

05: Query Execution

Andrew Crotty // CS497 // Fall 2023

Query Execution

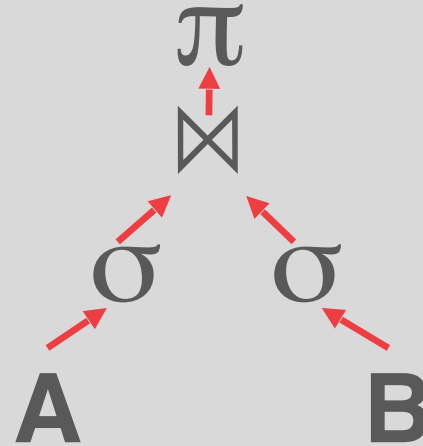
A query plan is a DAG of operators.

An operator instance is an invocation of an operator on a unique segment of data.

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

A task set is the collection of executable tasks for a logical pipeline.

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



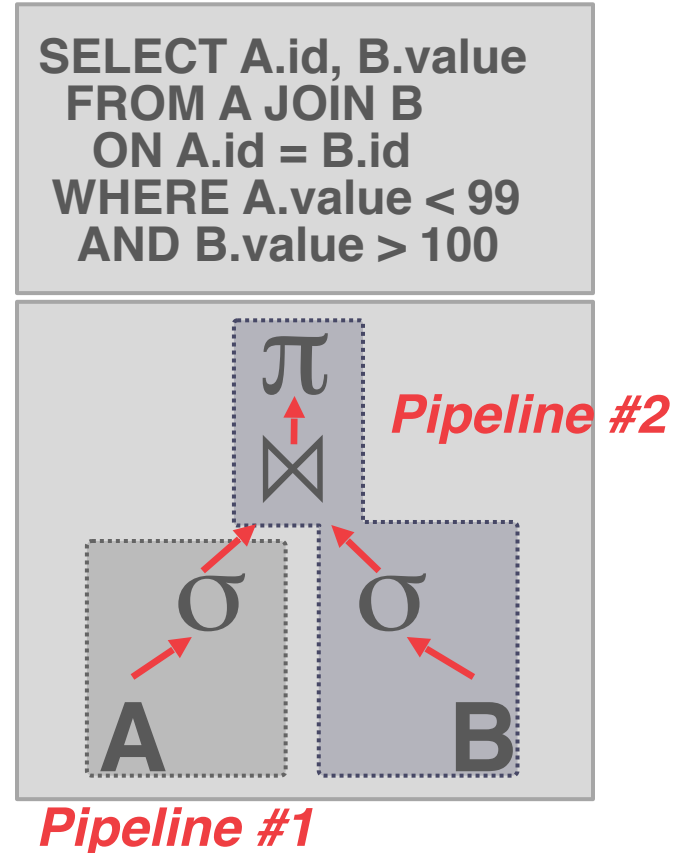
Query Execution

A query plan is a DAG of operators.

An operator instance is an invocation of an operator on a unique segment of data.

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

A task set is the collection of executable tasks for a logical pipeline.



Today's Agenda

Processing Models

Parallel Execution

Scheduling

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

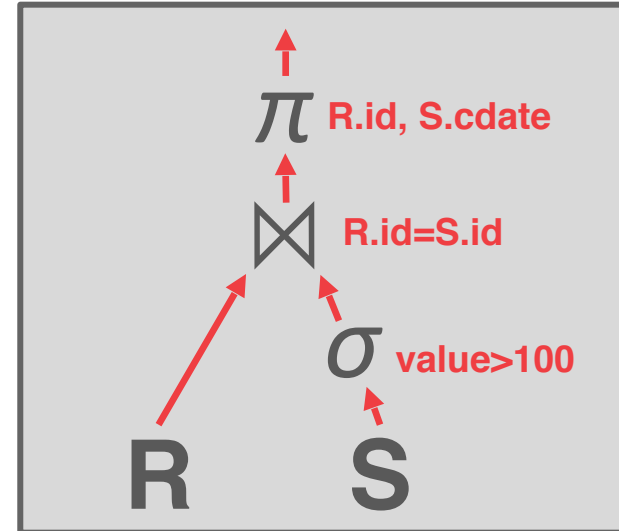
Each query plan operator implements a **next** function.

- 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.

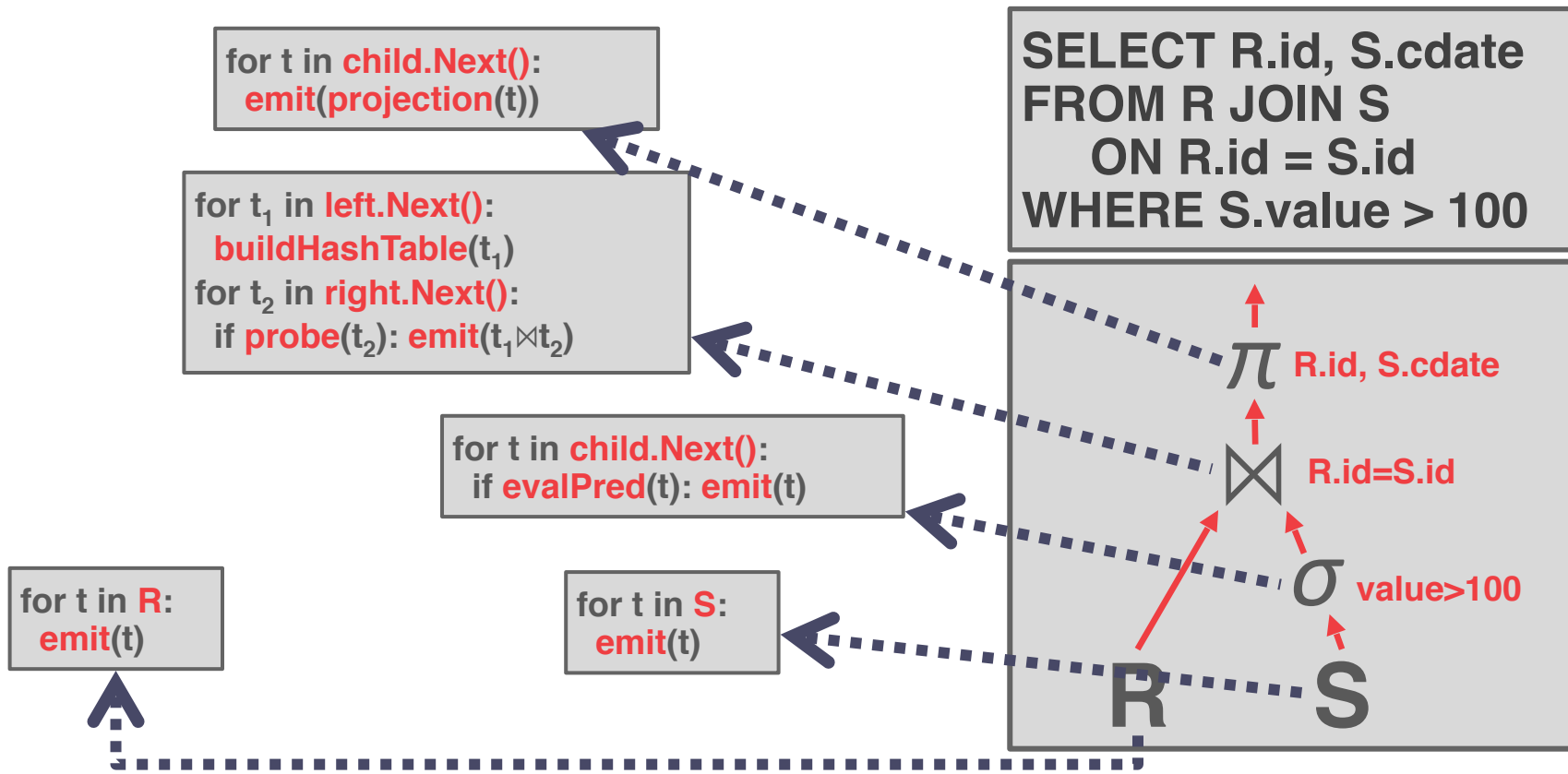
Also called Volcano or Pipeline Model.

Iterator Model

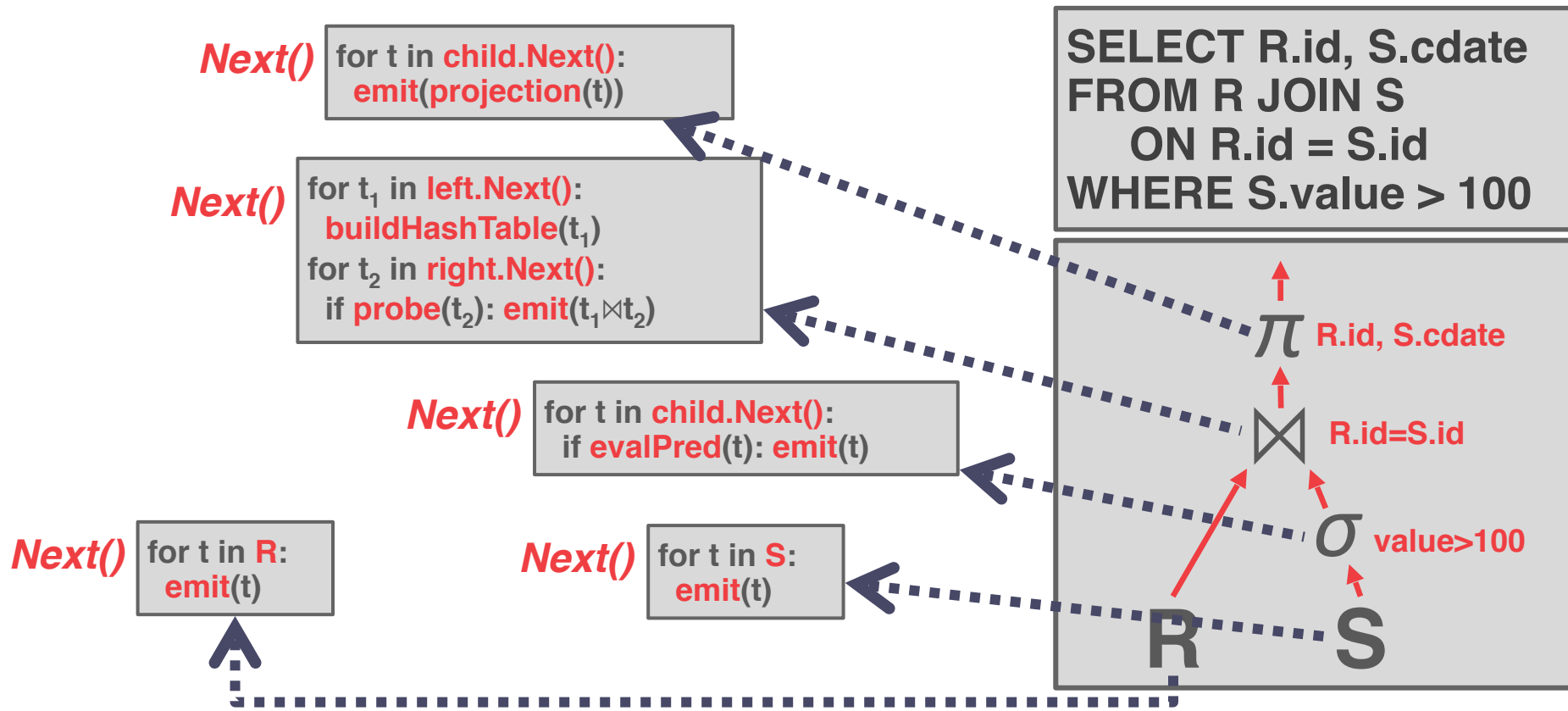
```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100
```



Iterator Model



Iterator Model



Iterator Model

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

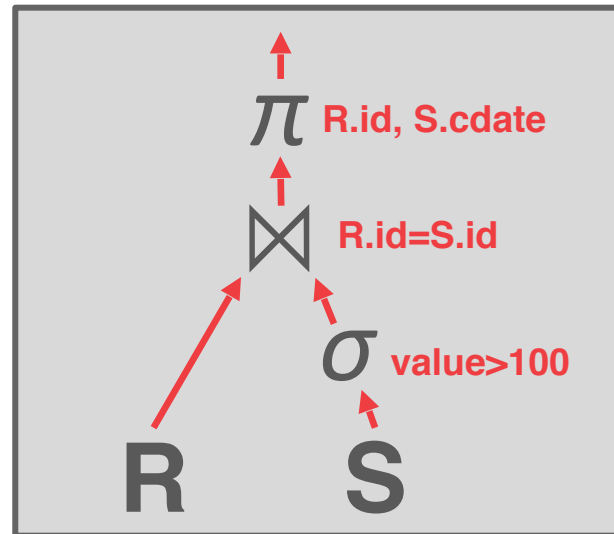
```
for t1 in left.Next():  
  buildHashTable(t1)  
for t2 in right.Next():  
  if probe(t2): emit(t1 ⋈ t2)
```

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

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

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

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



Iterator Model

1

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

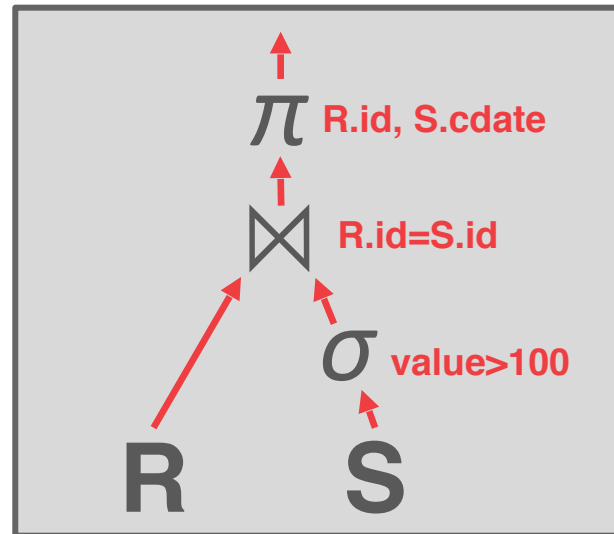
```
for t1 in left.Next():
    buildHashTable(t1)
for t2 in right.Next():
    if probe(t2): emit(t1 ⋈ t2)
```

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

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

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

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



Iterator Model

1

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

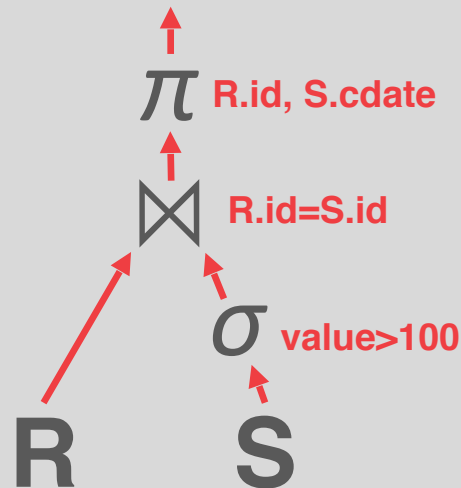
```
for t1 in left.Next():  
  buildHashTable(t1)  
for t2 in right.Next():  
  if probe(t2): emit(t1 ⋈ t2)
```

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

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

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

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



Iterator Model

1

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

2

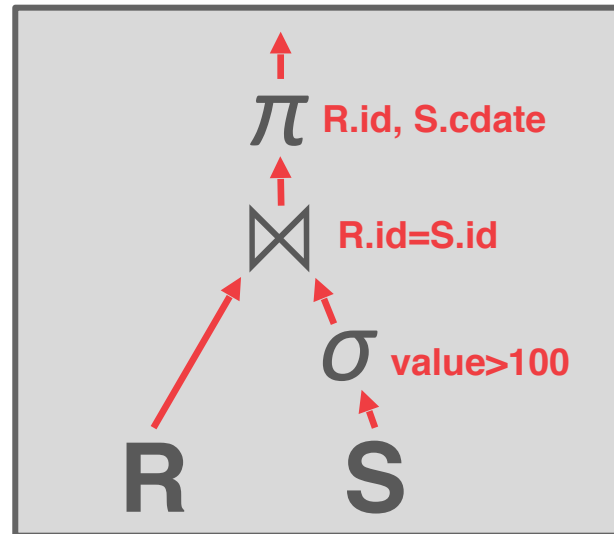
```
for t1 in left.Next():  
  buildHashTable(t1)  
for t2 in right.Next():  
  if probe(t2): emit(t1 ⋈ t2)
```

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

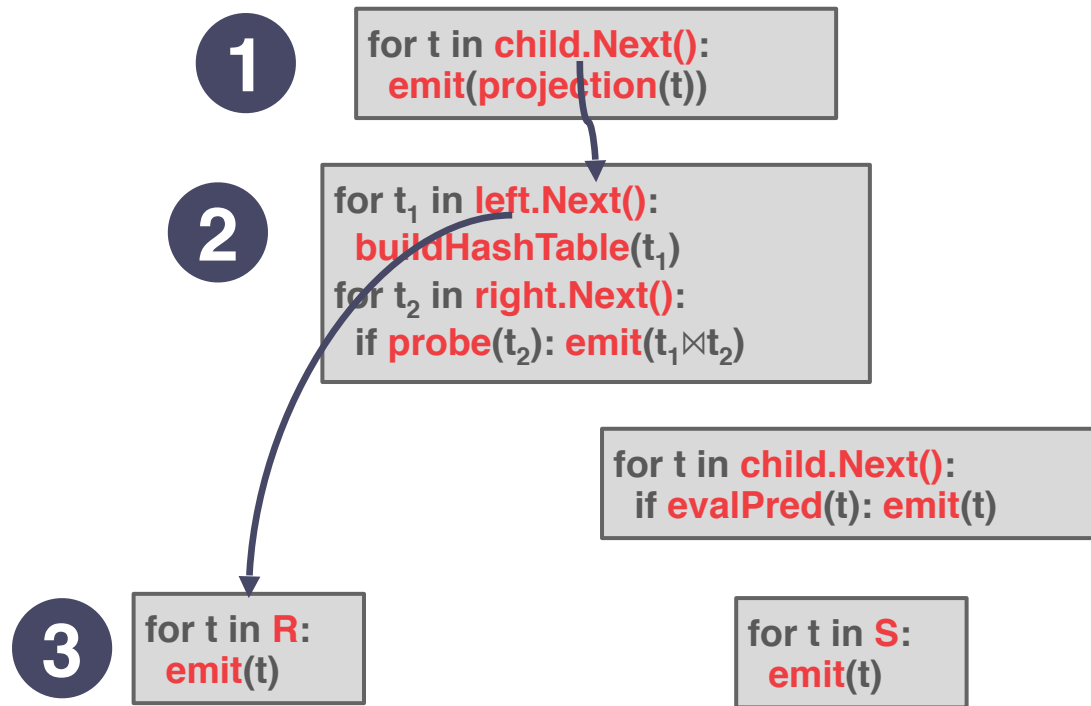
```
for t in R:  
  emit(t)
```

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

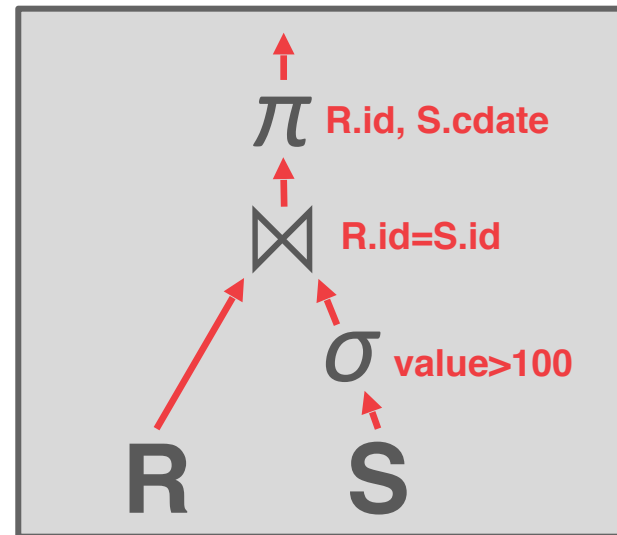
```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100
```



