/analytic-database/vector-smp-analytic-database/
https://www.actian.com/analytic-database/vectorh-sql-hadoop/

Additionally, there is a community edition download:

https://www.actian.com/analytic-database/vector-downloads/

I was not involved in any decision regarding Vector's availability but did want you to know that it is back.

Regards,

[redacted]

[redacted]
Actian | Vector Development
[redacted]
actian.com

Actian Vector update - cs.cmu.edu/Inbox – Kontact

File Edit View Message Settings Help

Print... Reply Forward Trash Create To-do

Actian Vector update

From: [redacted]
To: "pavlo@cs.cmu.edu" <pavlo@cs.cmu.edu>
Date: 4/13/18 2:27 PM

Andy,

I am a developer currently working on Actian Vector X100, originally from the Par
been watching your CMU Data Systems talks on YouTube and the classes are fant
much across areas of DB systems that I have not previously ventured. I appreciat
Vector in the L3 (L20 2017) session with one correction. Vector is no longer in hi
website.

Vector and VectorH are linked from the main page (under products):

https://www.actian.com/analytic-database/vector-smp-analytic-database/
https://www.actian.com/analytic-database/vectorh-sql-hadoop/

Additionally, there is a community edition download:

https://www.actian.com/analytic-database/vector-downloads/

I was not involved in any decision regarding Vector's availability but did want

Regards,

[redacted]

[redacted]
Actian | Vector Development
[redacted]
actian.com

BusinessWire
A Berkshire Hathaway Company

HOME SERVICES NEWS EDUCATION ABOUT US

Search

Log In Sign Up

Actian Avalanche Pioneers Next-Generation Cloud Data Warehouse

New fully managed, third-generation, cloud data warehouse delivers industry-leading performance, simplicity, scalability and savings

March 18, 2019 09:00 AM Eastern Daylight Time

PALO ALTO, Calif.--(BUSINESS WIRE)--Actian, the hybrid data management, analytics and integration company, today announced the Actian Avalanche™ cloud data warehouse, a fully managed service solution available on the AWS cloud platform. This innovative, third-generation, cloud data warehouse delivers breakthrough levels of speed, flexibility and economics designed to make it easier for companies to deploy and scale data analytics services in the cloud while saving millions over traditional data warehouse solutions.

"There is a major opportunity to deliver transformative technology to power an entirely new generation of cloud data warehouses to replace traditional solutions that are fundamentally tapped out and alleviate the challenges of new demanding BI, advanced analytics and machine learning workloads."

Tweet this

As enterprises continue along their digital transformation journey, it's imperative that they derive more value from their diverse, hybrid data to improve profitability and remain competitive in their industries. However, traditional data analytics solutions have substantial limitations when it comes to complexity, cost and performance. Similarly, first-generation cloud solutions have also experienced challenges in areas of concurrency, performance at scale and resource optimization. Many organizations, while embracing the cloud, still critically need to address their hybrid data legacy on-premise as well as compliance requirements.

"Enterprises are being driven to get more value out of their data by enabling faster, more actionable insights that drive better business outcomes. This trend is fueling a market need for a new generation of operational data warehouses that are simple to use and deliver real-time insights at scale, without the traditional data lake performance penalties," said Jim Curtis, Senior Analyst for the Data Platforms and Analytics Channel at 451 Research. "Actian has recognized this need by concentrating on price/performance benefits with its next generation of cloud data warehouse and on-premise analytics offerings."

Gen III Cloud Data Warehouse Solution

actian

ACTIAN CORPORATION

More News

Contacts

Actian Corporation
Jeff Veis
SVP & Chief Marketing Officer
jeff.veis@actian.com

PAN Communications
Caitlin Noll
Account Supervisor
Actian@pancomm.com

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.

→ Flynn's Taxonomy: Single Instruction stream, Single Data stream (**SISD**)

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 has to 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.

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
```


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
```

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

Scalar (Branching)

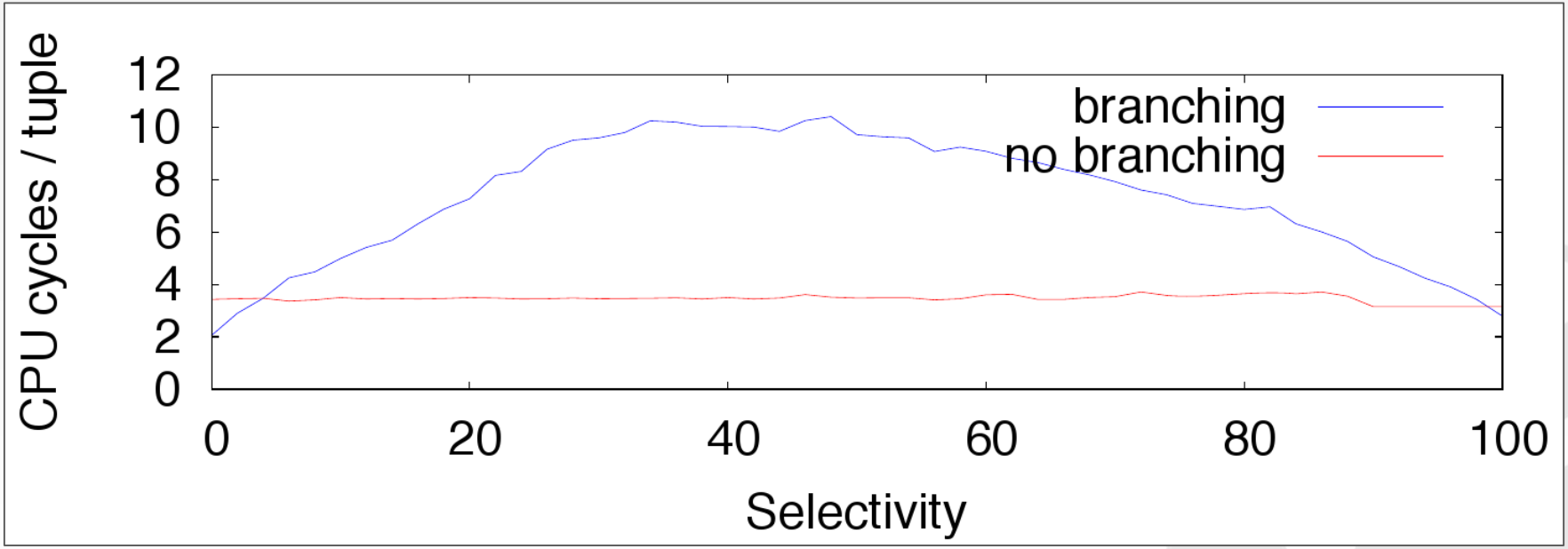
```
i = 0
for t in table:
  key = t.key
  if (key≥low) && (key≤high):
    copy(t, output[i])
    i = i + 1
```

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;
```

PROCESSING MODEL

A DBMS's **processing model** defines how the system executes a query plan.
→ Different trade-offs for different workloads.

Approach #1: Iterator Model

Approach #2: Materialization Model

Approach #3: Vectorized / Batch Model

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

```
for t in child.Next():
  emit(projection(t))
```

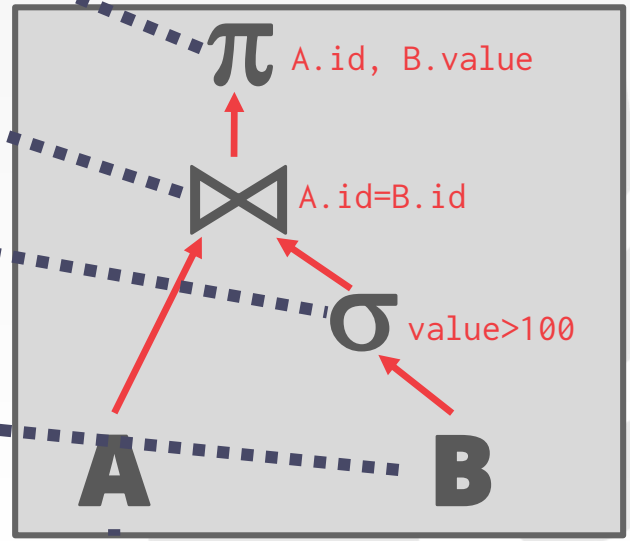
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
for t in child.Next():
  if evalPred(t): emit(t)
```

```
for t in A:
  emit(t)
```

```
for t in B:
  emit(t)
```

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



ITERATOR MODEL

1

```
for t in child.Next():
  emit(projection(t))
```

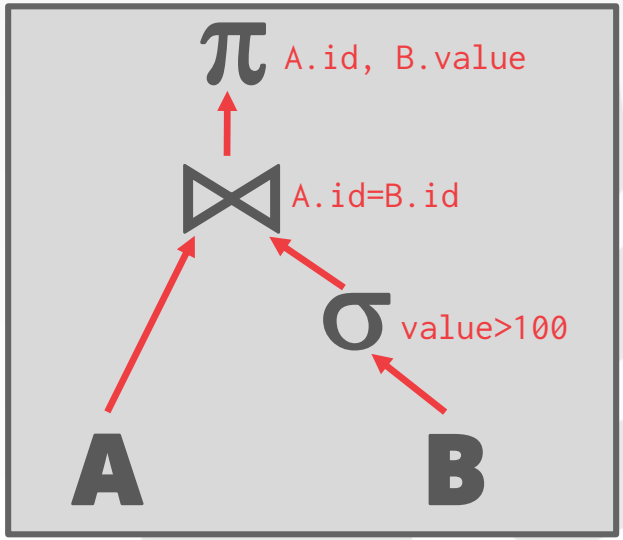
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
for t in child.Next():
  if evalPred(t): emit(t)
```

```
for t in A:
  emit(t)
```

```
for t in B:
  emit(t)
```

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



ITERATOR MODEL

1

```
for t in child.Next():
  emit(projection(t))
```

2

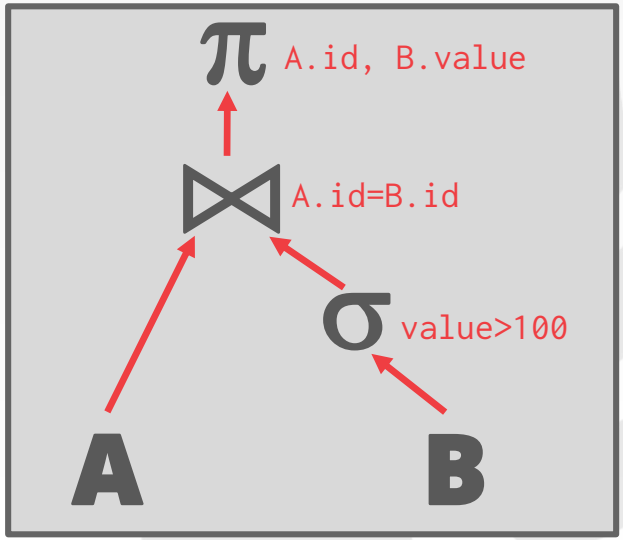
```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

```
for t in child.Next():
  if evalPred(t): emit(t)
```

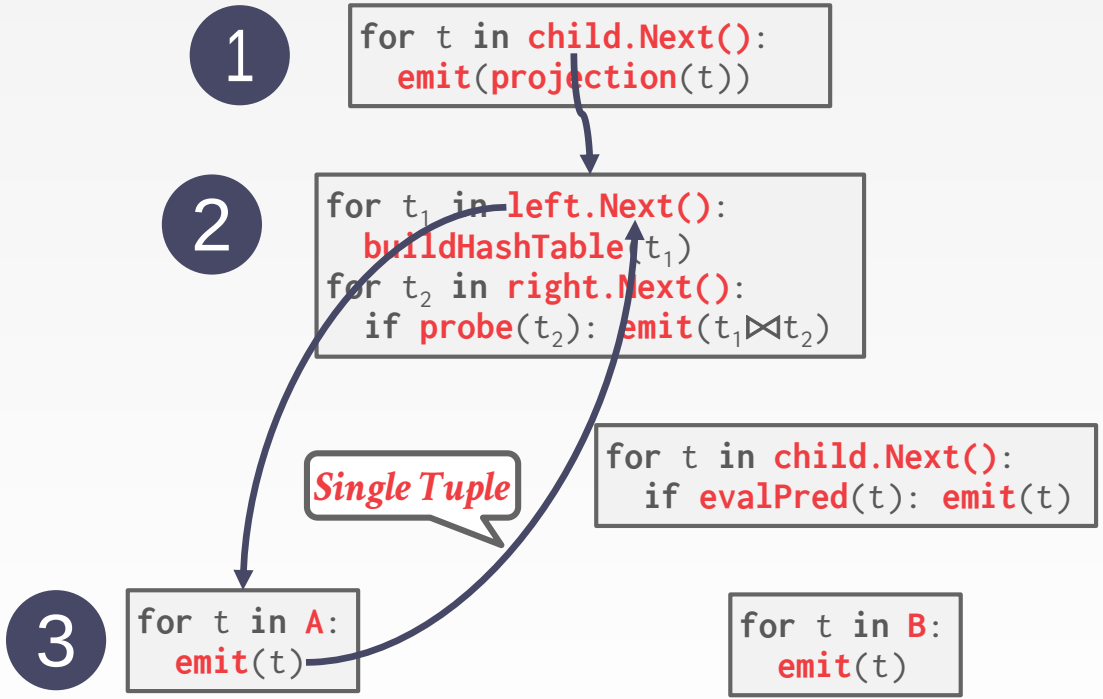
```
for t in A:
  emit(t)
```

```
for t in B:
  emit(t)
```

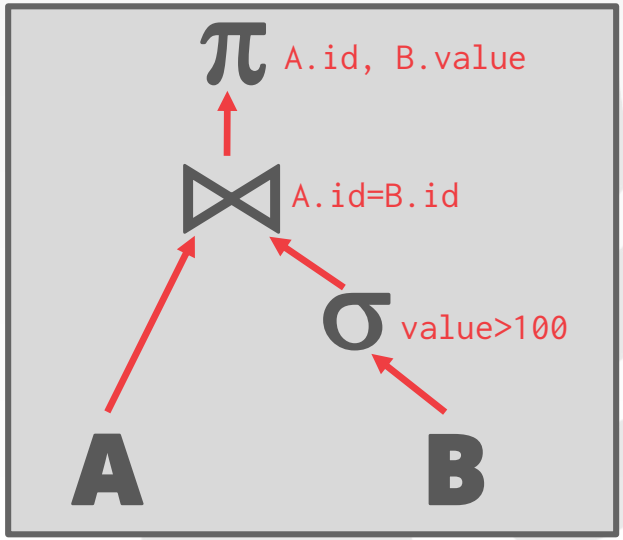
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