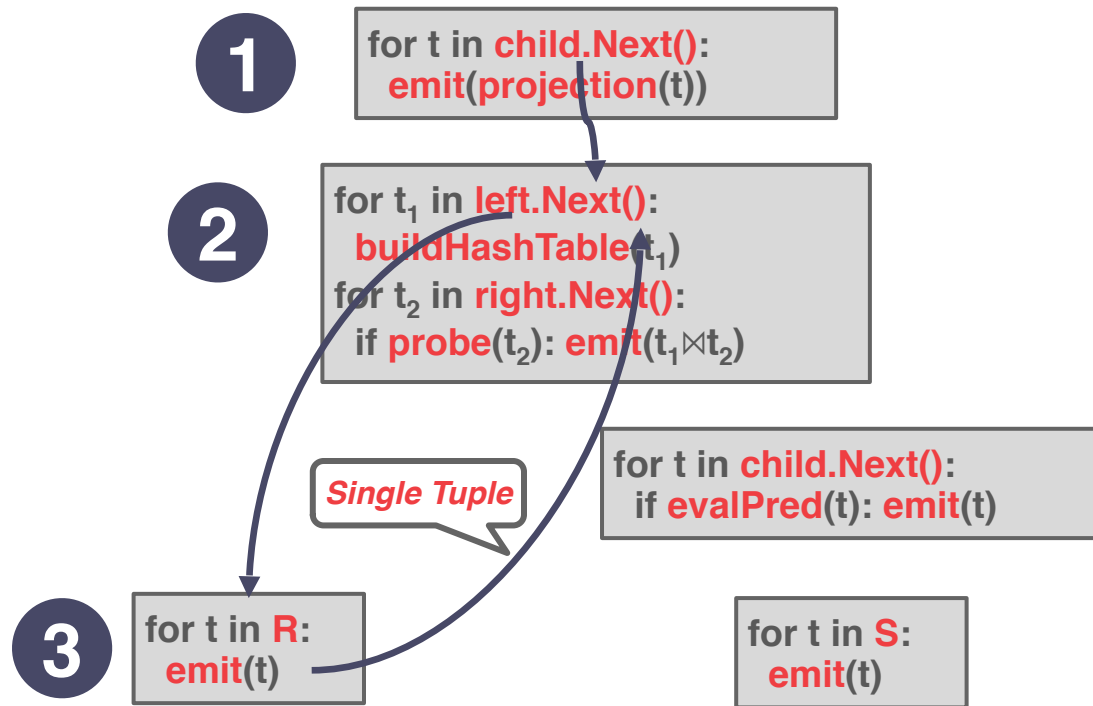
Iterator Model



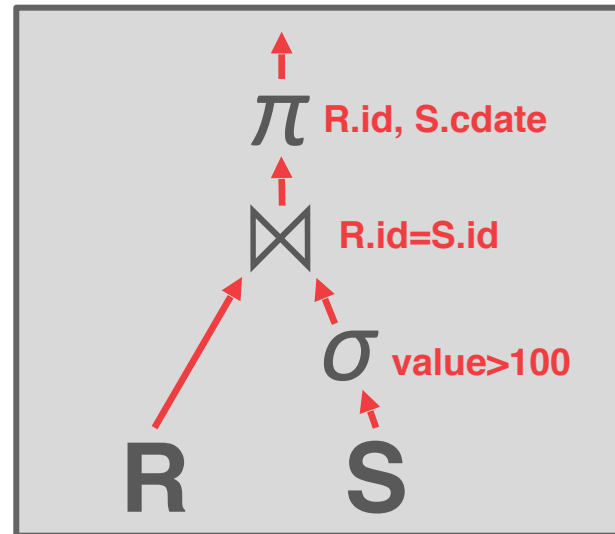
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



Iterator Model



```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.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)
```

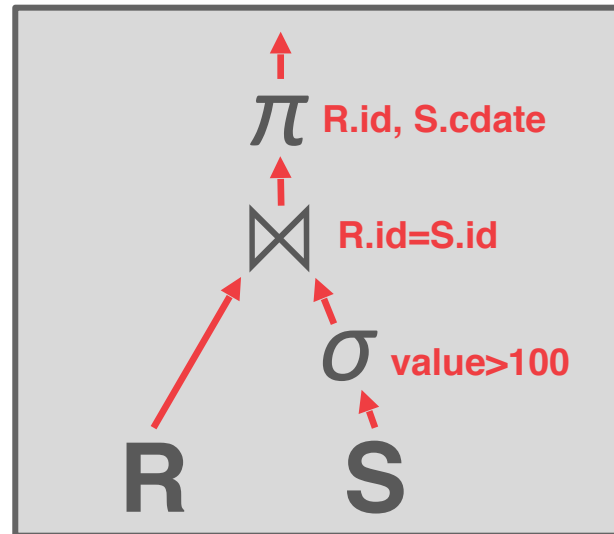
```
for t in child.Next():  
  if evalPred(t): emit(t)
```

3

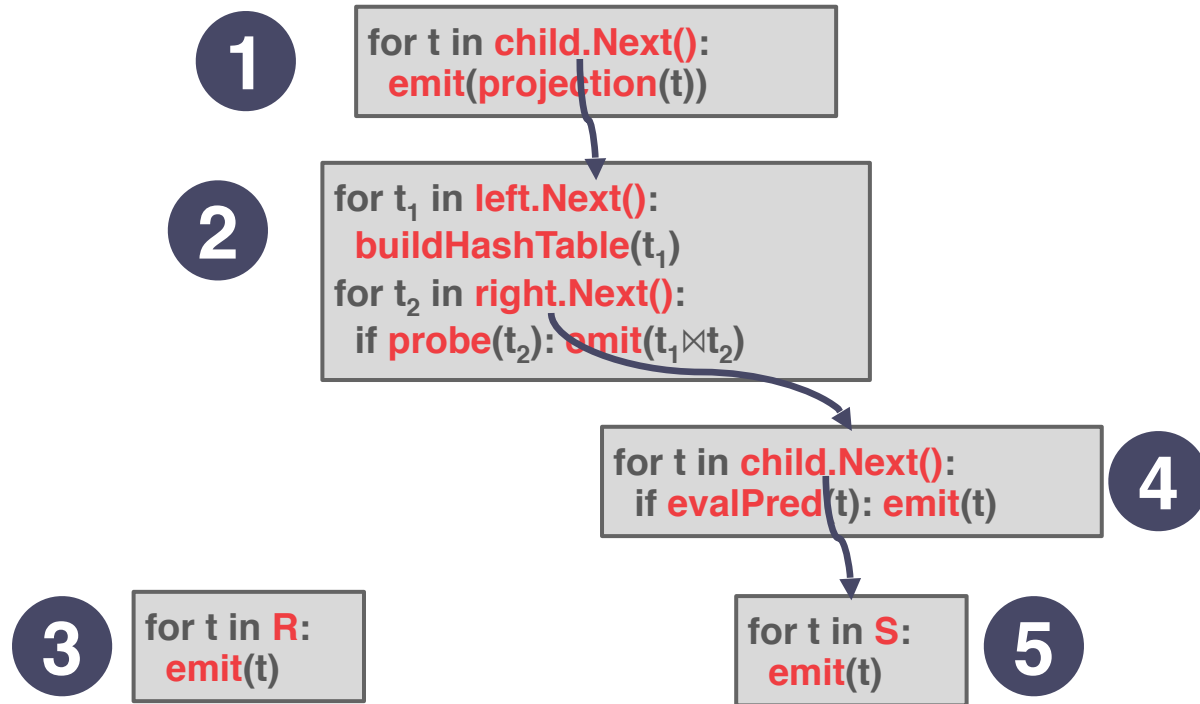
```
for t in R:  
  emit(t)
```

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

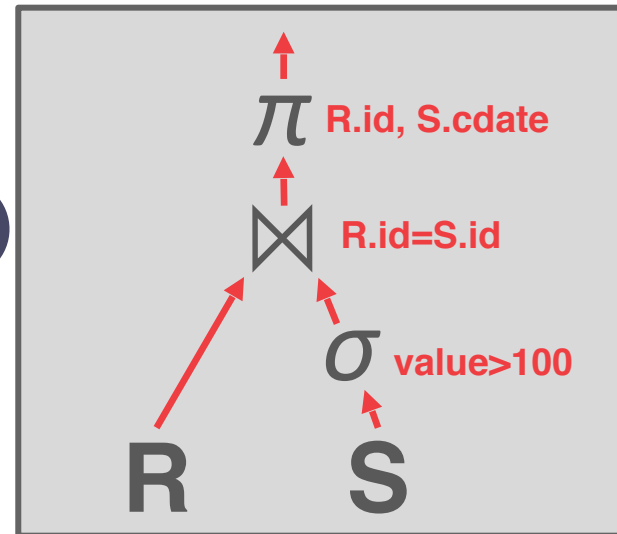
```
SELECT R.id, S.cdate  
FROM R JOIN S  
ON R.id = S.id  
WHERE S.value > 100
```



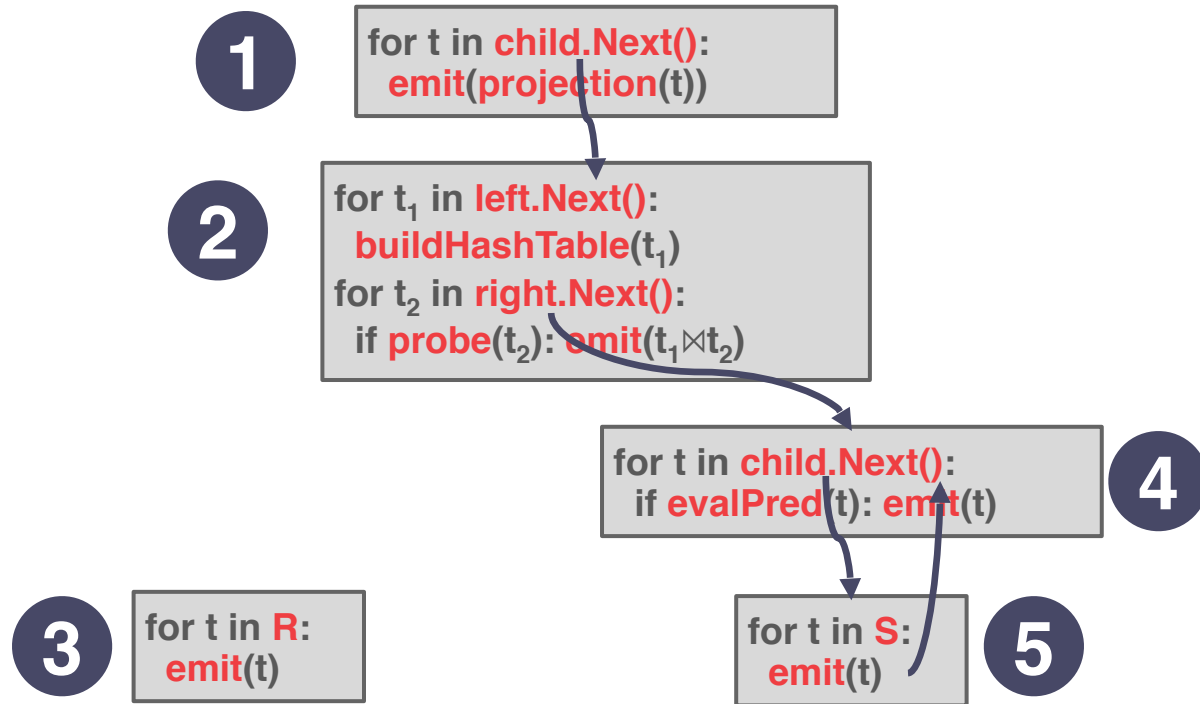
Iterator Model



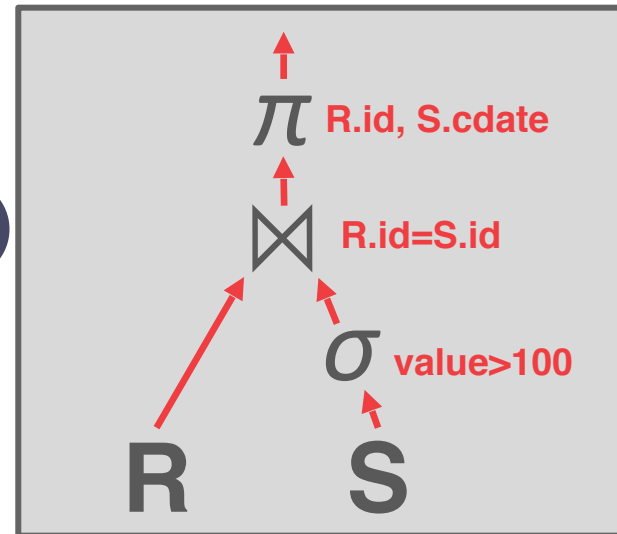
```
SELECT R.id, S.cdate
FROM R JOIN S
      ON R.id = S.id
WHERE S.value > 100
```



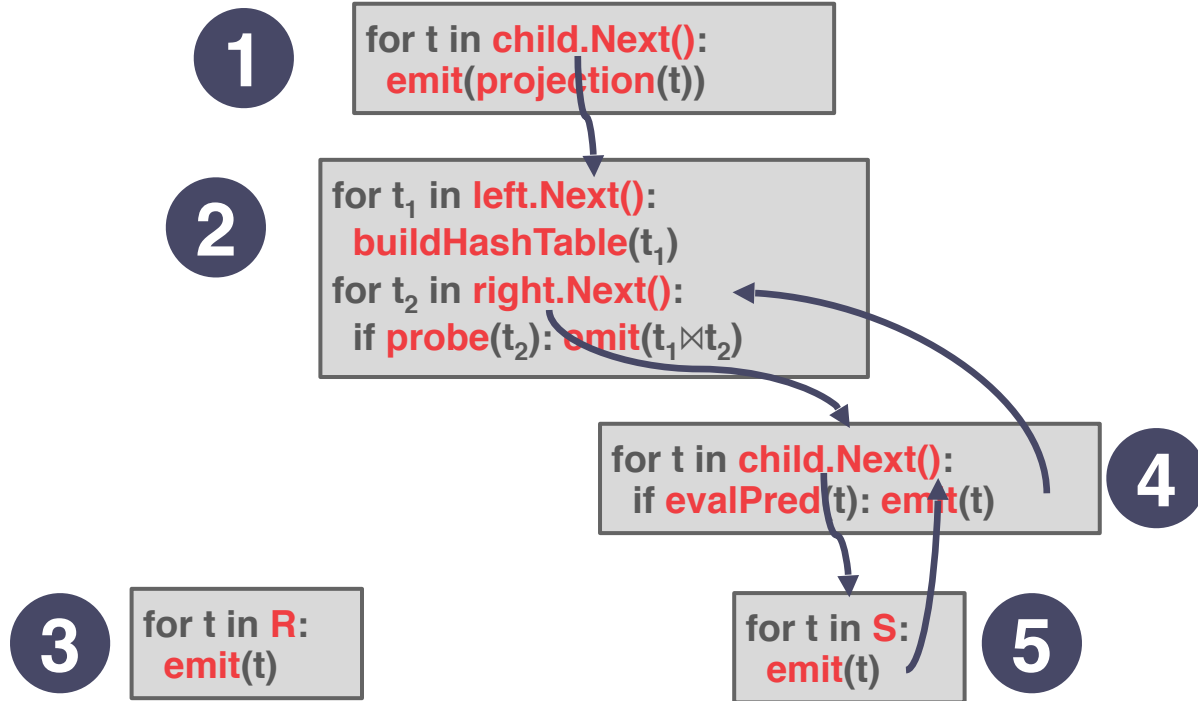
Iterator Model



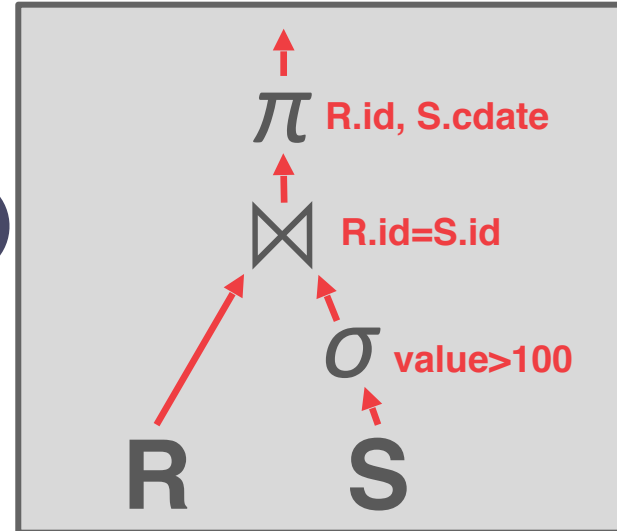
```
SELECT R.id, S.cdate
FROM R JOIN S
      ON R.id = S.id
WHERE S.value > 100
```



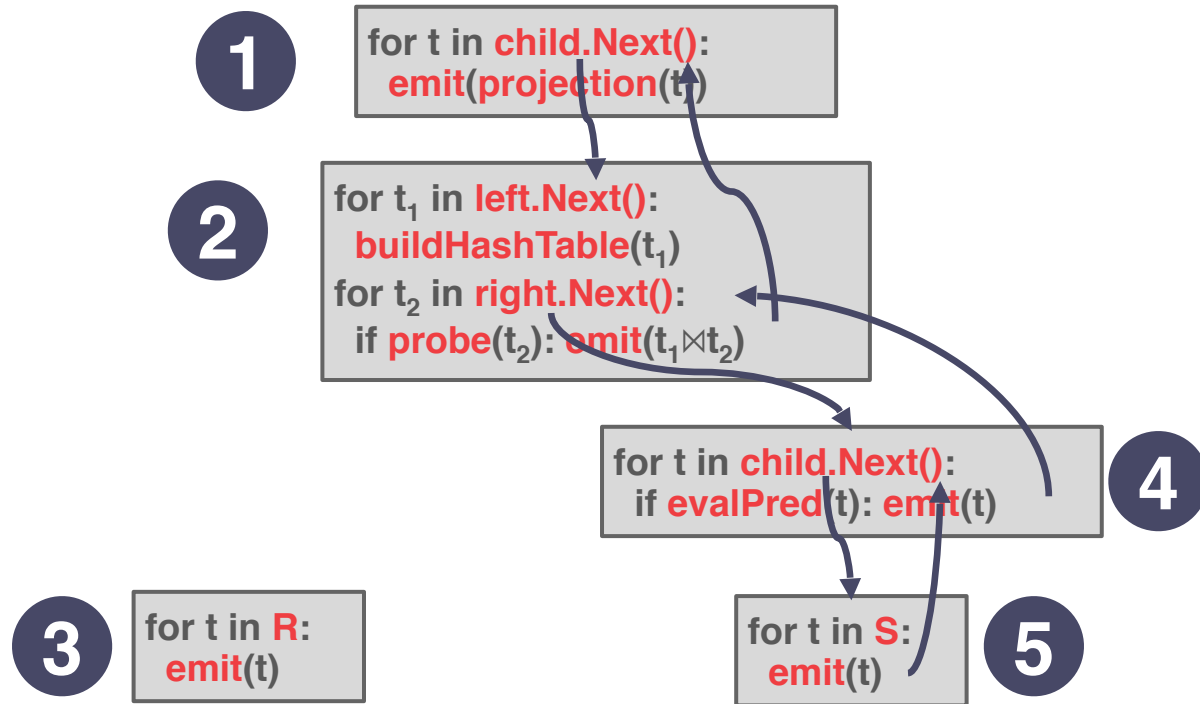
Iterator Model



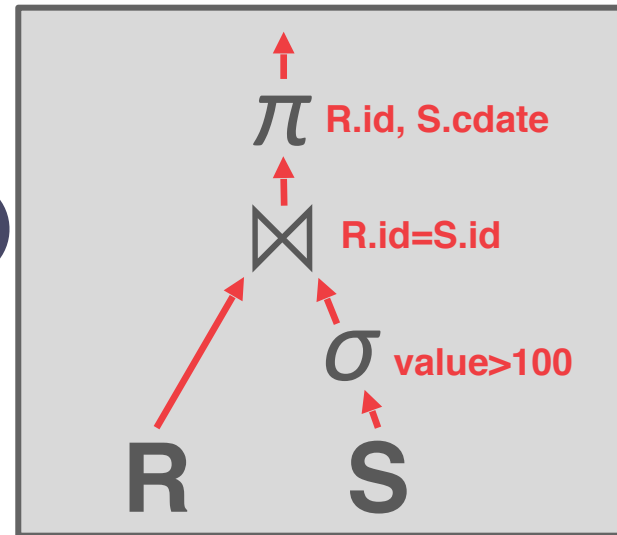
```
SELECT R.id, S.cdate
FROM R JOIN S
      ON R.id = S.id
WHERE S.value > 100
```