ITERATOR MODEL




```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



ITERATOR MODEL

1

```
for t in child.Next():
  emit(projection(t))
```

2

```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

3

```
for t in A:
  emit(t)
```

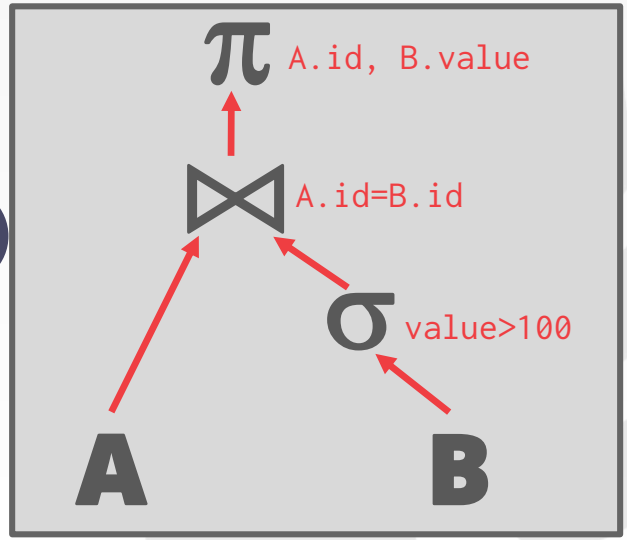
4

```
for t in child.Next():
  if evalPred(t): emit(t)
```

5

```
for t in B:
  emit(t)
```

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



ITERATOR MODEL

1

```
for t in child.Next():
  emit(projection(t))
```

2

```
for t1 in left.Next():
  buildHashTable(t1)
for t2 in right.Next():
  if probe(t2): emit(t1⋈t2)
```

3

```
for t in A:
  emit(t)
```

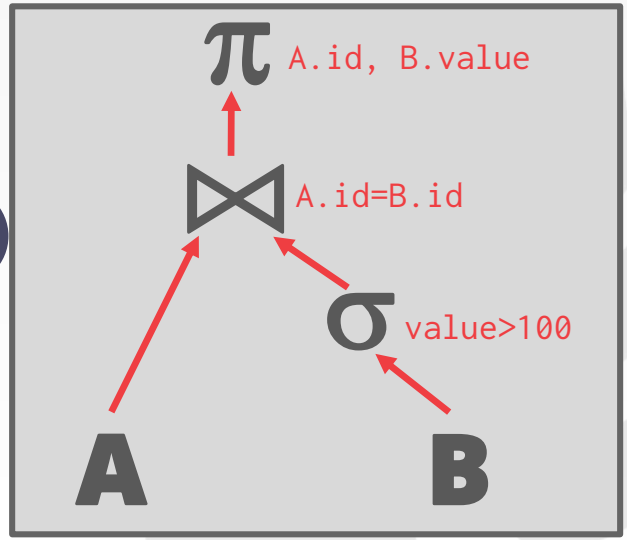
4

```
for t in child.Next():
  if evalPred(t): emit(t)
```

5

```
for t in B:
  emit(t)
```

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



ITERATOR MODEL

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

Some operators have to block until their children emit all of 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
```

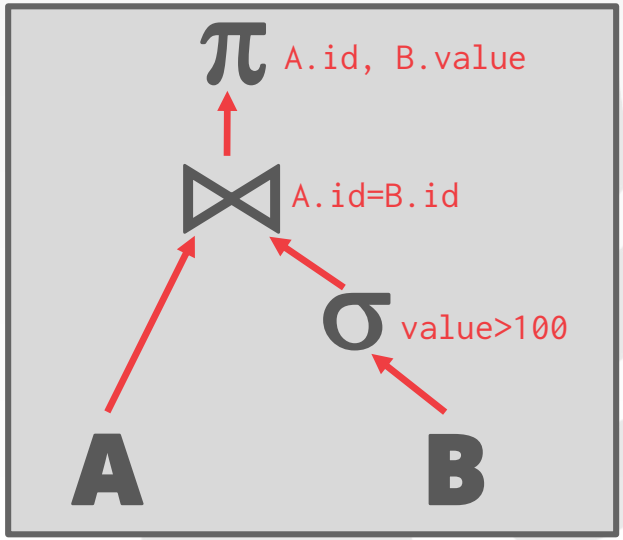
```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
return out
```

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

```
out = { }
for t in A:
  out.add(t)
return out
```

```
out = { }
for t in B:
  out.add(t)
return out
```

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.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
```

3

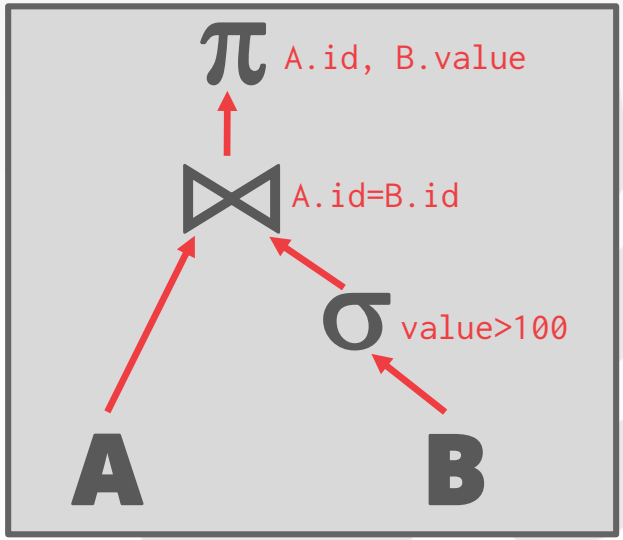
```
out = { }
for t in A:
  out.add(t)
return out
```

All Tuples

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

```
out = { }
for t in B:
  out.add(t)
return out
```

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.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
```

3