Iterator Model



```
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" its output as a single result.
- The DBMS can push down hints (e.g., **LIMIT**) 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

```

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 R:
    out.add(t)
return out

```

```

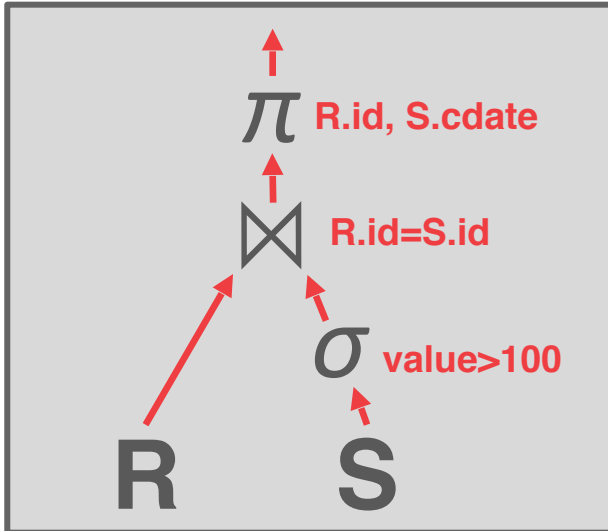
out = [ ]
for t in S:
    out.add(t)
return out

```

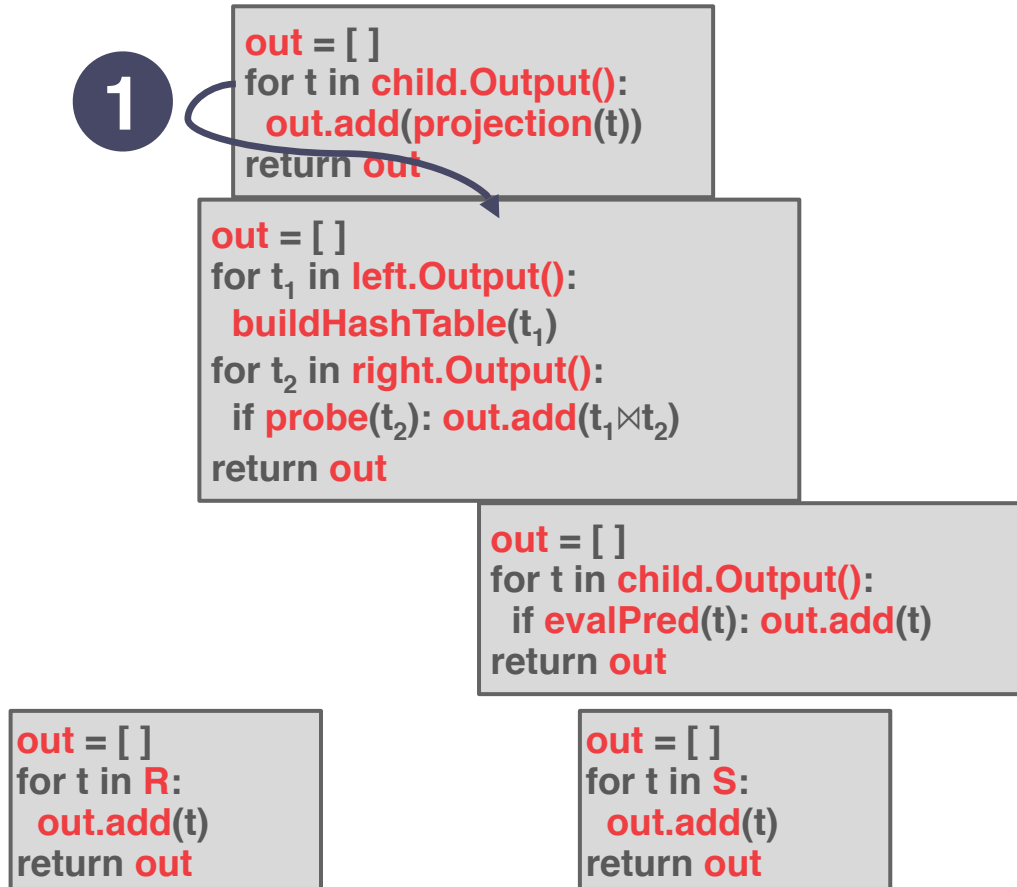
```

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

```

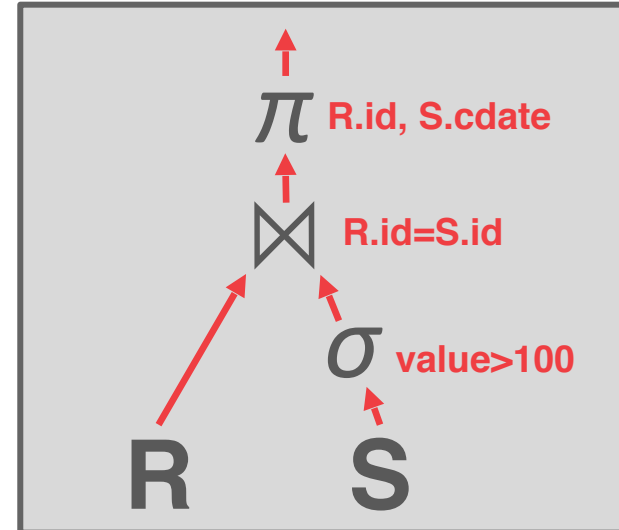


Materialization Model

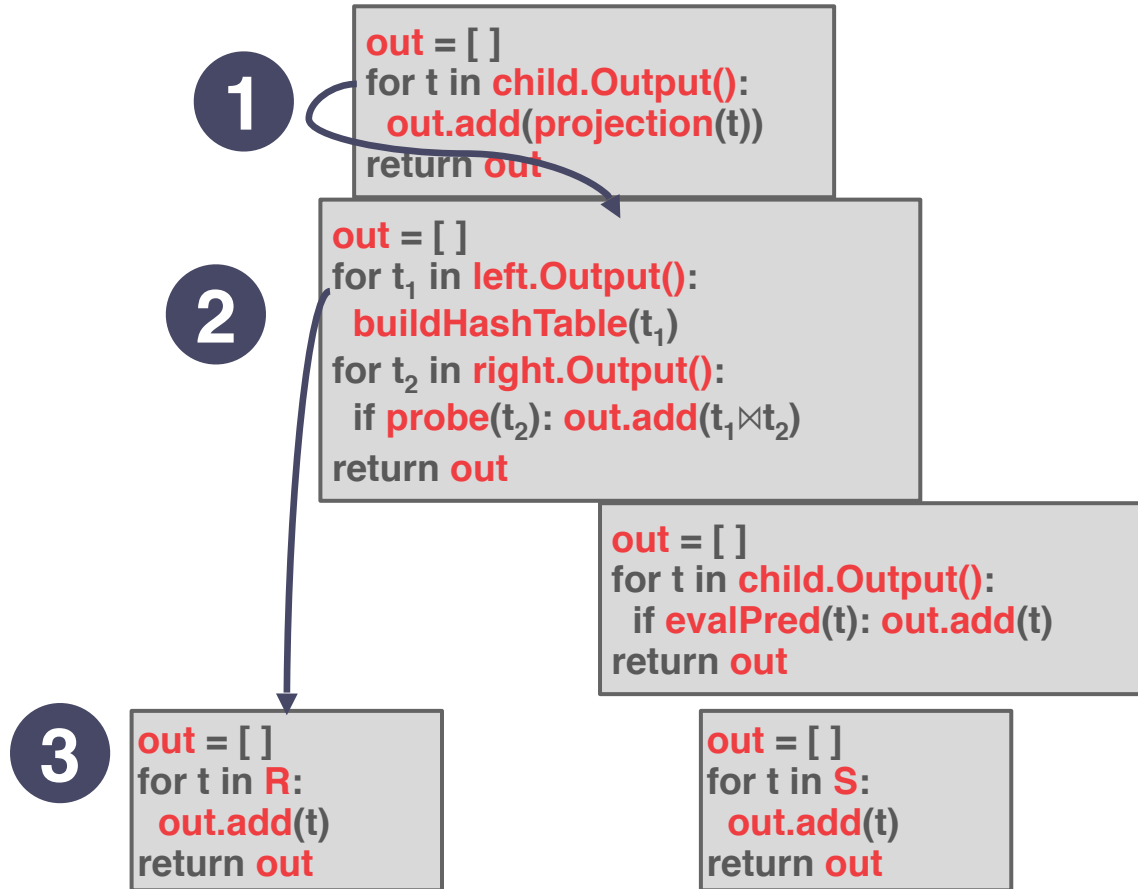


```

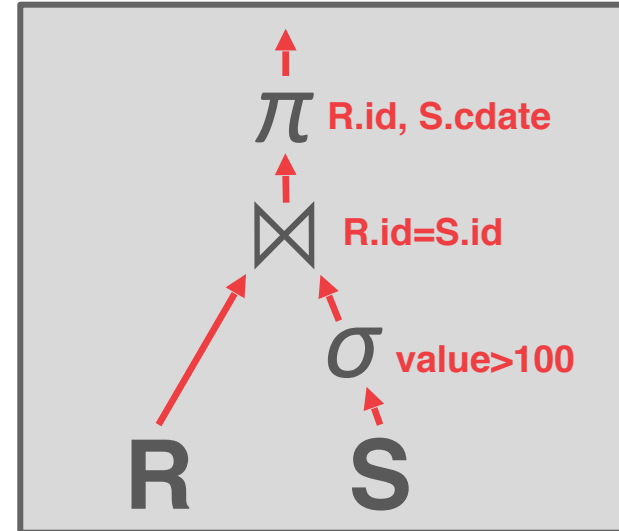
SELECT R.id, S.cdate
FROM R JOIN S
      ON R.id = S.id
WHERE S.value > 100
  
```



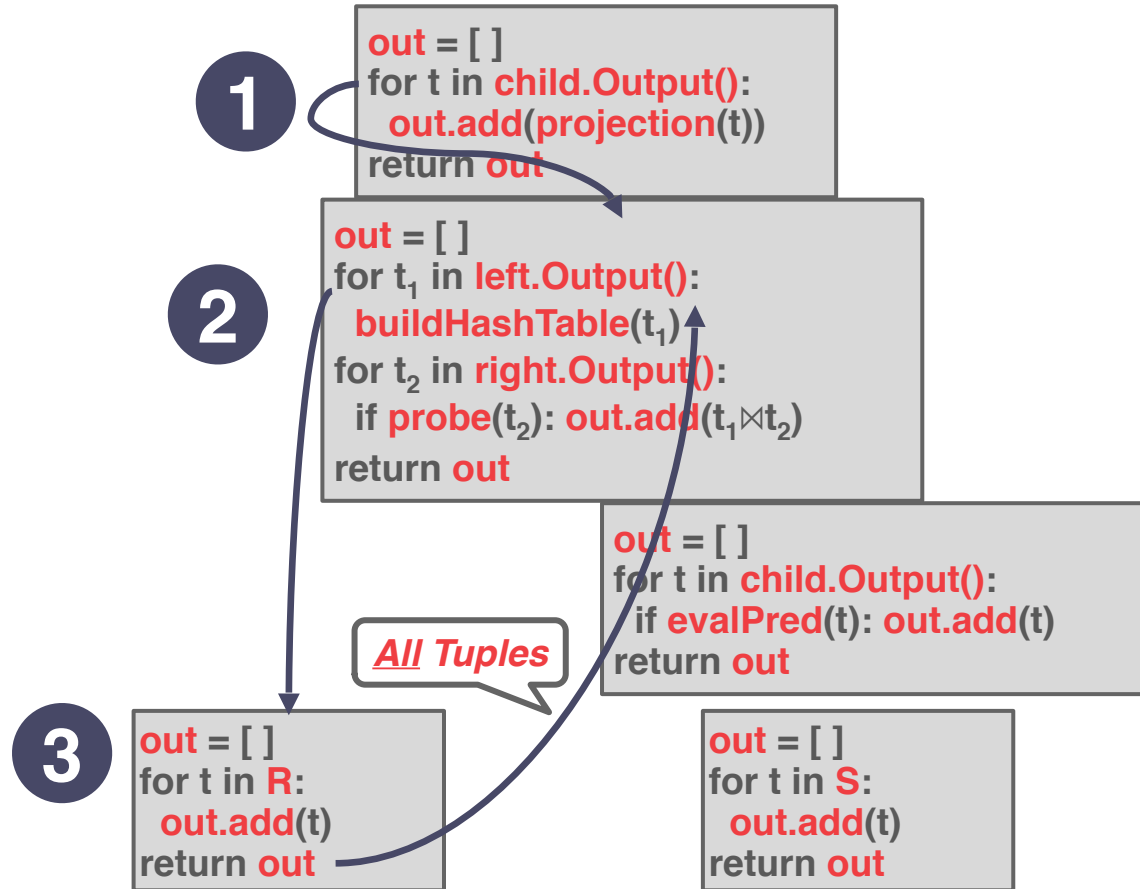
Materialization Model



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



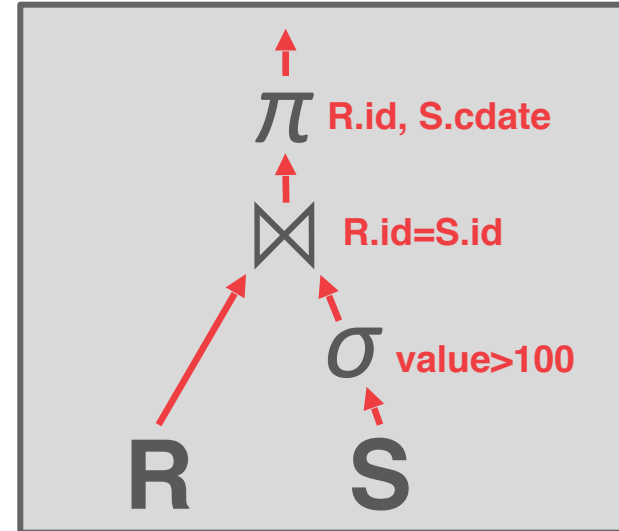
Materialization Model



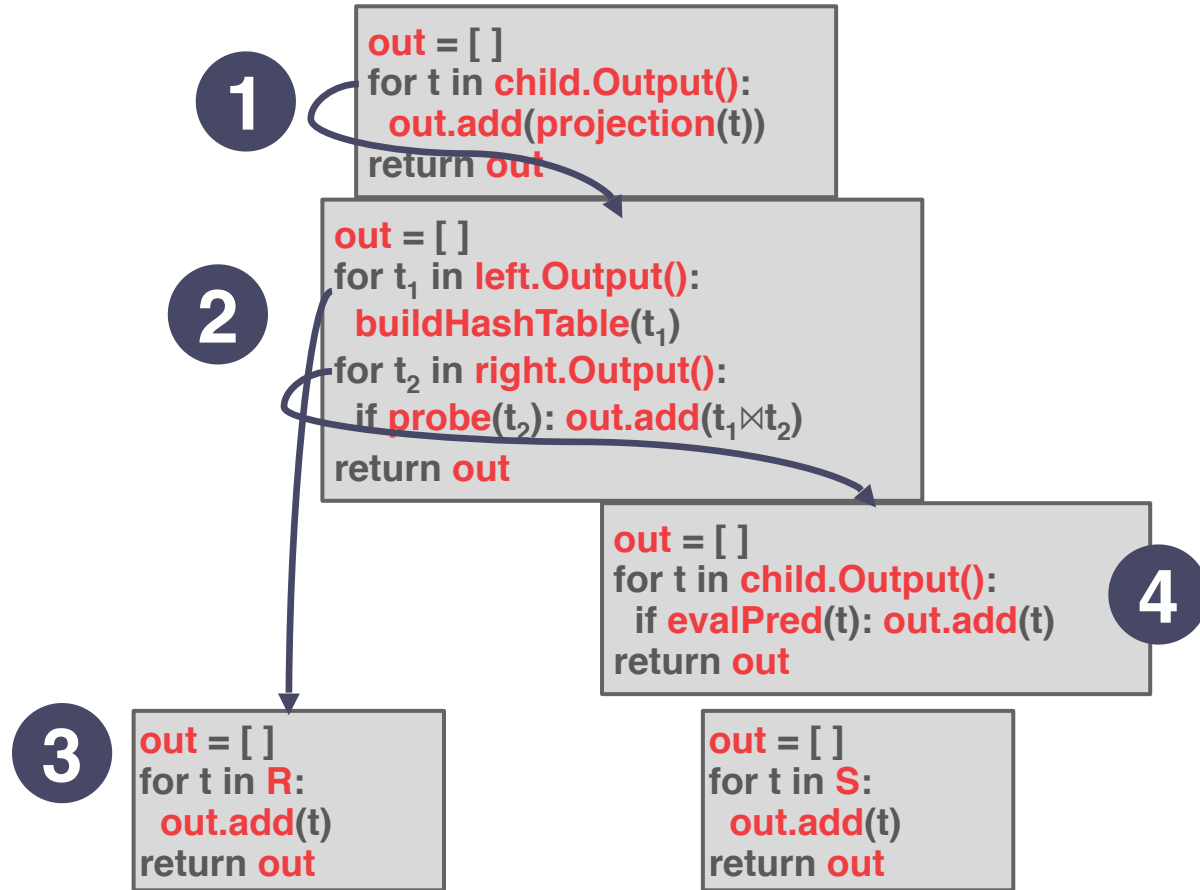
```