```
out = { }
for t in A:
  out.add(t)
return out
```

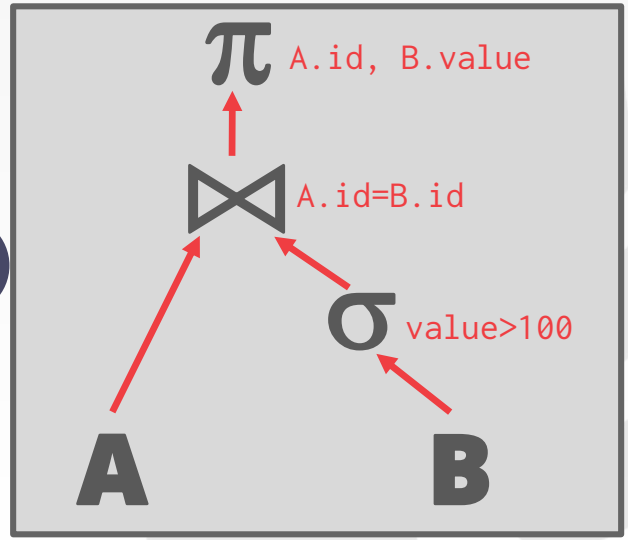
4

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

5

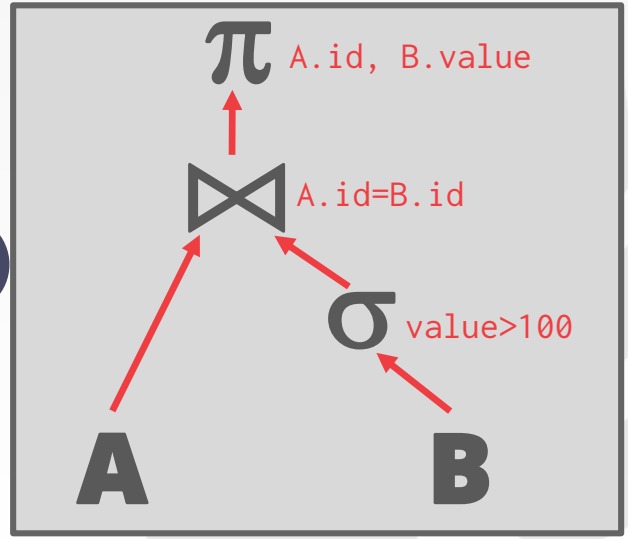
```
out = { }
for t in B:
  out.add(t)
return out
```

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



MATERIALIZATION MODEL

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



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
```

3

```
out = { }
for t in A:
  out.add(t)
return out
```

4

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

5

```
out = { }
for t in B:
  out.add(t)
return out
```

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, each operator implements a `next` function in this model.

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.Output():
  out.add(projection(t))
  if |out|>n: emit(out)
```

1

```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

2

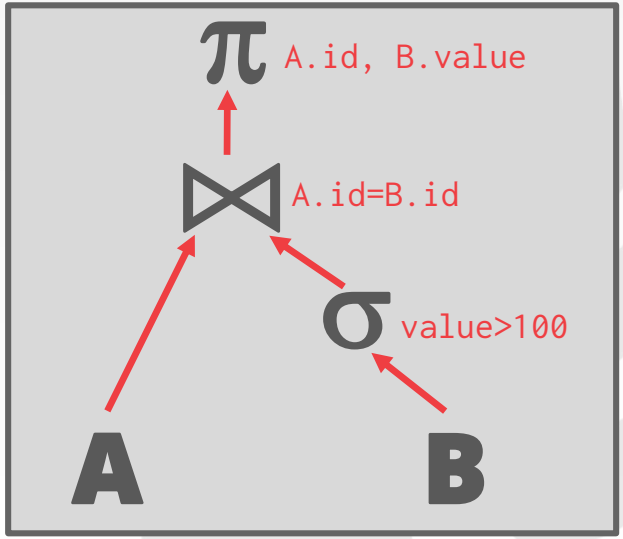
```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

```
out = { }
for t in A:
  out.add(t)
  if |out|>n: emit(out)
```

3

```
out = { }
for t in B:
  out.add(t)
  if |out|>n: emit(out)
```

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.value > 100
```



VECTORIZATION MODEL

```
out = { }
for t in child.Output():
  out.add(projection(t))
  if |out|>n: emit(out)
```

1

```
out = { }
for t1 in left.Output():
  buildHashTable(t1)
for t2 in right.Output():
  if probe(t2): out.add(t1⋈t2)
  if |out|>n: emit(out)
```

2

```
out = { }
for t in A:
  out.add(t)
  if |out|>n: emit(out)
```

3

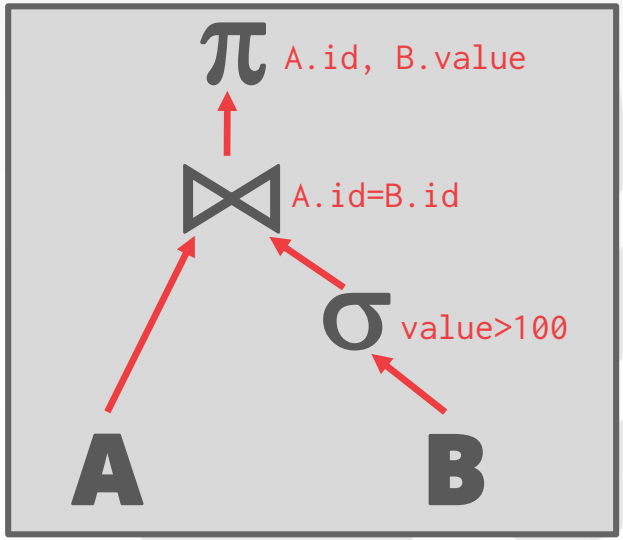
```
out = { }
for t in child.Output():
  if evalPred(t): out.add(t)
  if |out|>n: emit(out)
```

4

```
out = { }
for t in B:
  out.add(t)
  if |out|>n: emit(out)
```

5

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND B.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.

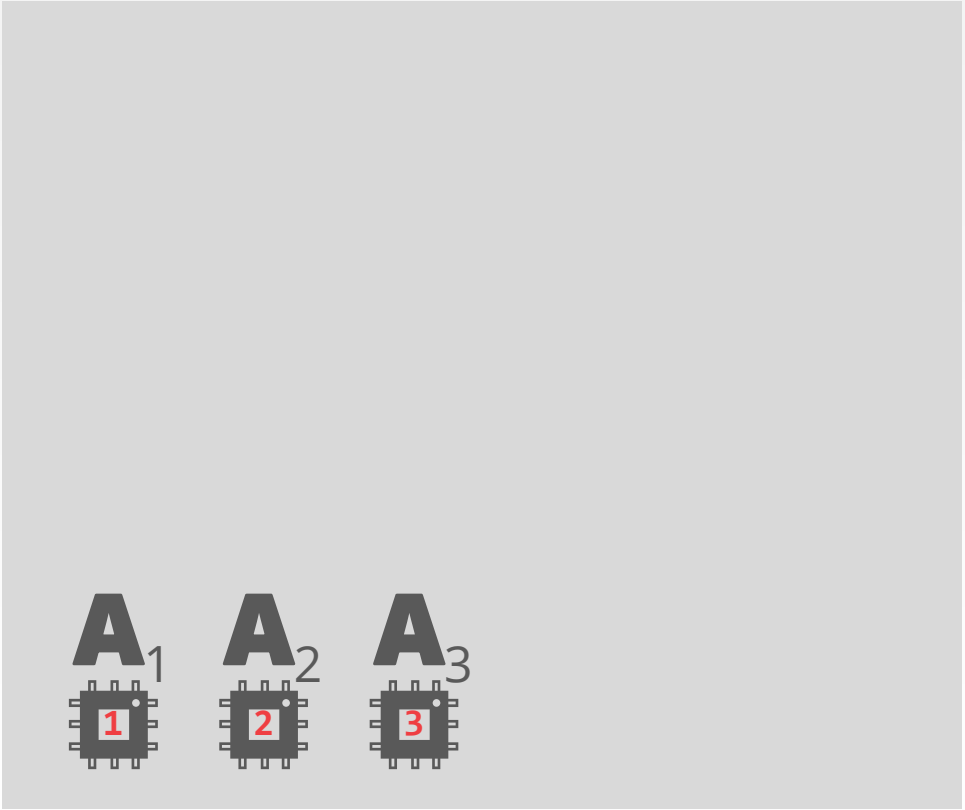
INTRA-OPERATOR PARALLELISM

Approach #1: Intra-Operator (Horizontal)

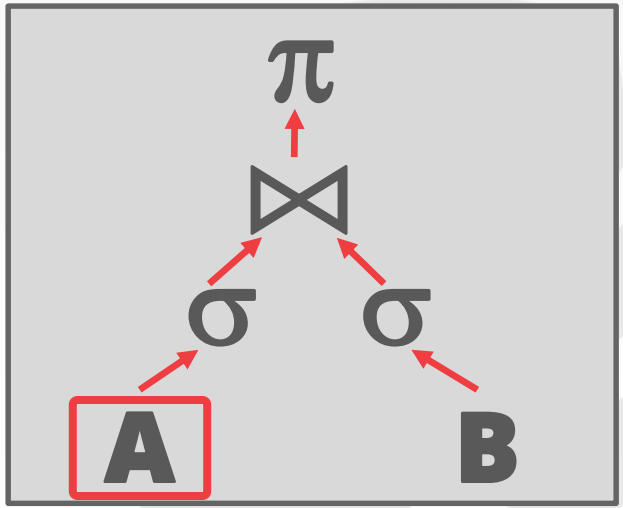
→ Operators are decomposed into independent instances that perform the same function on different subsets of data.

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

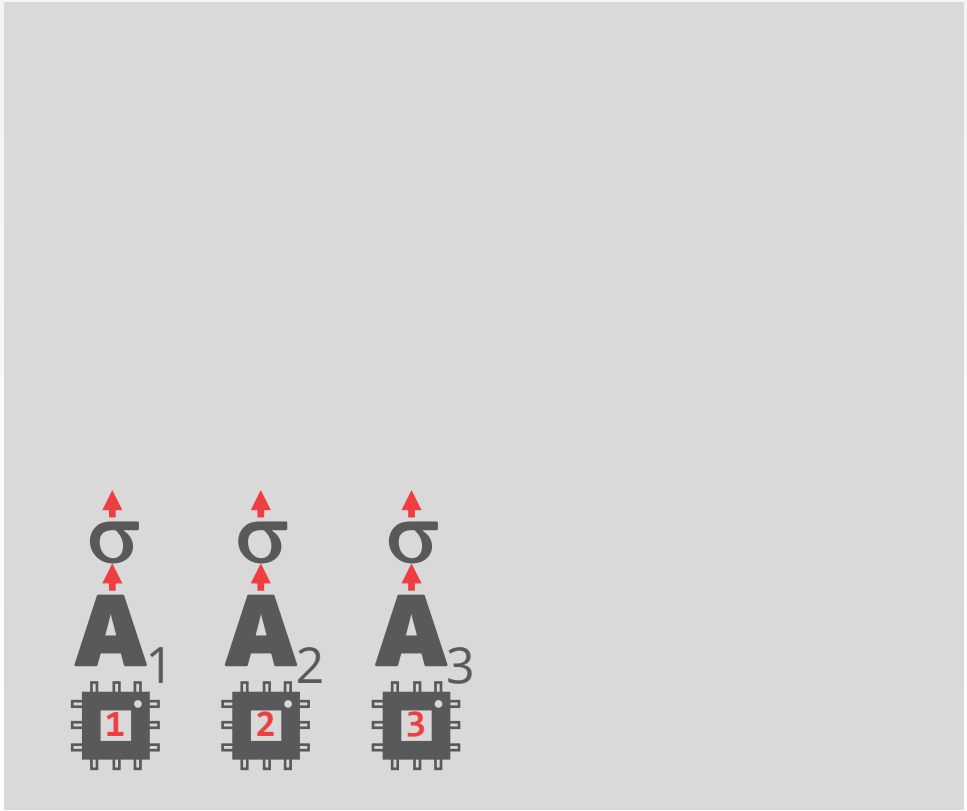
INTRA-OPERATOR PARALLELISM



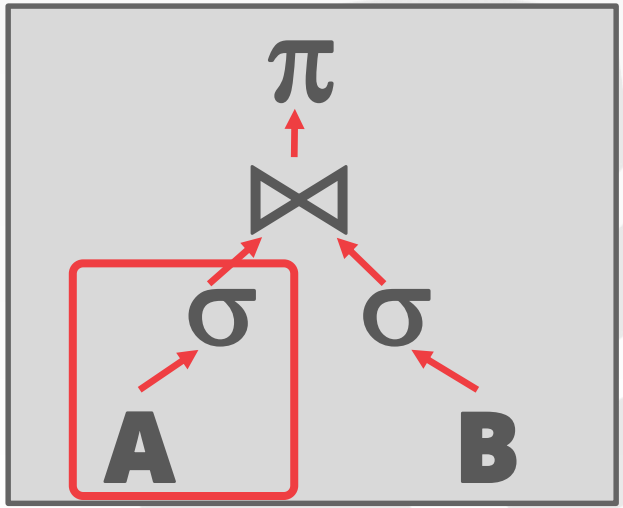

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



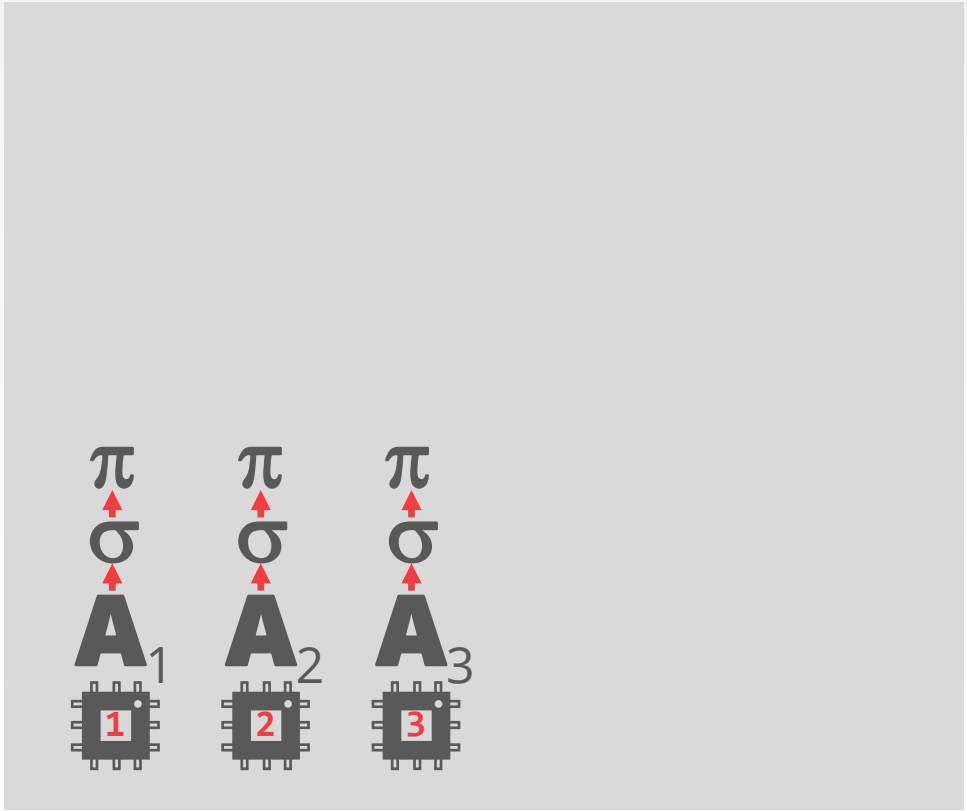
INTRA-OPERATOR PARALLELISM



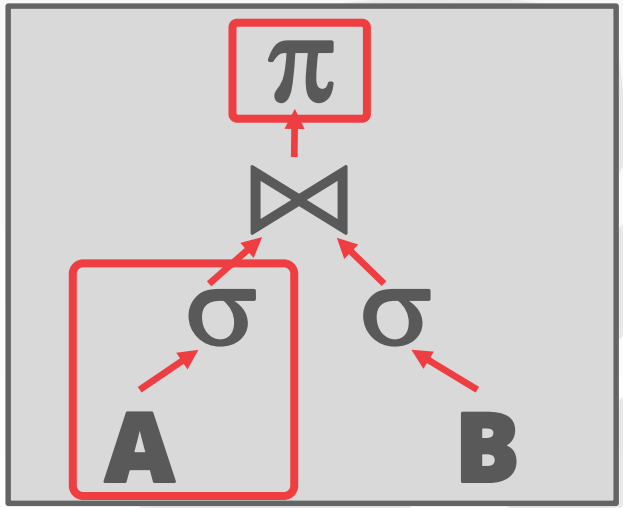

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