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

```



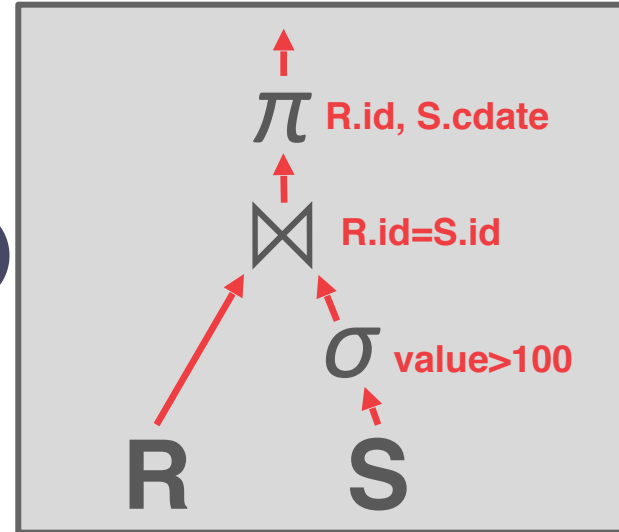
Materialization Model



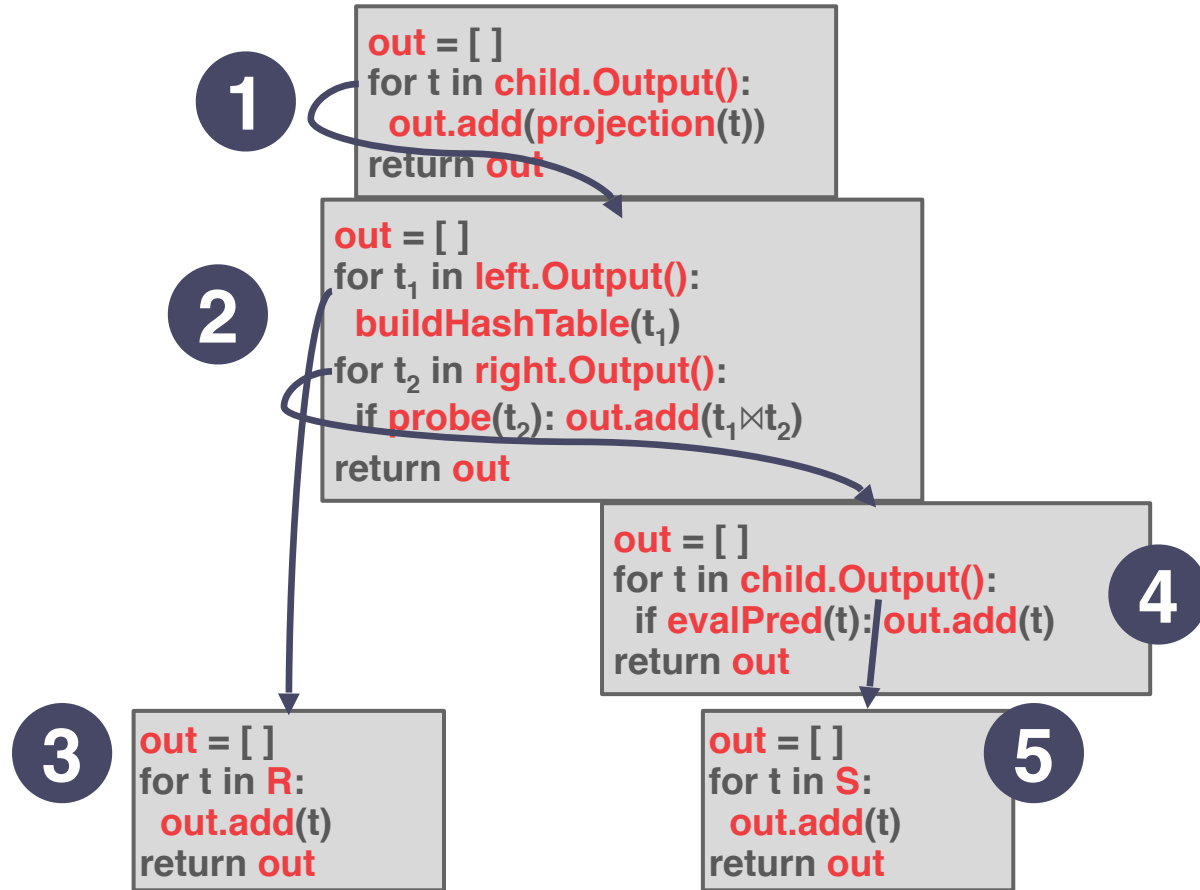
```

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

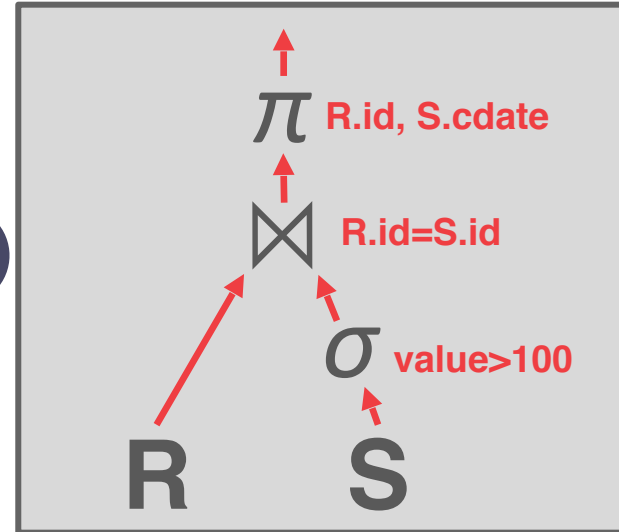
```



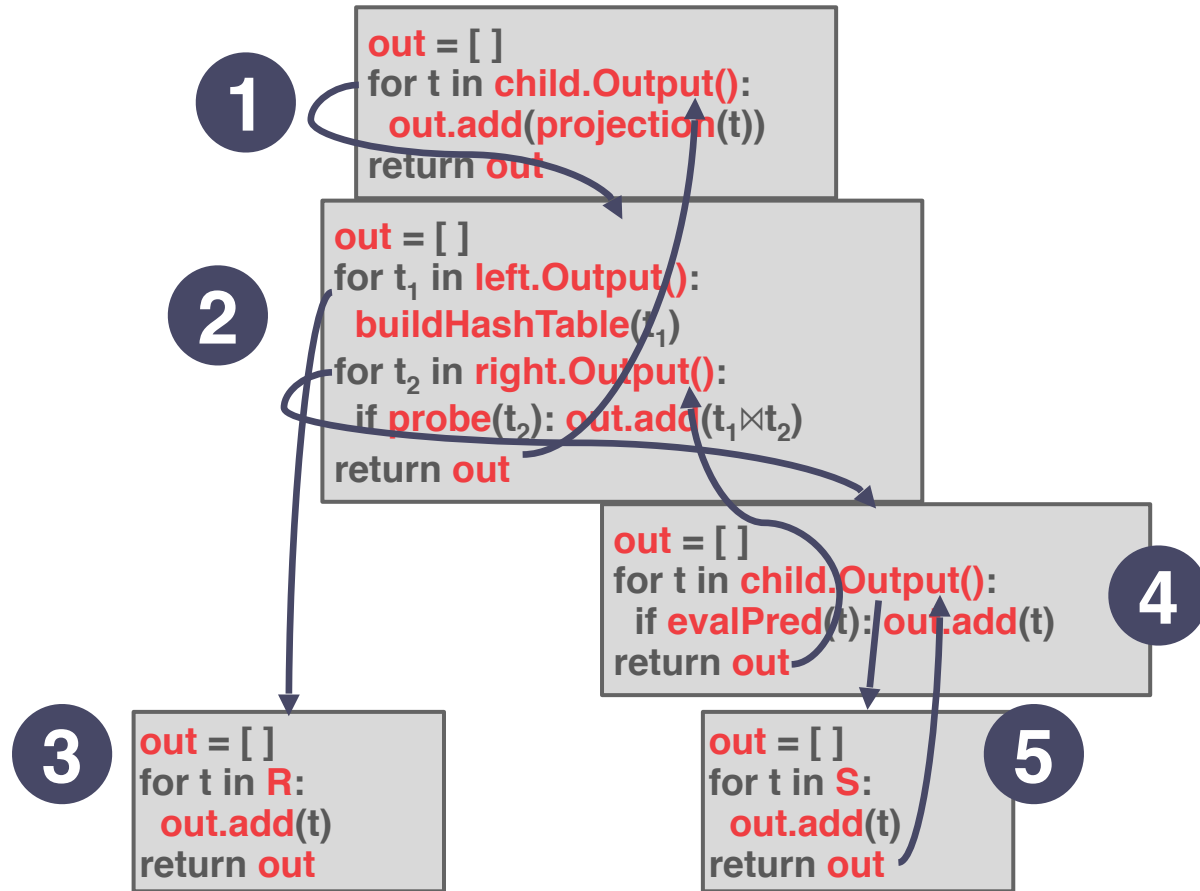
Materialization Model



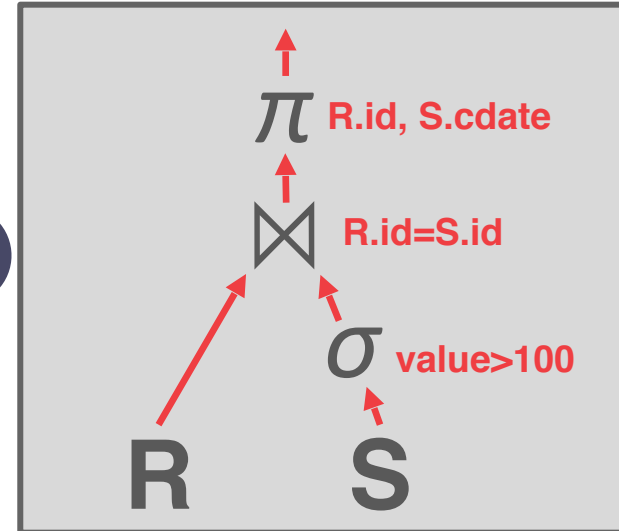
```
SELECT R.id, S.cdate
FROM R JOIN S
ON R.id = S.id
WHERE S.value > 100
```



Materialization Model



```
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)
  
```

```

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)
  
```

```

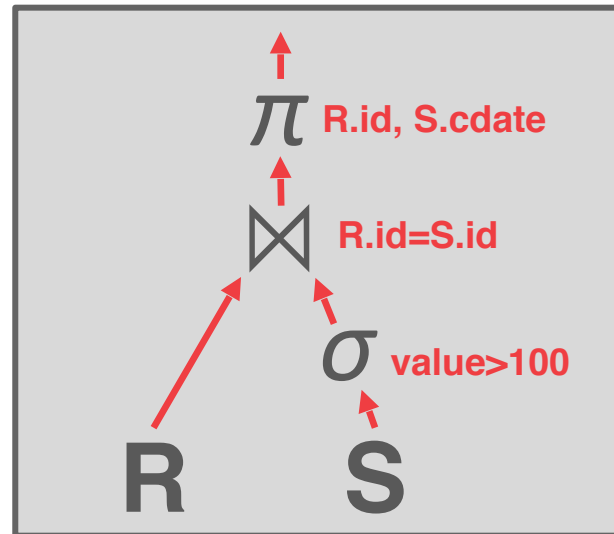
out = [ ]
for t in R:
    out.add(t)
    if |out|>n: emit(out)
  
```

```

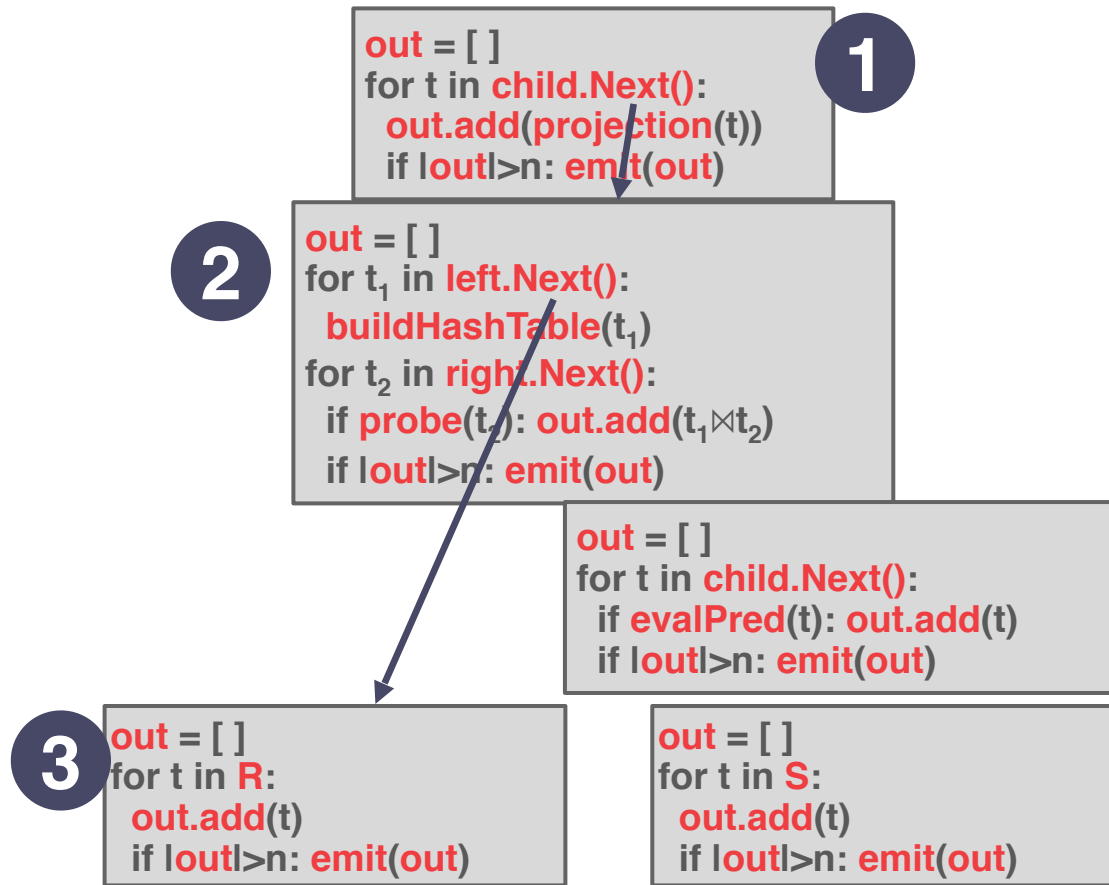
out = [ ]
for t in S:
    out.add(t)
    if |out|>n: emit(out)
  
```

```

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

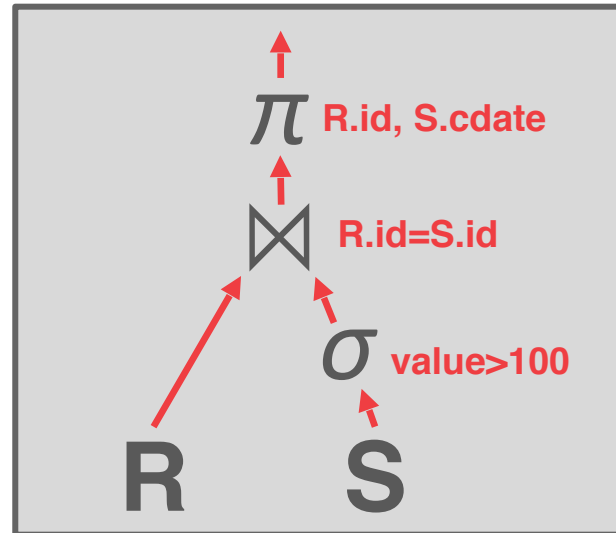


Vectorization Model

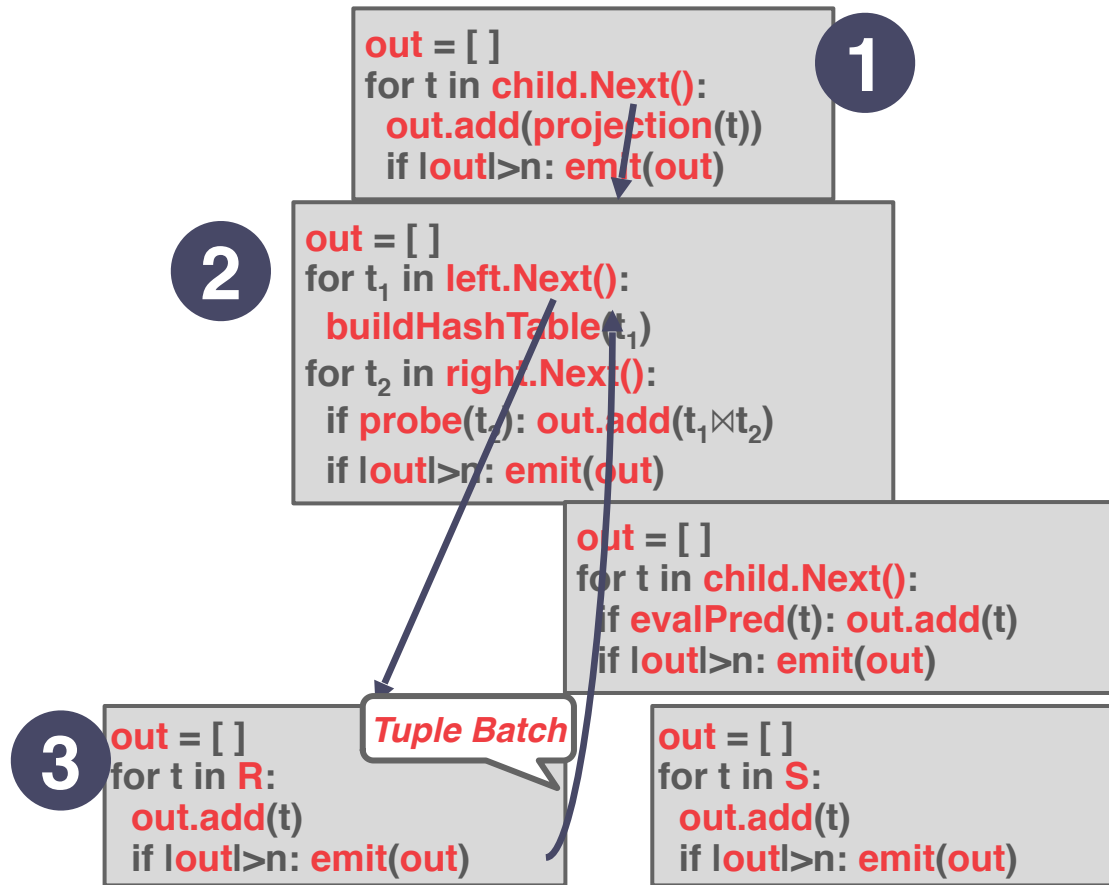


```

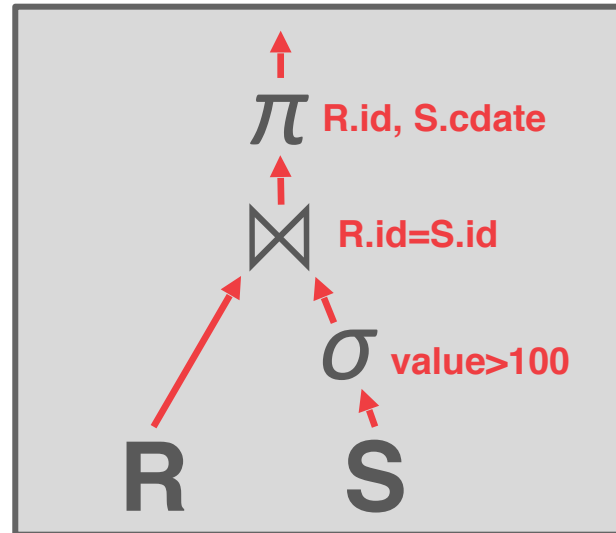
SELECT R.id, S.cdate
FROM R JOIN S
      ON R.id = S.id
WHERE S.value > 100
  
```



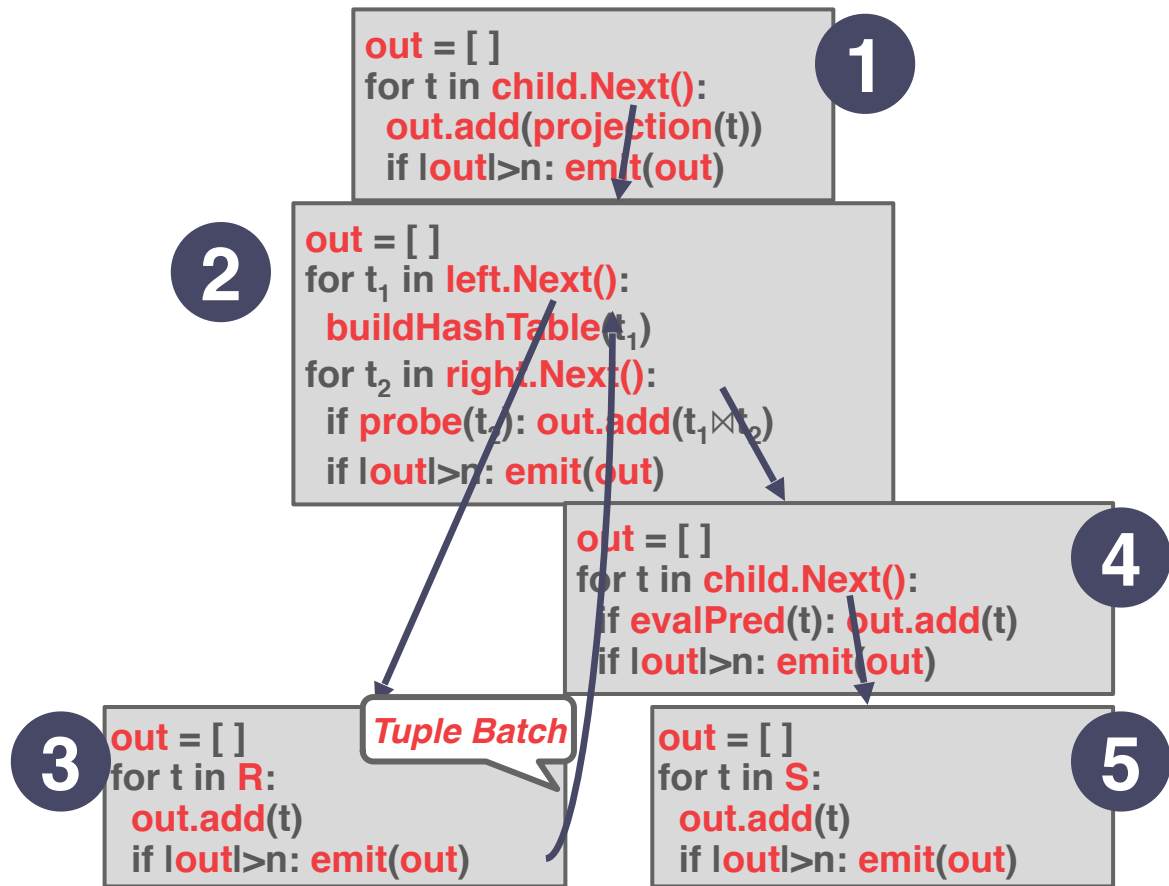
Vectorization Model



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

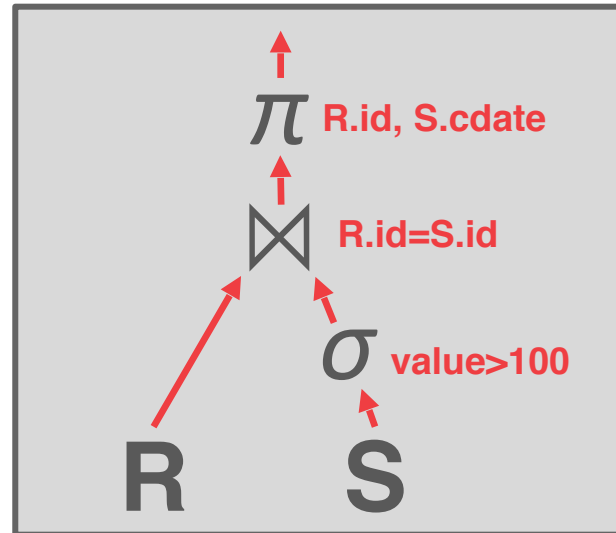


Vectorization Model



```

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 more easily use vectorized (SIMD) instructions to process batches of tuples.



Observation

In the previous examples, the DBMS was starting at the root of the query plan and pulling data up from leaf operators.

This is how most DBMSs implement their execution engine.

Plan Processing Direction

Approach #1: Top-to-Bottom (Pull)

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

Approach #2: Bottom-to-Top (Push)

- Start with leaf nodes and "push" data to their parents.
- We will see this later in [HyPer](#) and [Peloton ROF](#).



Plan Processing Direction

Approach #1:

- Start with the root children.
- Tuples are always

Approach #2:

- Start with leaf node parents.
- We will see this

Move to push-based execution model #1583



Mythardn opened this issue on Apr 8, 2021 · 2 comments



Mythardn commented on Apr 8, 2021 · edited =

collaborator · ...

Currently our execution model operates in a pull-based volcano-like fashion. That means that an operator exposes a `GetChunk` function that fetches a result chunk from the operator. The operator will, in turn, fetch result chunks from its children using this same interface until it reaches a source node (e.g. a base table scan or a parquet file) which can actually emit files, after which execution resumes.

A simple example of such an operator is the projection:

```
void PhysicalProjection::GetChunkInternal(ExecutionContext* context, DataChunk& chunk, PhysicalOperatorState* state) {
    auto state = reinterpret_cast<PhysicalProjectionState*>(state_p);

    // get the next chunk from the child
    children[0] -> GetChunk(context, state->child_chunk, state->child_state.get());
    if (state->child_chunk.size() == 0) {
        return;
    }

    state->executor.PushState(state->child_chunk, chunk);
}
```

This works semi-elegantly and has generally served us well. However, now that we have introduced pipeline parallelism the model is beginning to show cracks. In the pipeline parallelism model, we no longer want to have the behavior of "pulling from the root node". Instead, we want to execute pipelines separately.

The way this is done right now is a semi-hacky solution on top of this model. If we have a pipeline (e.g. a hash table build), we pull from the child node of the hash table using `GetChunk`, and then call `Chunk` with the result of this. Partitioning is done by writing partition information to the thread-local `ExecutionContext` object, and using that in the source node to determine the desired partitioning. For example, here is how this is done in the TableScan:

```
// table scan
auto task = context.task;
// check if there is any parallel state to fetch
state.parallel_state = nullptr;
auto task_info = task.task_info.find(this);
```

Plan Processing Direction

Approach #1: Top-to-Bottom (Pull)

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

Approach #2: Bottom-to-Top (Push)

- Start with leaf nodes and "push" data to their parents.
- We will see this later in [HyPer](#) and [Peloton ROF](#).



Plan Processing Direction

Approach #1: Top-to-Bottom (Pull)

- Easy to control output via **LIMIT**.
- Parent operator blocks until its child returns with a tuple.
- Additional overhead because operators' **next** functions are implemented as virtual functions.
- Branching costs on each **next** invocation.

Approach #2: Bottom-to-Top (Push)

- Allows for tighter control of caches/registers in pipelines.
- Difficult to control output via LIMIT.
- Difficult to implement Sort-Merge Join.



Today's Agenda

~~Processing Models~~

Parallel Execution

Scheduling

Parallel Execution

The DBMS executes multiple tasks simultaneously to improve hardware utilization.

→ Active tasks do not need to belong to the same query.

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.

OLAP queries have parallelizable and non-parallelizable phases. The goal is to always keep all cores active.

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)

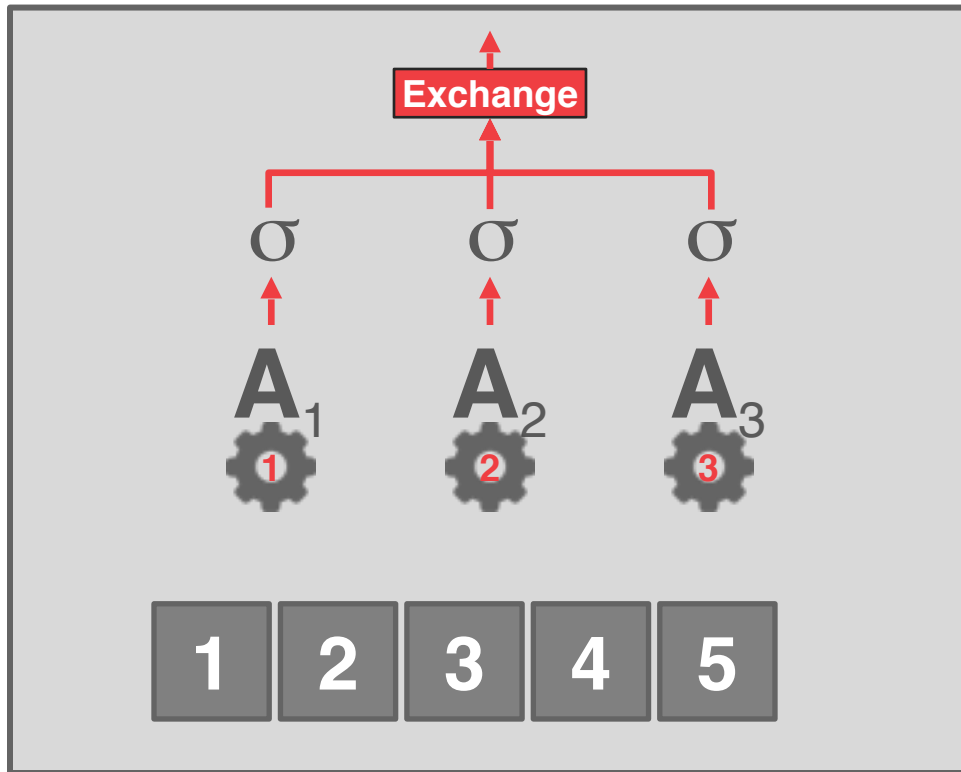
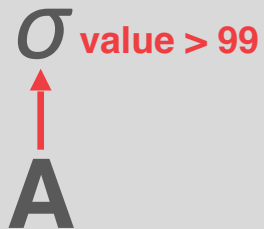
- Decompose operators into independent fragments that perform the same function on different subsets of data.

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

- Postgres calls this "gather"

Intra-Operator Parallelism

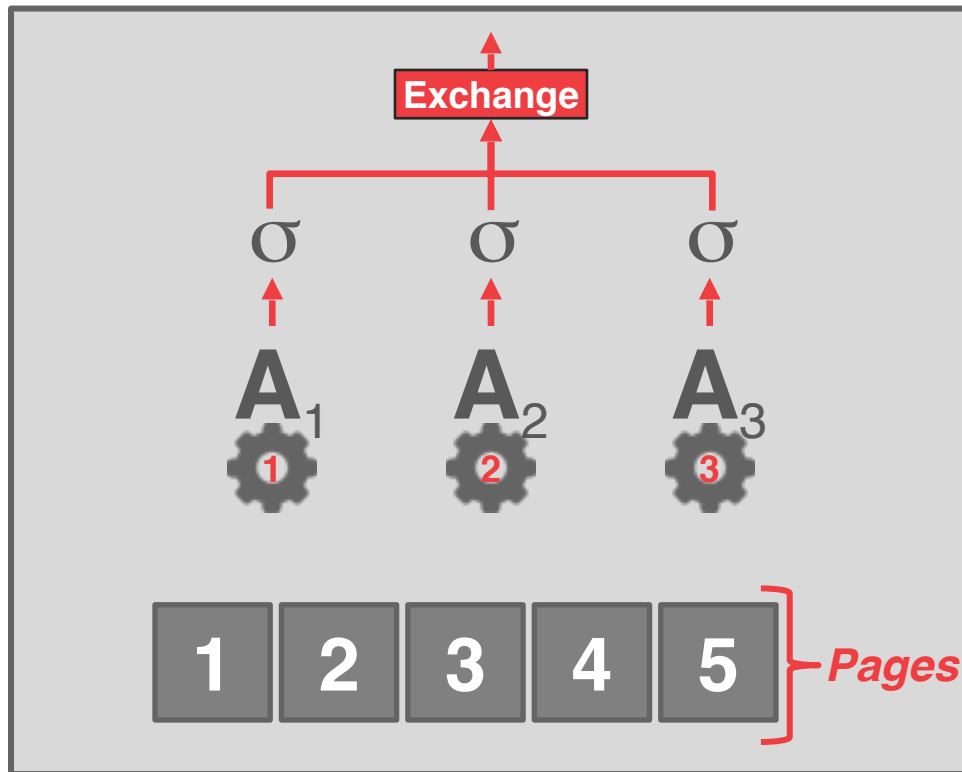
SELECT * FROM A
WHERE A.val > 99



Intra-Operator Parallelism

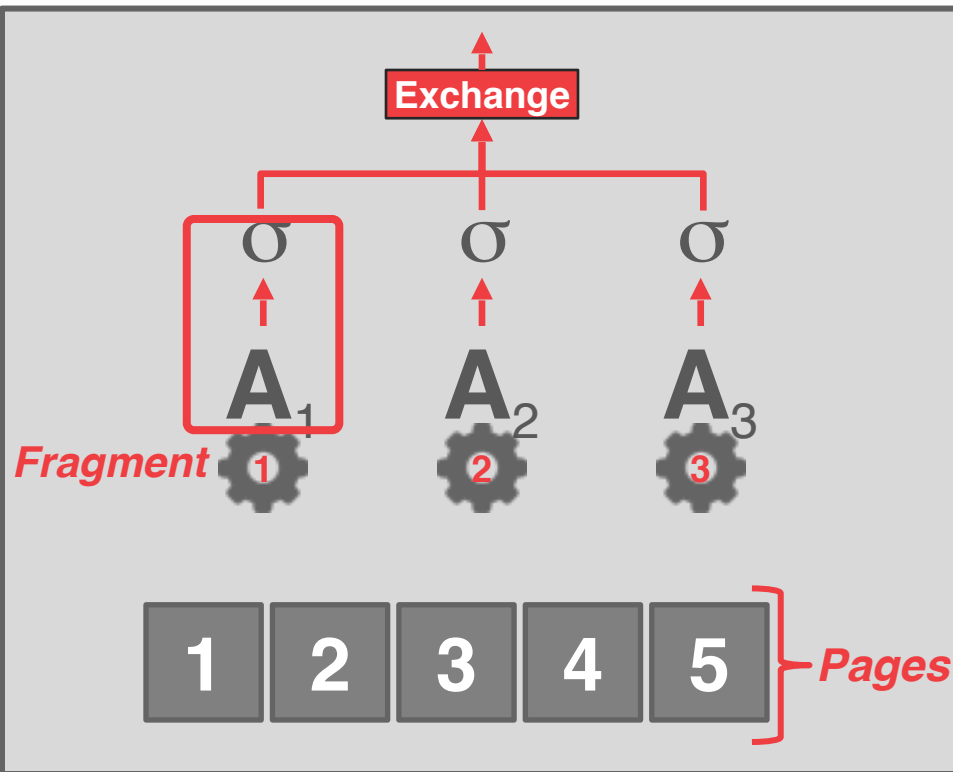
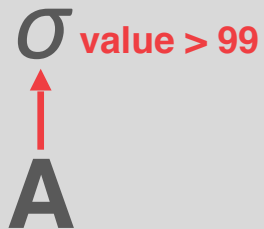
SELECT * FROM A
WHERE A.val > 99

$\sigma_{\text{value} > 99}$
↑
A



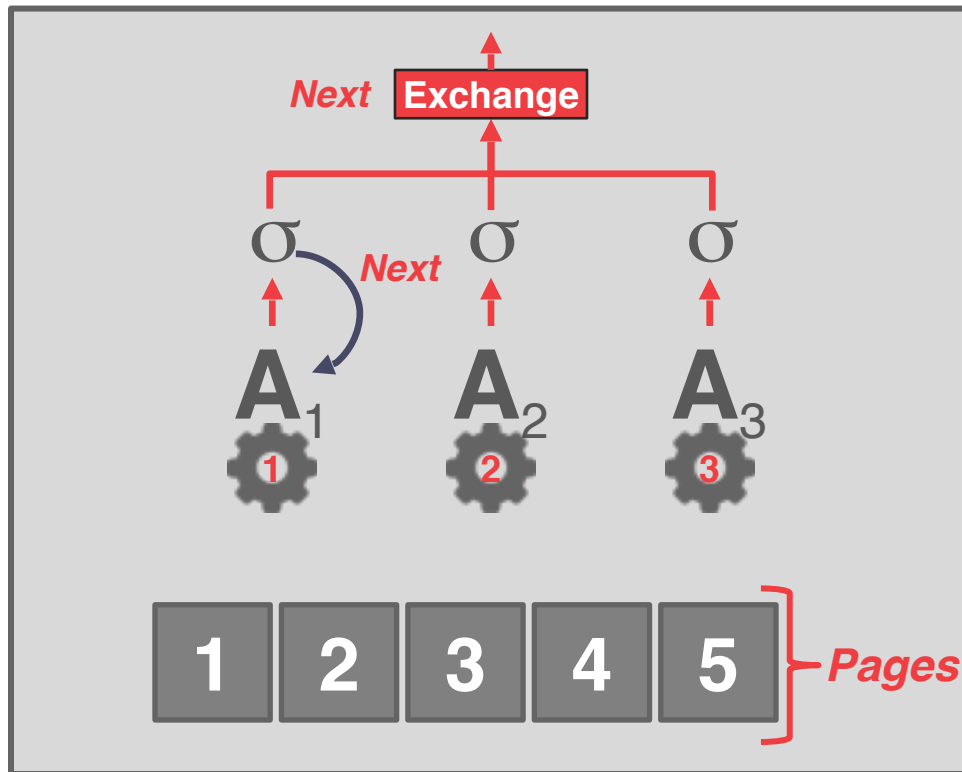
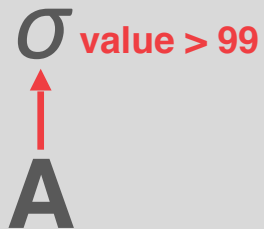
Intra-Operator Parallelism

SELECT * FROM A
WHERE A.val > 99



Intra-Operator Parallelism

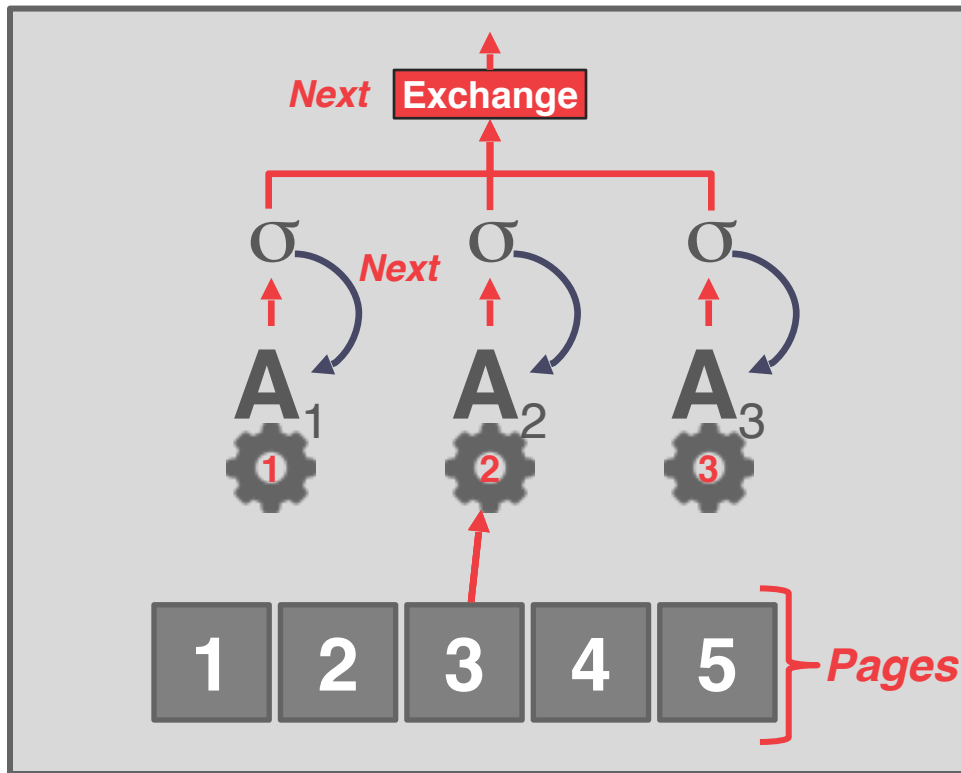
SELECT * FROM A
WHERE A.val > 99



Intra-Operator Parallelism

SELECT * FROM A
WHERE A.val > 99

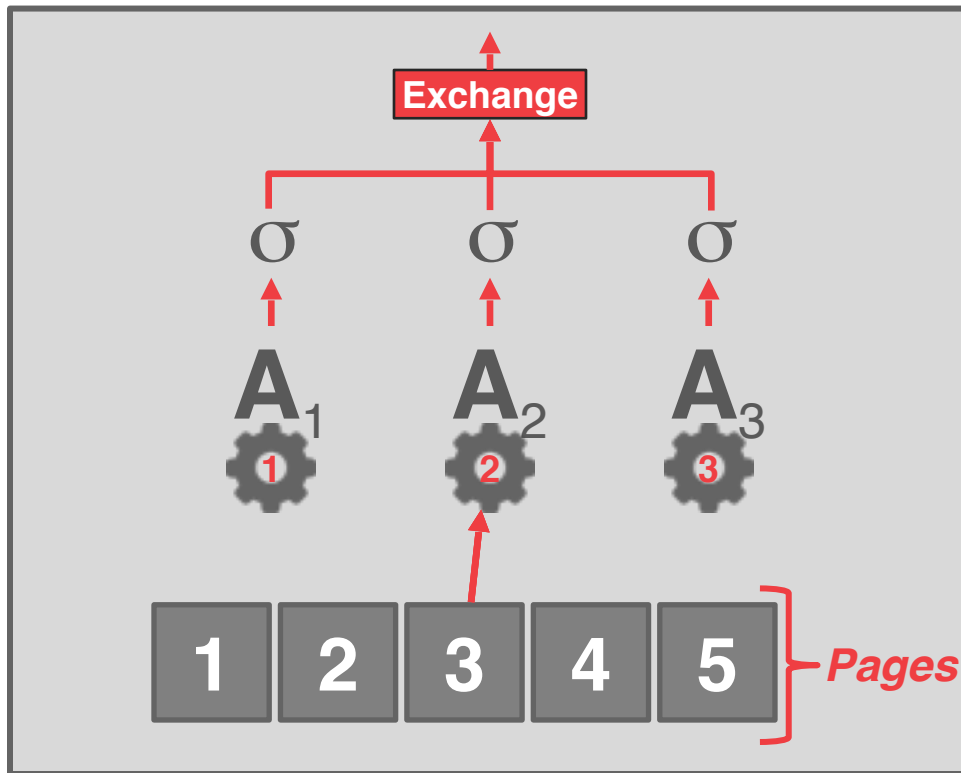
$\sigma_{\text{value} > 99}$
↑
A



Intra-Operator Parallelism

SELECT * FROM A
WHERE A.val > 99

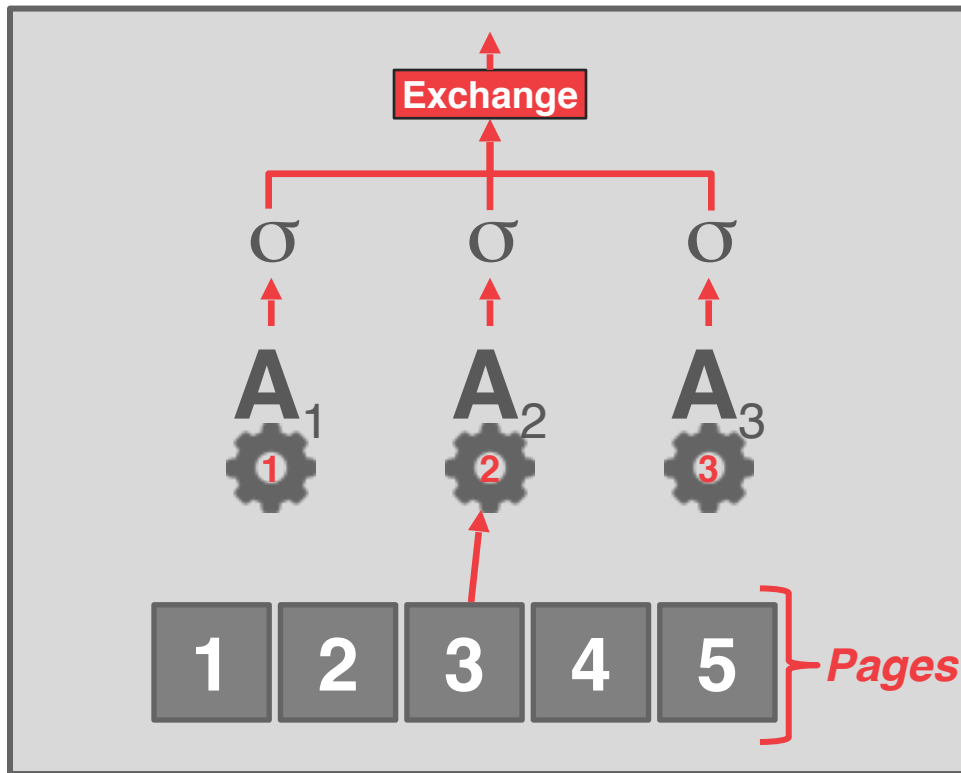
$\sigma_{\text{value} > 99}$
↑
A



Intra-Operator Parallelism

SELECT * FROM A
WHERE A.val > 99

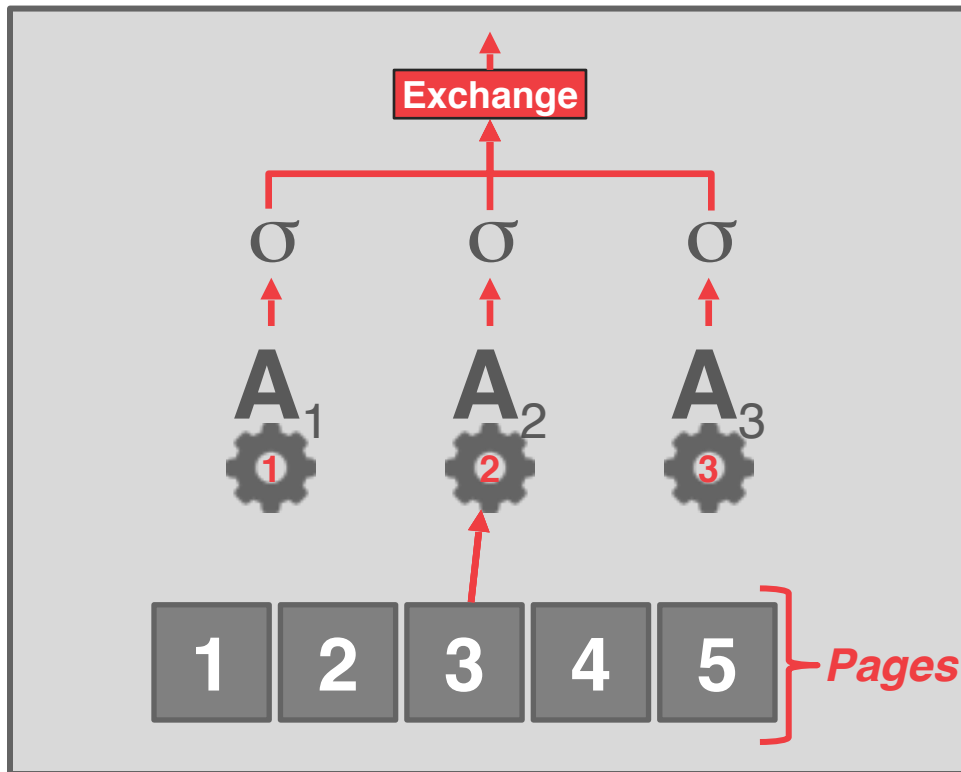
$\sigma_{\text{value} > 99}$
↑
A



Intra-Operator Parallelism

SELECT * FROM A
WHERE A.val > 99

$\sigma_{\text{value} > 99}$
↑
A

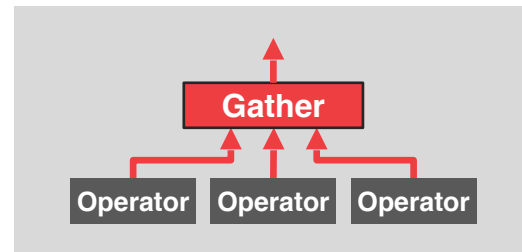


Exchange Operator

Exchange Operator

Exchange Type #1 – Gather

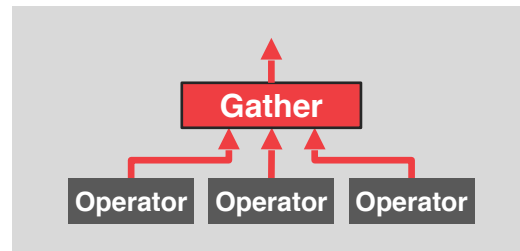
→ Combine the results from multiple workers into a single output stream.



Exchange Operator

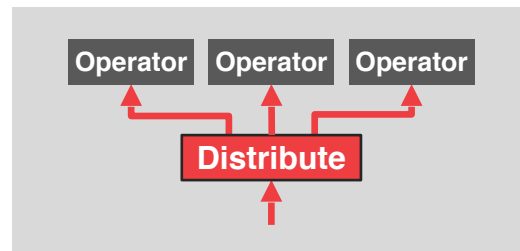
Exchange Type #1 – Gather

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



Exchange Type #2 – Distribute

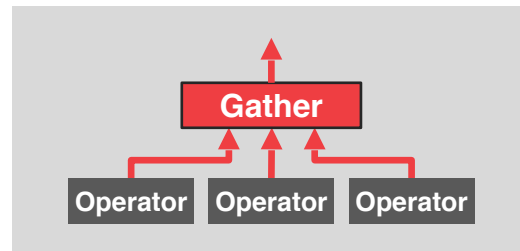
→ Split a single input stream into multiple output streams.



Exchange Operator

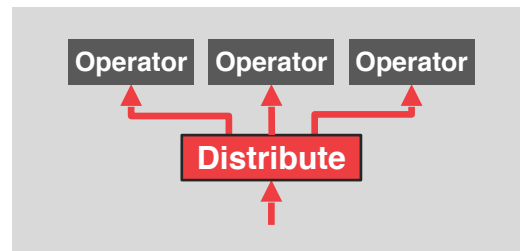
Exchange Type #1 – Gather

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



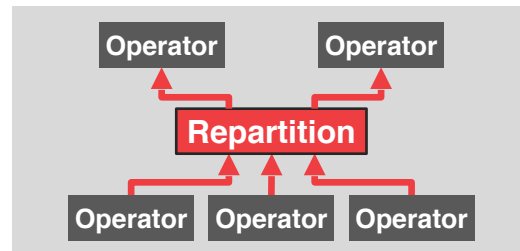
Exchange Type #2 – Distribute

→ Split a single input stream into multiple output streams.



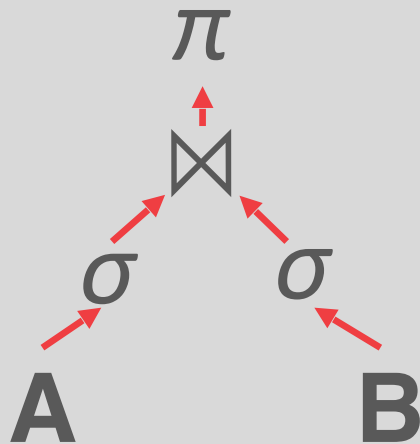
Exchange Type #3 – Repartition

→ Shuffle multiple input streams across multiple output streams.



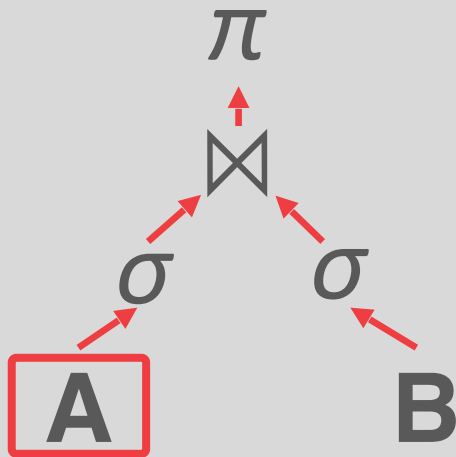
Intra-Operator Parallelism

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



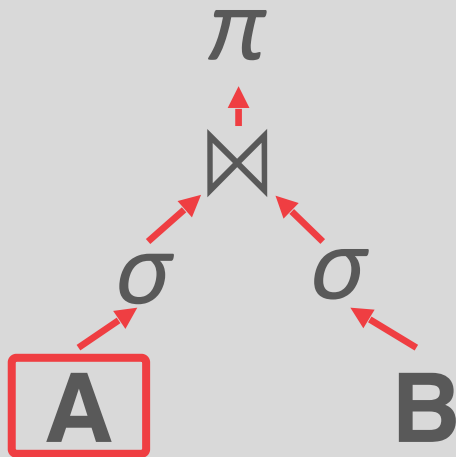
Intra-Operator Parallelism

```
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

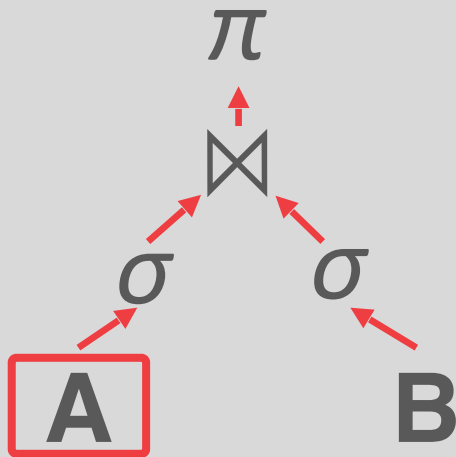
```
SELECT A.id, B.value  
FROM A JOIN B  
ON A.id = B.id  
WHERE A.value < 99  
AND B.value > 100
```



A_1 A_2 A_3

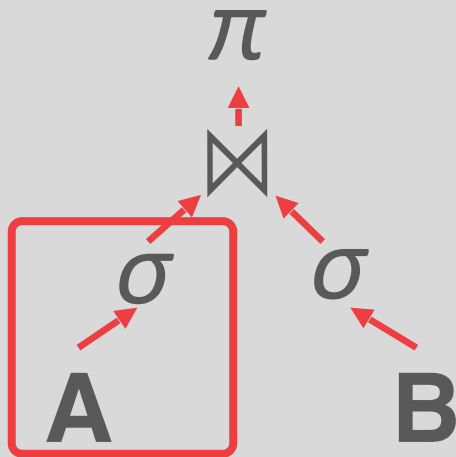
Intra-Operator Parallelism

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



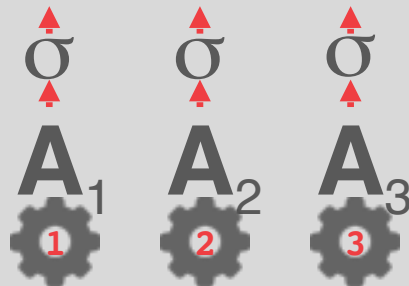
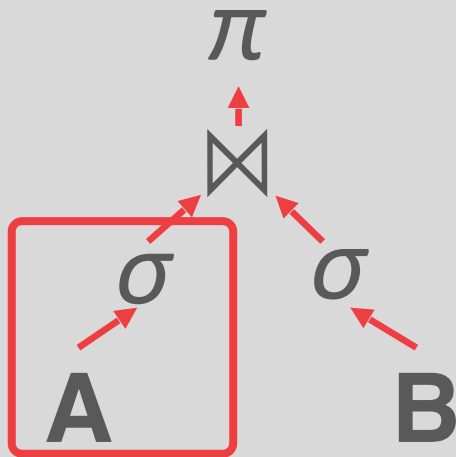
Intra-Operator Parallelism

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



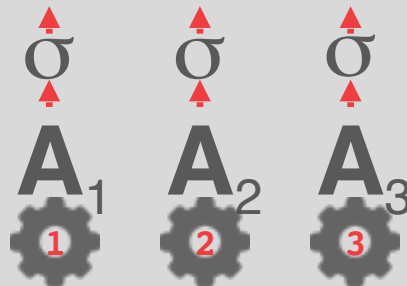
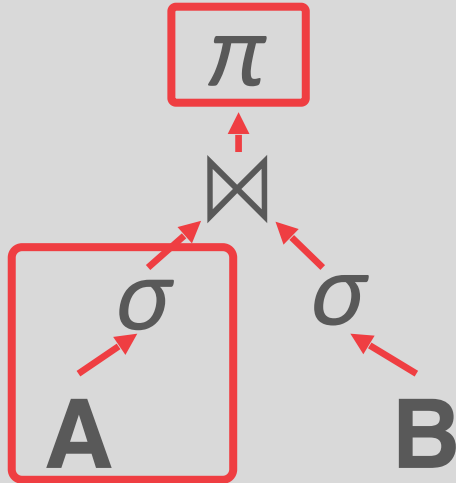
Intra-Operator Parallelism

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



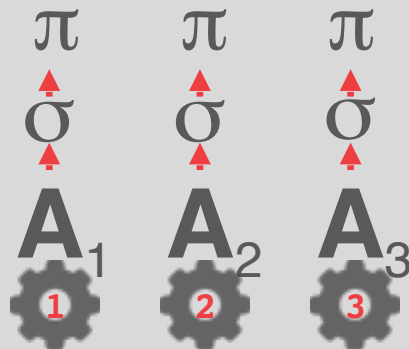
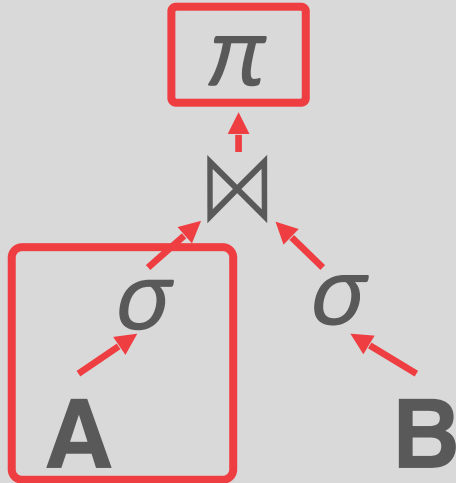
Intra-Operator Parallelism