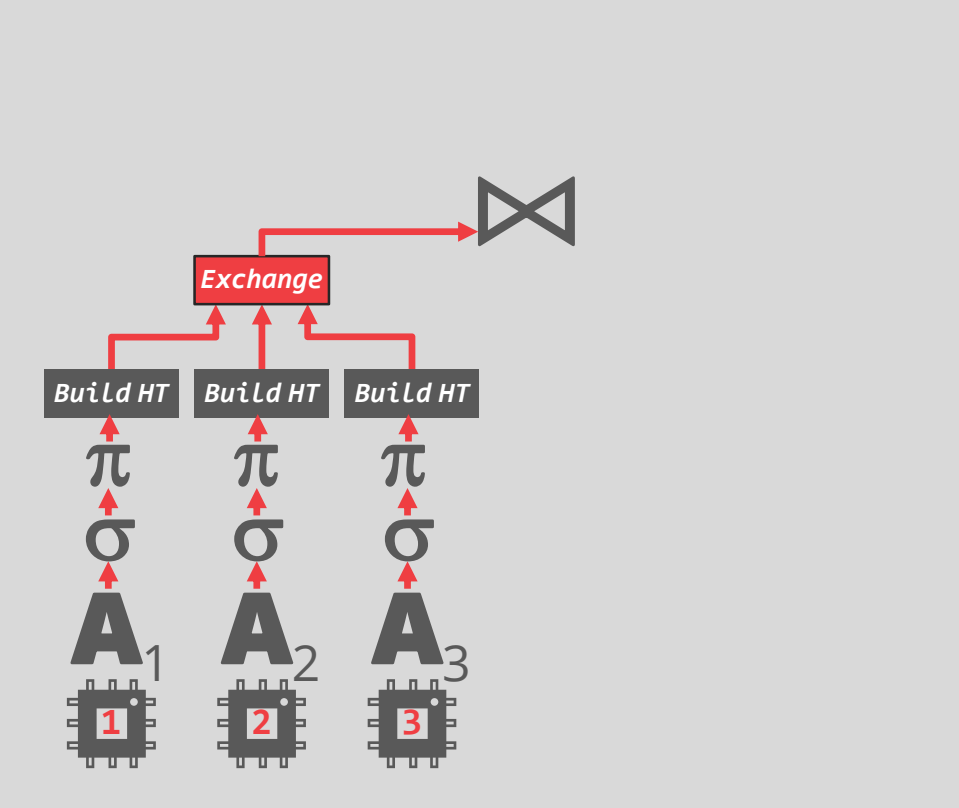
INTRA-OPERATOR PARALLELISM



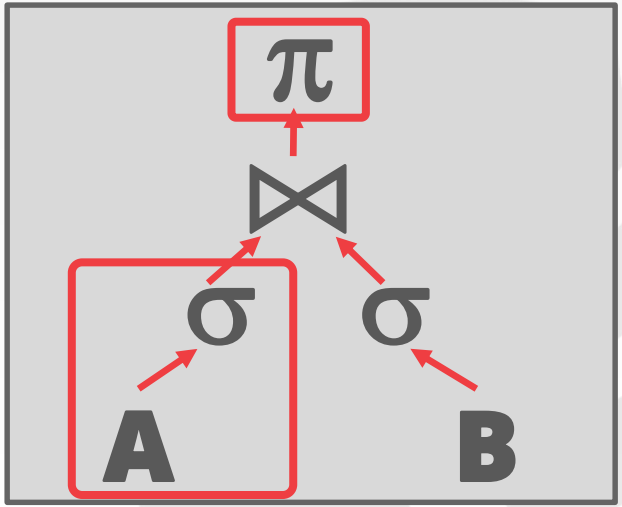

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



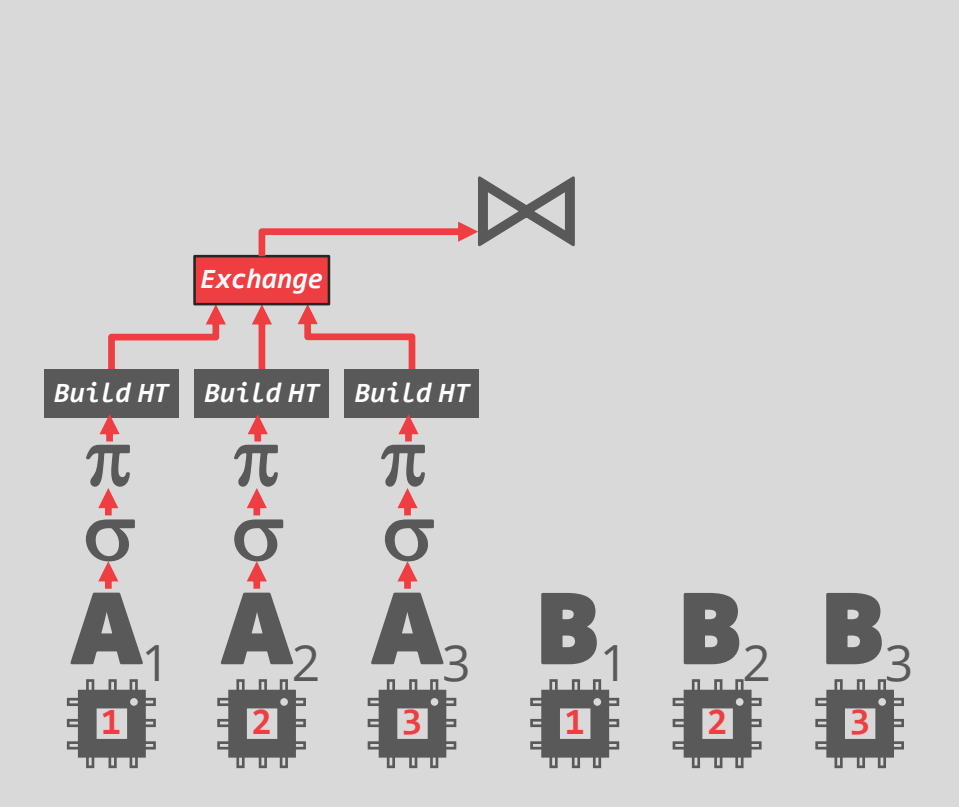
INTRA-OPERATOR PARALLELISM



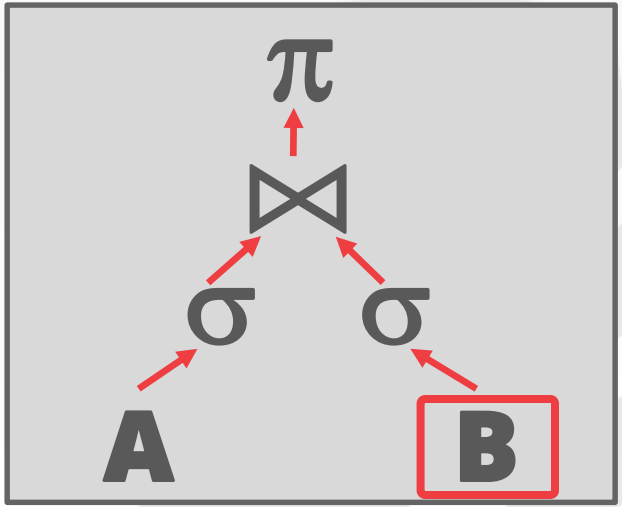

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



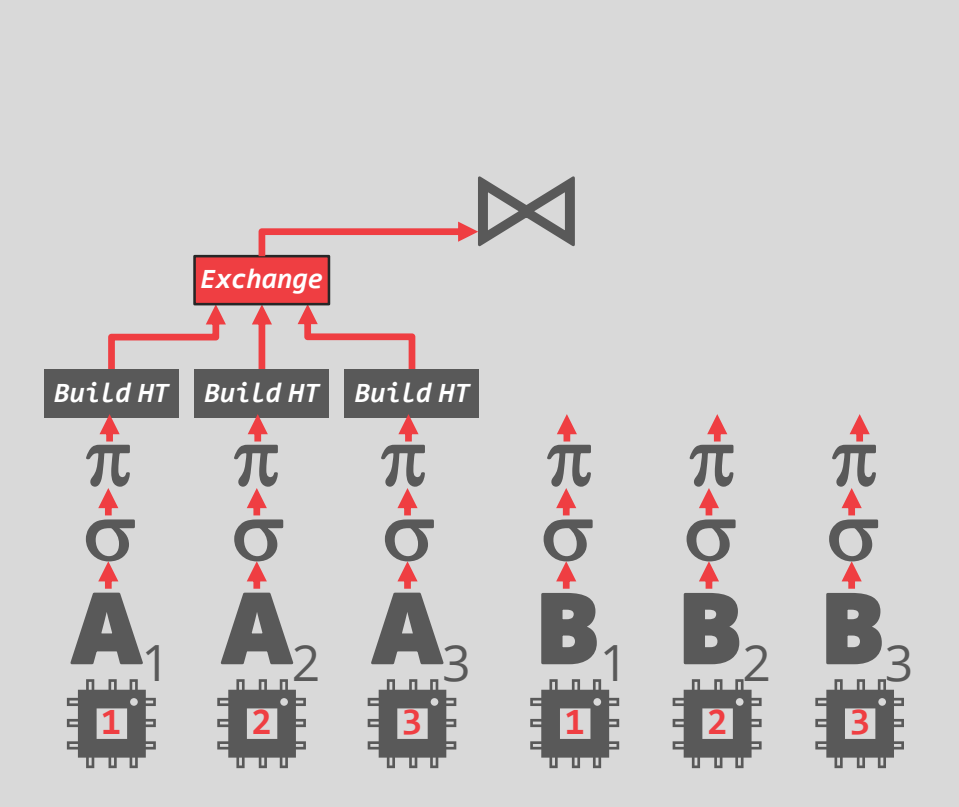
INTRA-OPERATOR PARALLELISM



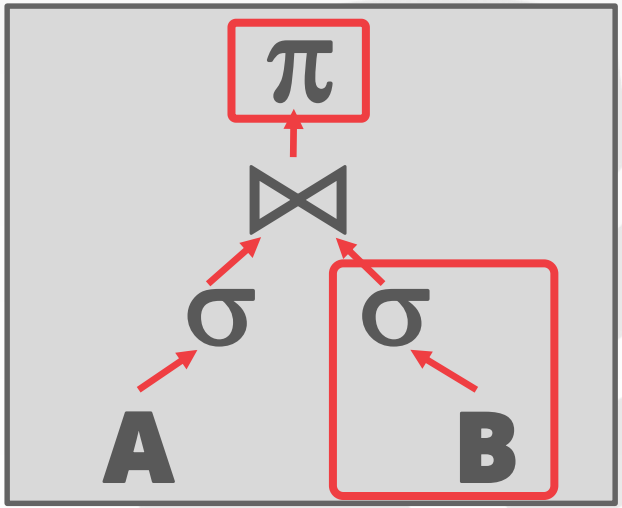

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



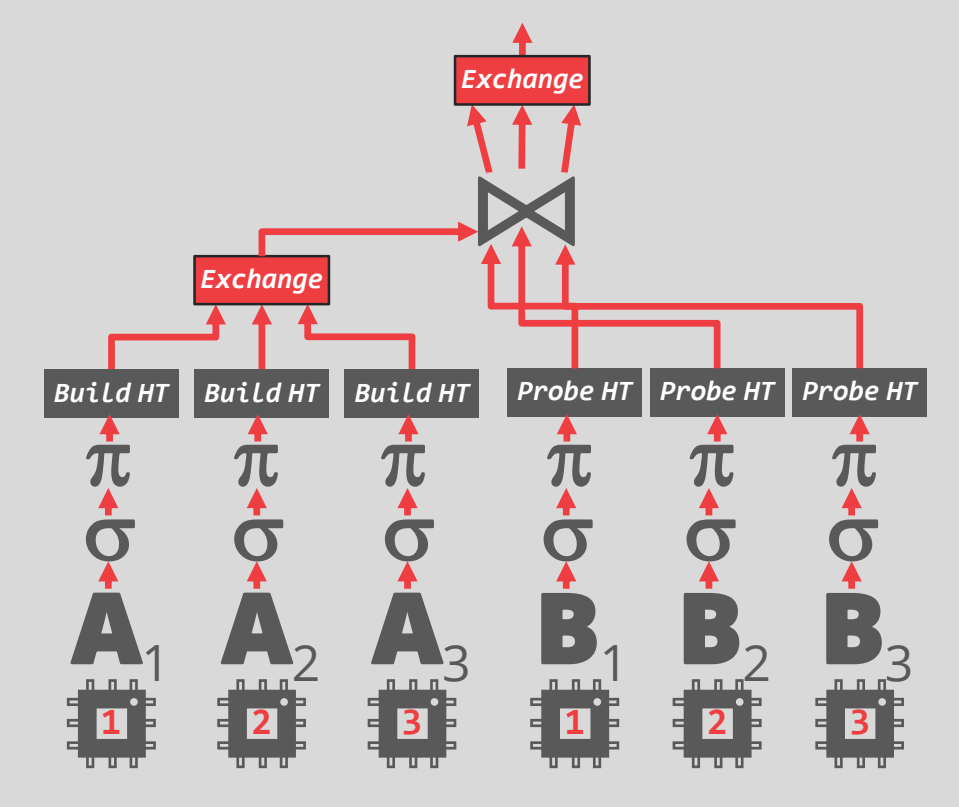
INTRA-OPERATOR PARALLELISM



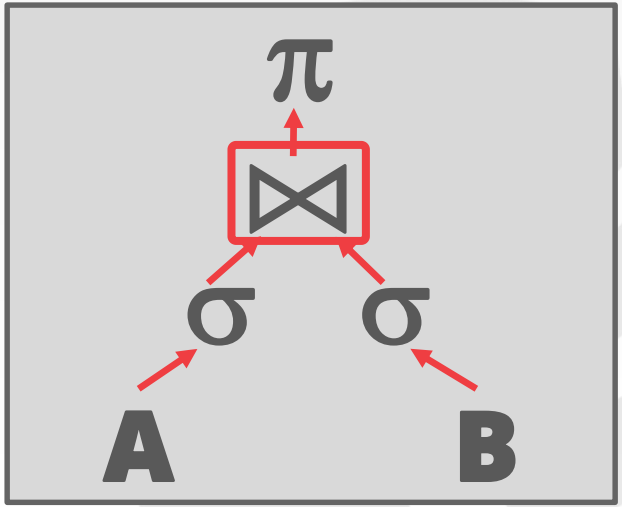

```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



INTRA-OPERATOR PARALLELISM




```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND 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.

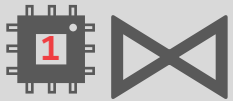
Also called **pipelined parallelism**.

AFAIK, this approach is not widely used in traditional relational DBMSs.

→ Not all operators can emit output until they have seen all of the tuples from their children.