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



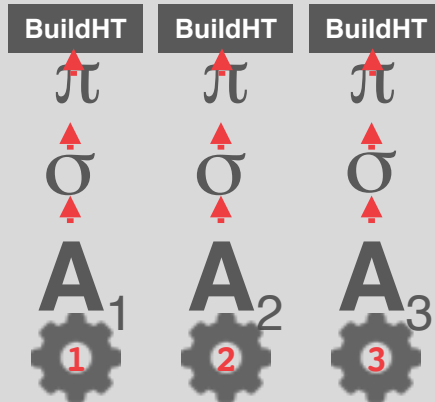
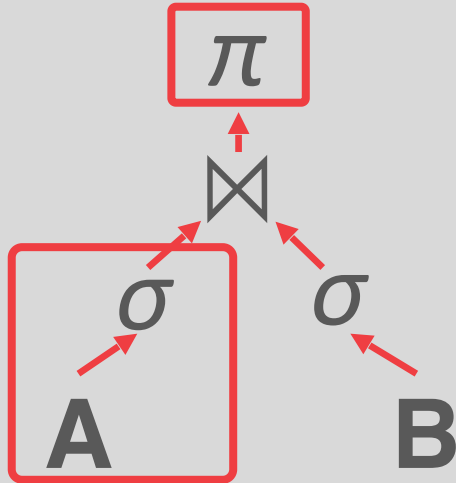
Intra-Operator Parallelism

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



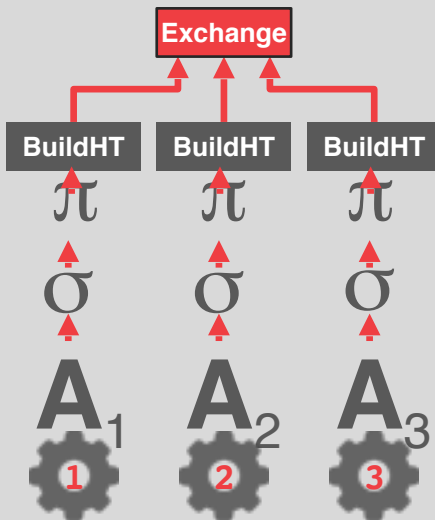
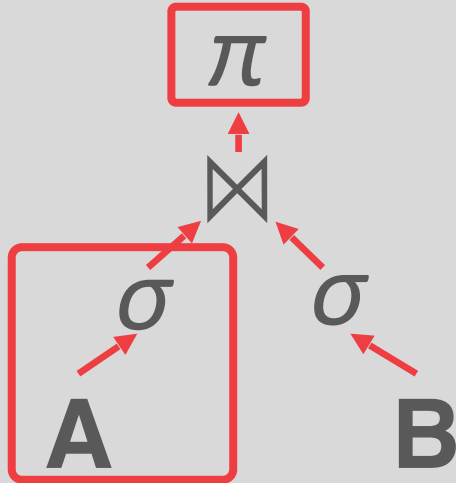
Intra-Operator Parallelism

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



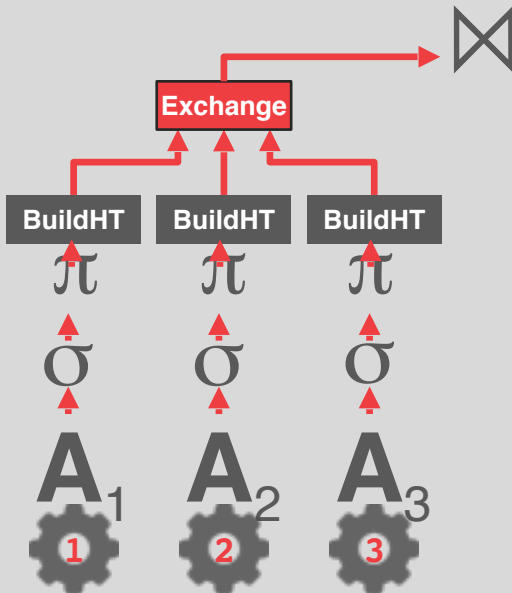
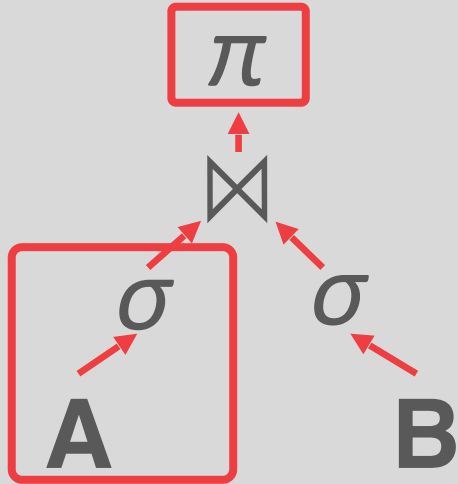
Intra-Operator Parallelism

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



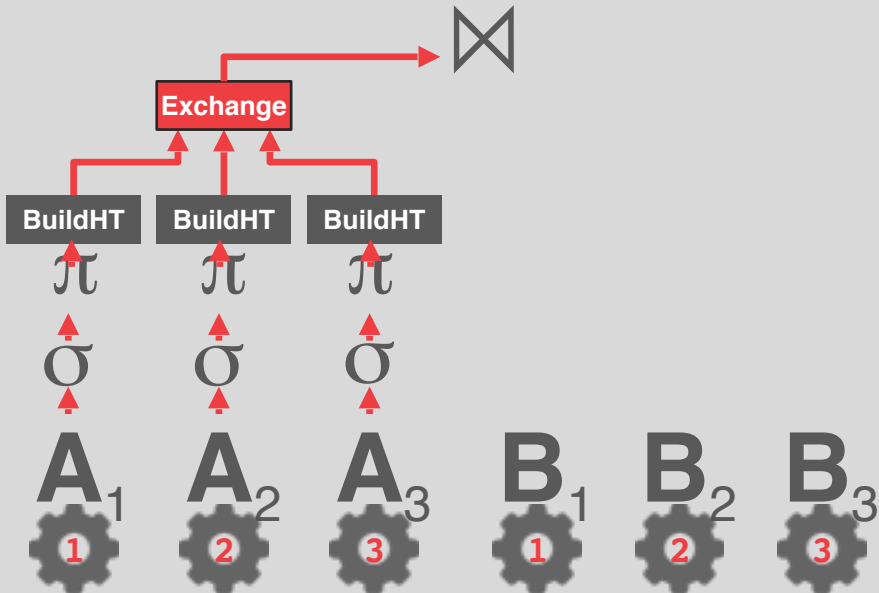
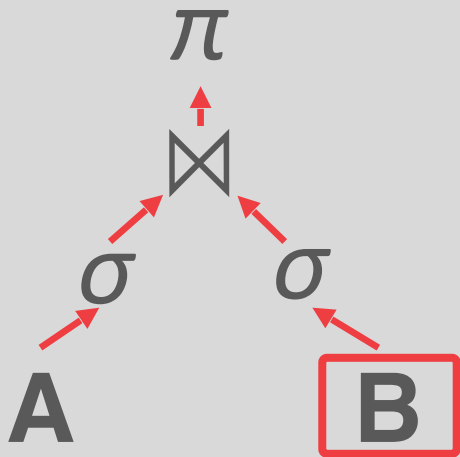
Intra-Operator Parallelism

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



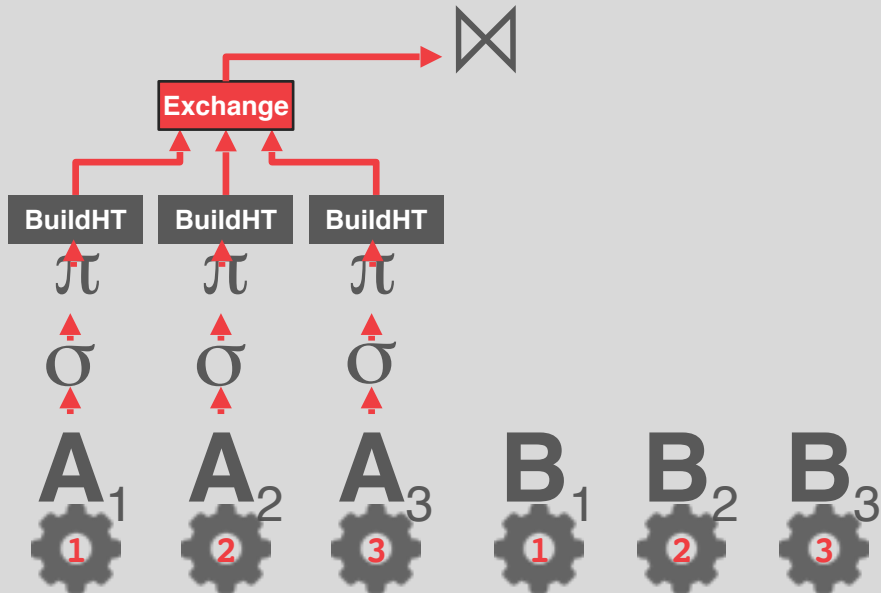
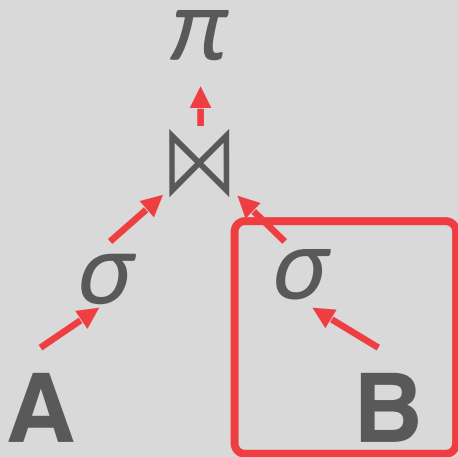
Intra-Operator Parallelism

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



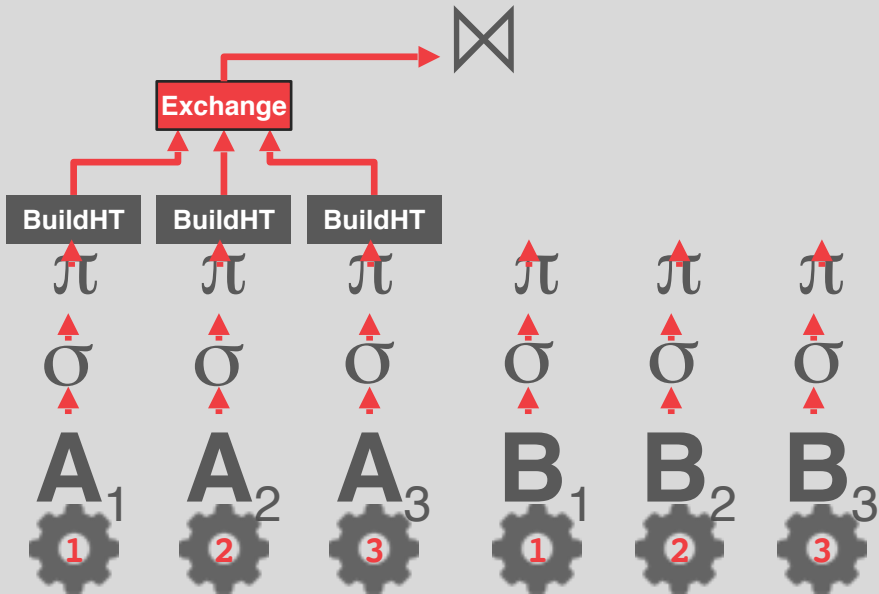
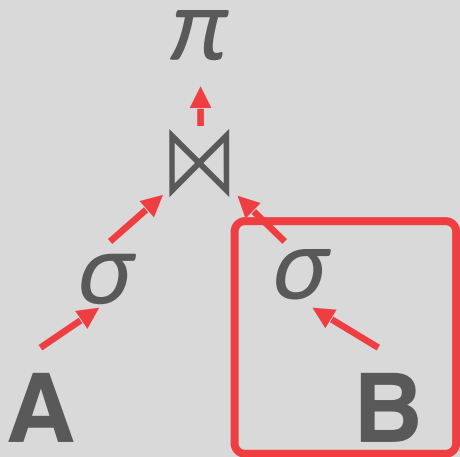
Intra-Operator Parallelism

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



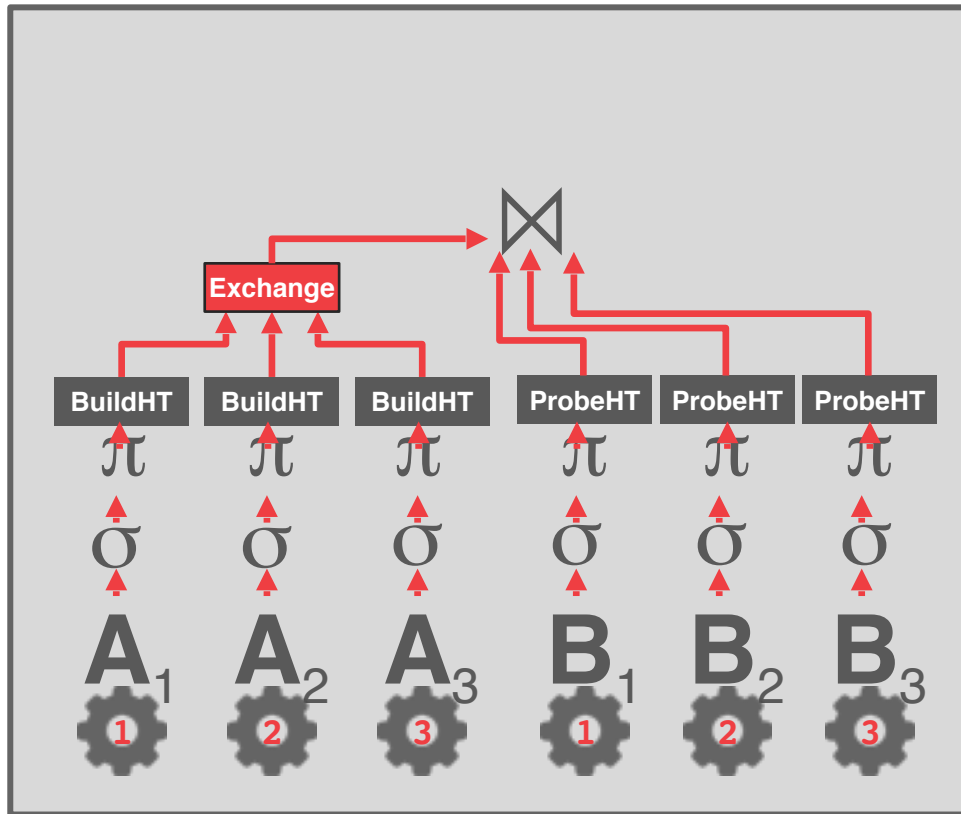
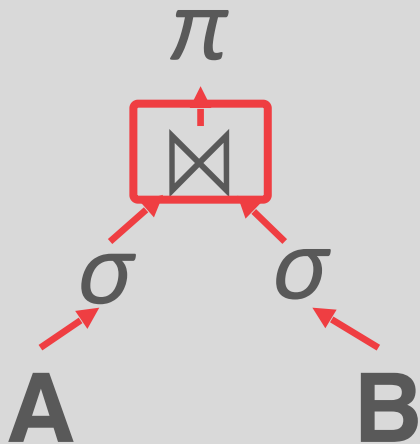
Intra-Operator Parallelism

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



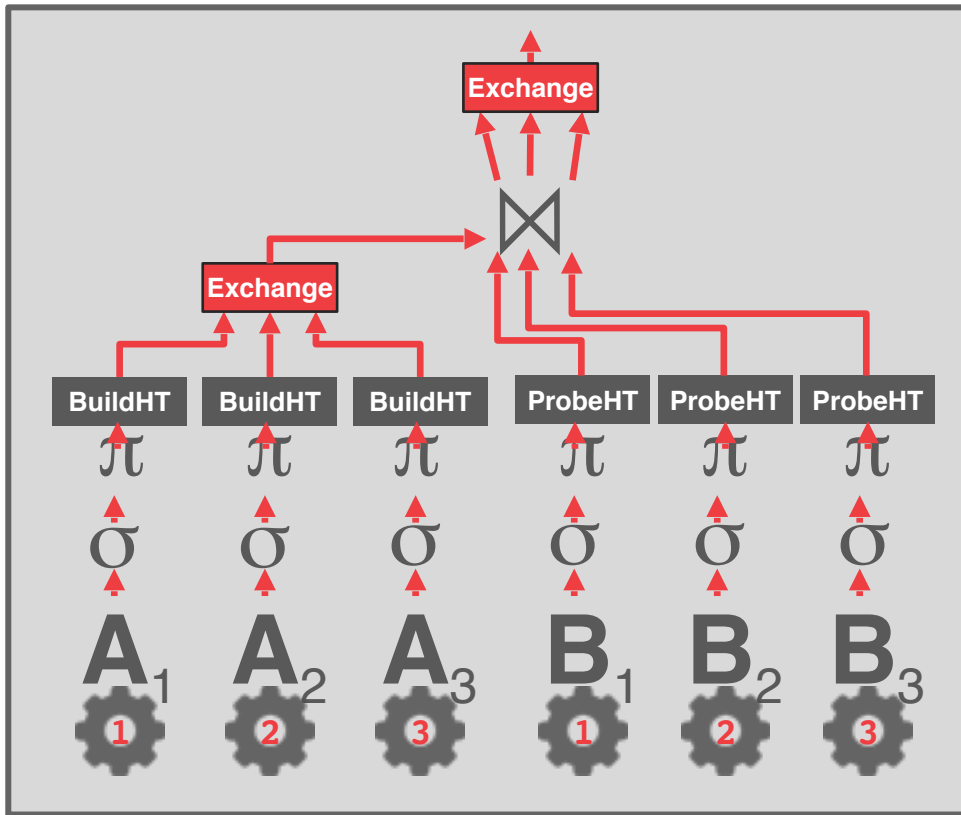
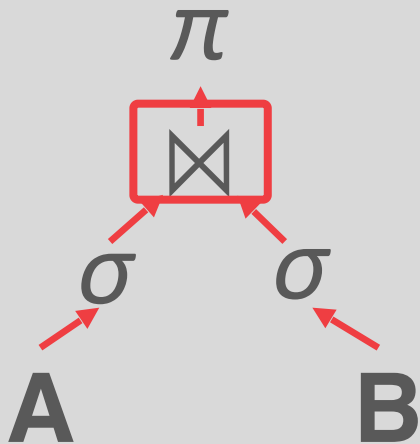
Intra-Operator Parallelism

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



Inter-Operator Parallelism

Approach #2: Inter-Operator (Vertical)

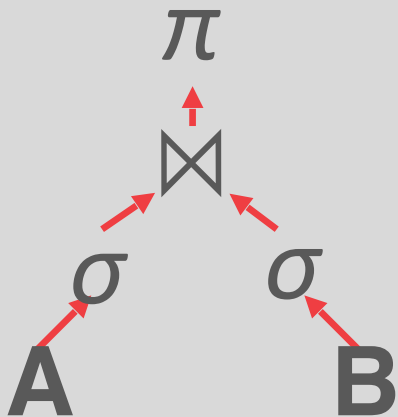
- Operations are overlapped in order to pipeline data from one stage to the next without materialization.
- Workers execute operators from different segments of a query plan at the same time.
- More common in streaming systems (continuous queries)

Also called pipeline parallelism.