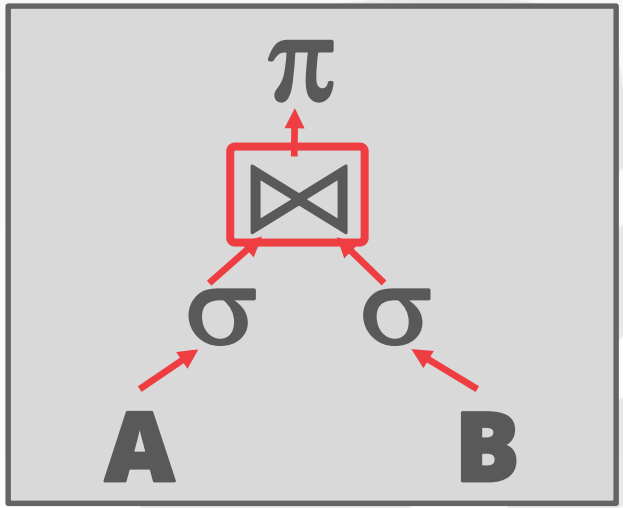
→ It is more common in **stream processing systems**.

INTER-OPERATOR PARALLELISM

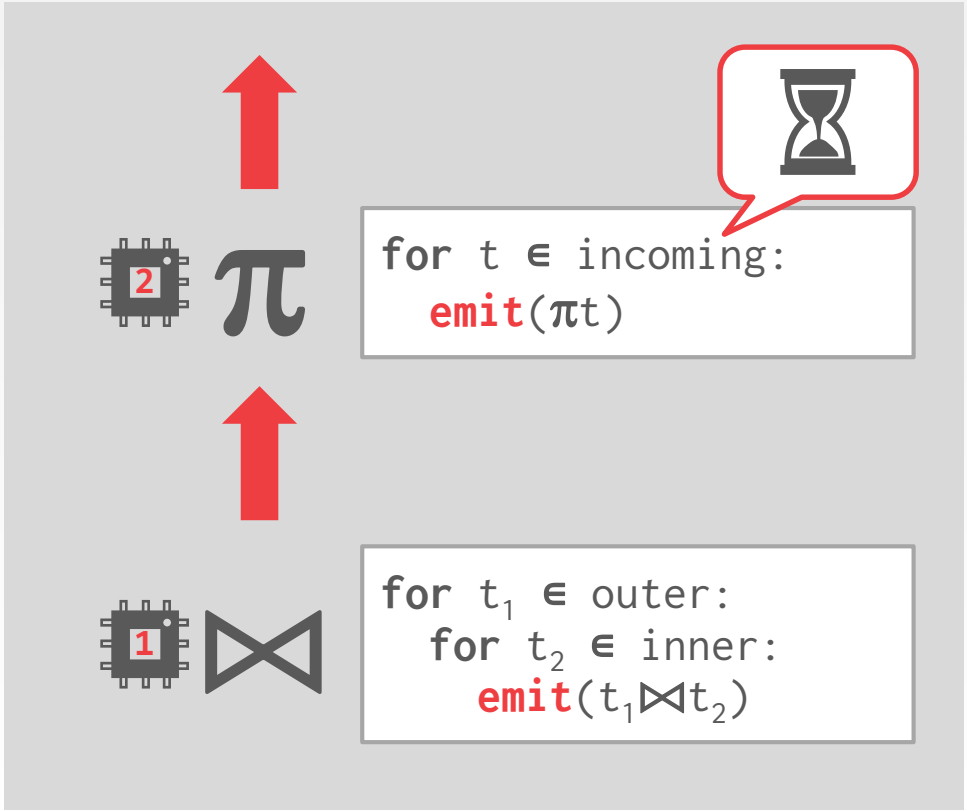



```
for t1 ∈ outer:
  for t2 ∈ inner:
    emit(t1⋈t2)
```

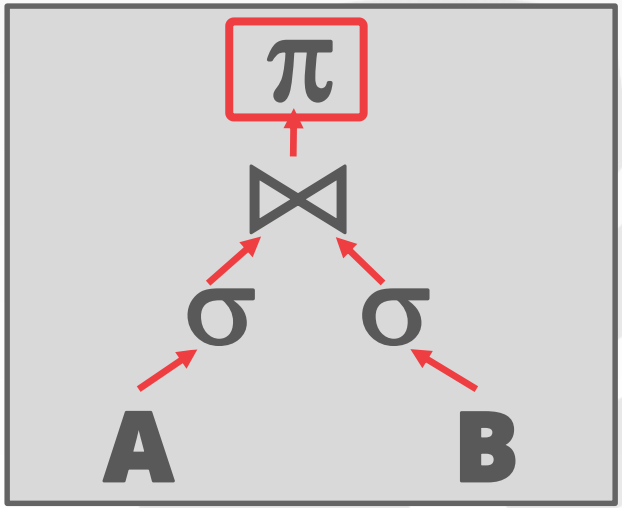
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



INTER-OPERATOR PARALLELISM




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



OBSERVATION

Coming up with 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

User-defined Functions

Stored Procedures