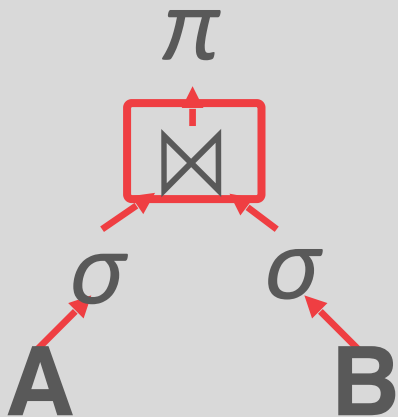
Inter-Operator Parallelism

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



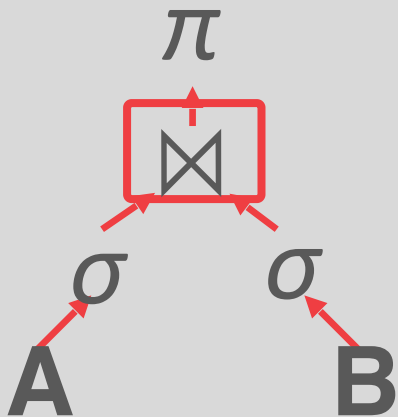
Inter-Operator Parallelism

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



Inter-Operator Parallelism

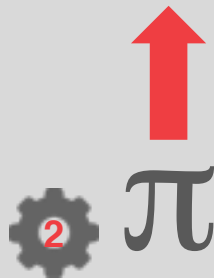
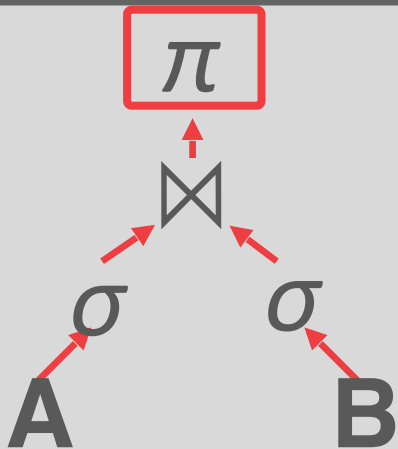
```
SELECT A.id, B.value  
FROM A JOIN B  
ON A.id = B.id  
WHERE A.value < 99  
AND B.value > 100
```



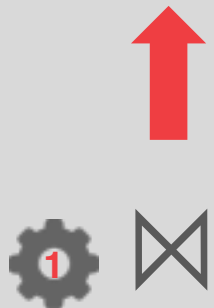
```
for  $r_1 \in$  outer:  
  for  $r_2 \in$  inner:  
    emit( $r_1 \bowtie r_2$ )
```

Inter-Operator Parallelism

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



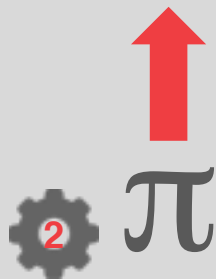
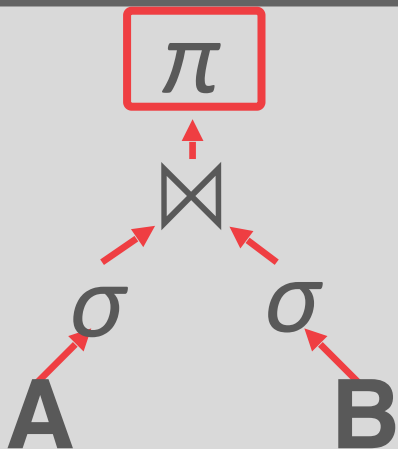
for $r \in$ incoming:
emit(πr)



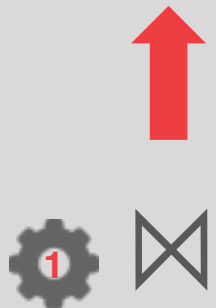
for $r_1 \in$ outer:
for $r_2 \in$ inner:
emit($r_1 \bowtie r_2$)

Inter-Operator Parallelism

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



for $r \in \text{incoming}$:
emit(πr)



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

Today's Agenda

~~Processing Models~~

~~Parallel Execution~~

Scheduling

Scheduling

For each query plan, the DBMS must decide 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 ***always*** knows more than the OS.

Scheduling Goals

Goal #1: Throughput

→ Maximize the # of completed queries.

Scheduling Goals

Goal #1: Throughput

→ Maximize the # of completed queries.

Goal #2: Fairness

→ Ensure that no query is starved for resources.

Scheduling Goals

Goal #1: Throughput

→ Maximize the # of completed queries.

Goal #2: Fairness

→ Ensure that no query is starved for resources.

Goal #3: Query Responsiveness

→ Minimize tail latencies (especially for short queries).

Scheduling Goals

Goal #1: Throughput

→ Maximize the # of completed queries.

Goal #2: Fairness

→ Ensure that no query is starved for resources.

Goal #3: Query Responsiveness

→ Minimize tail latencies (especially for short queries).

Goal #4: Low Overhead

→ Workers should spend most of their time executing tasks, not figuring out what task to run next.

Process Model

A DBMS's process model defines how the system is architected to support concurrent requests from a multi-user application.

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

We will assume that the DBMS is multi-threaded.



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 #2: Pull

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

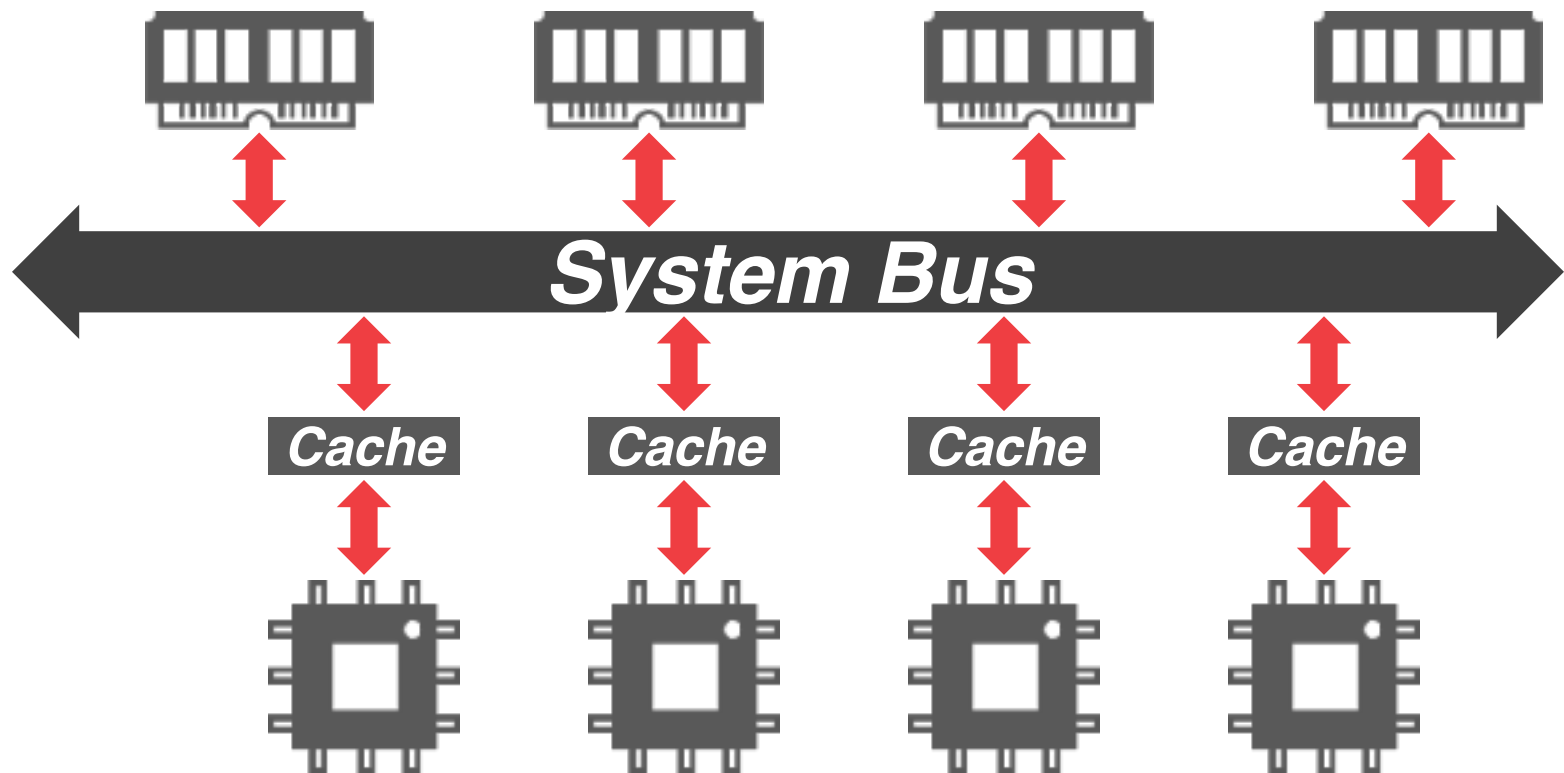
Observation

Regardless of what worker allocation or task assignment policy the DBMS uses, it's important that workers operate on local data.

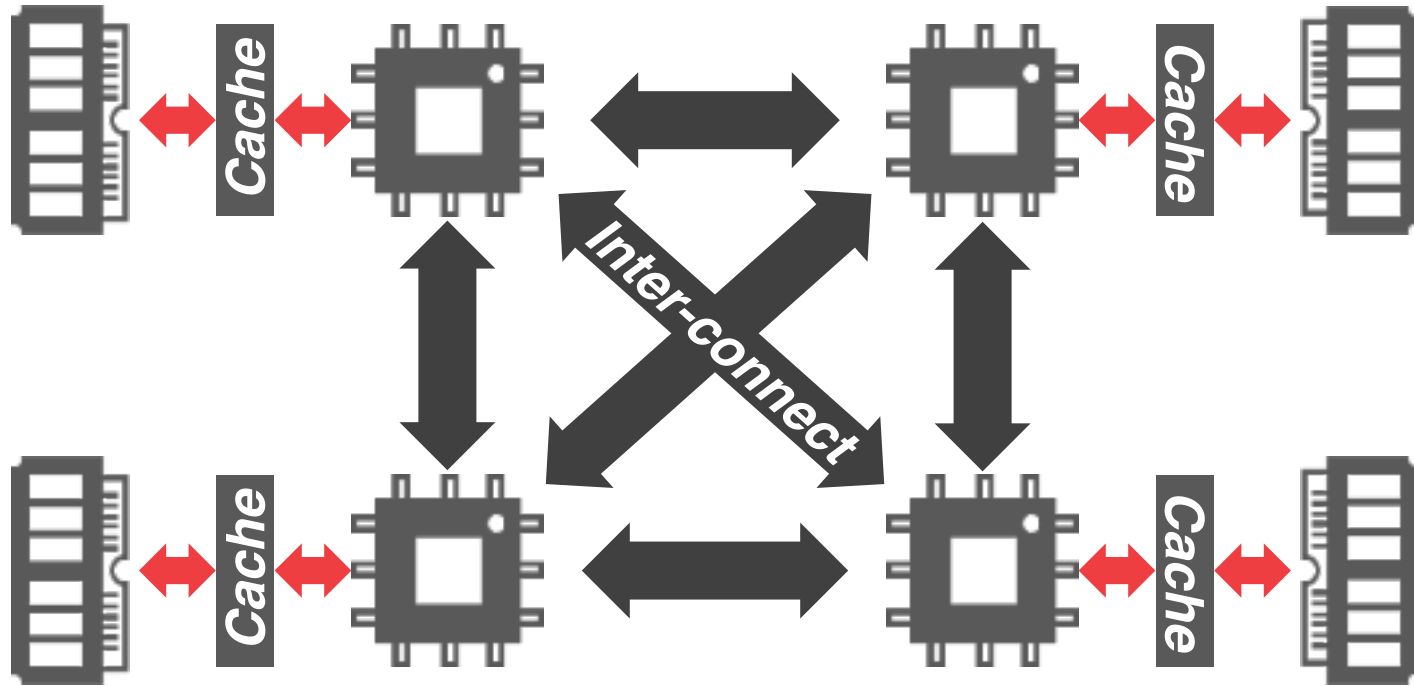
The DBMS's scheduler must be aware of its hardware memory layout.

→ Uniform vs. Non-Uniform Memory Access

Uniform Memory Access



Non-Uniform Memory Access



Data Placement

The DBMS can partition memory for a database and assign each partition to a CPU.

By controlling and tracking the location of partitions, it can schedule operators to execute on workers at the closest CPU core.

See Linux's [move_pages](#) and [numactl](#)

Memory Allocation

What happens when the DBMS calls **malloc**?

→ Assume that the allocator doesn't already have a chunk of memory that it can give out.

Memory Allocation

What happens when the DBMS calls **malloc**?

→ Assume that the allocator doesn't already have a chunk of memory that it can give out.

Almost nothing:

Memory Allocation

What happens when the DBMS calls **malloc**?

→ Assume that the allocator doesn't already have a chunk of memory that it can give out.

Almost nothing:

- The allocator will extend the process' data segment.
- But this new virtual memory is not immediately backed by physical memory.
- The OS only allocates physical memory when there is a page fault on access.

Memory Allocation

What happens when the DBMS calls **malloc**?

→ Assume that the allocator doesn't already have a chunk of memory that it can give out.

Almost nothing:

- The allocator will extend the process' data segment.
- But this new virtual memory is not immediately backed by physical memory.
- The OS only allocates physical memory when there is a page fault on access.

Now after a page fault, where does the OS allocate physical memory in a NUMA system?

Memory Allocation Location

Approach #1: Interleaving

→ Distribute allocated memory uniformly across CPUs.

Approach #2: First-Touch

→ At the CPU of the thread that accessed the memory location that caused the page fault.

The OS can try to move memory to another NUMA region from observed access patterns.

Partitioning vs. Placement

A **partitioning** scheme is used to split the database based on some policy.

- Round-robin
- Attribute Ranges
- Hashing
- Partial/Full Replication

A **placement** scheme then tells the DBMS where to put those partitions.

- Round-robin
- Interleave across cores

Observation

We have the following so far:

- Task Assignment Model
- Data Placement Policy

But how do we decide how to create a set of tasks from a logical query plan?

- This is relatively easy for OLTP queries.
- Much harder for OLAP queries...

Static Scheduling

The DBMS decides how many threads to use to execute the query during planning.

It does **not** change while the query executes.

- The easiest approach is to just use the same # of tasks as the # of cores.
- Can still assign tasks to threads based on data location to maximize local data processing.

Morsel-Driven Scheduling

Dynamic scheduling of tasks that operate over horizontal partitions called "morsels" distributed across cores.

- One worker per core.
- One morsel per task.
- Pull-based task assignment.
- Round-robin data placement.

Supports parallel, NUMA-aware operator implementations.

HyPer – Architecture

No separate dispatcher thread.

The workers perform cooperative scheduling for each query plan using a single task queue.

- Each worker tries to select tasks that will execute on morsels that are local to it.
- If there are no local tasks, then the worker just pulls the next task from the global work queue.

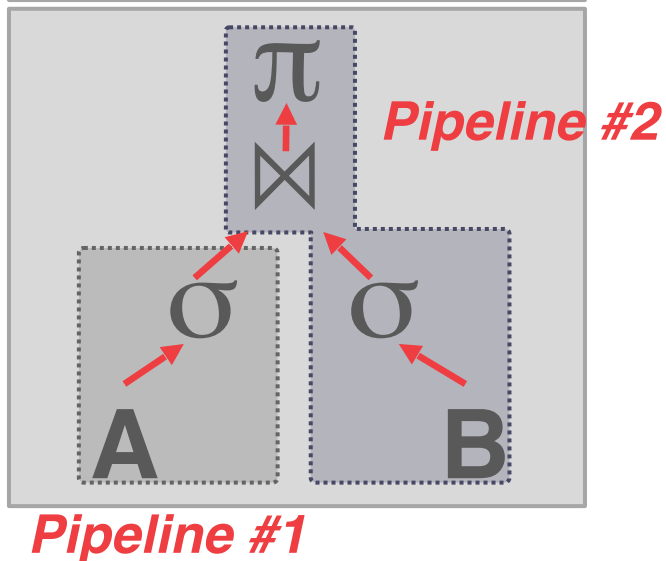
Pipeline

Pipeline #2

[illegible][illegible]

HyPer – Data Partitioning

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



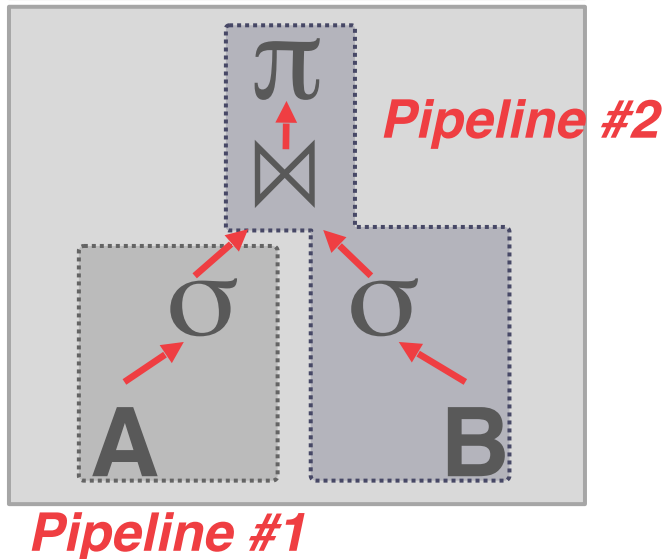
Data Table

Morsels

	id	a1	a2	a3
A_1				
A_2				
A_3				

HyPer – Data Partitioning




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

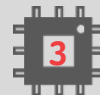
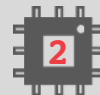
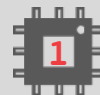


Data Table

Morsels

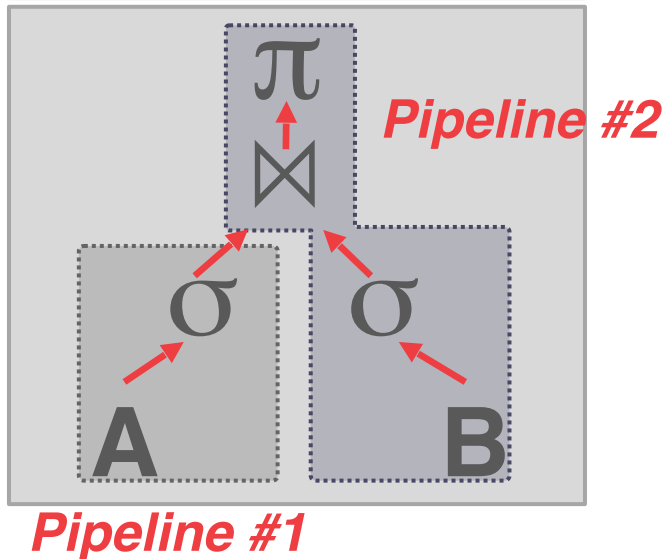
	id	a1	a2	a3		
A_1					}	
						}
A_2					}	
						}
A_3					}	
						}

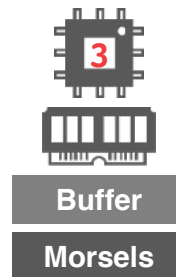
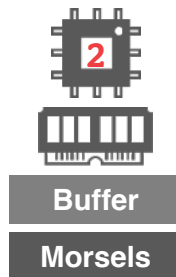
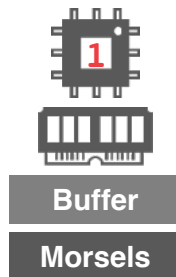
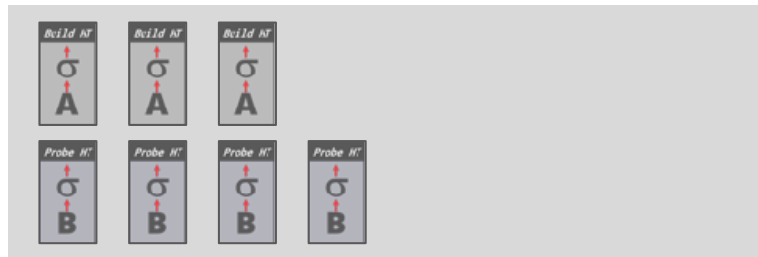


HyPer – Execution Example

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

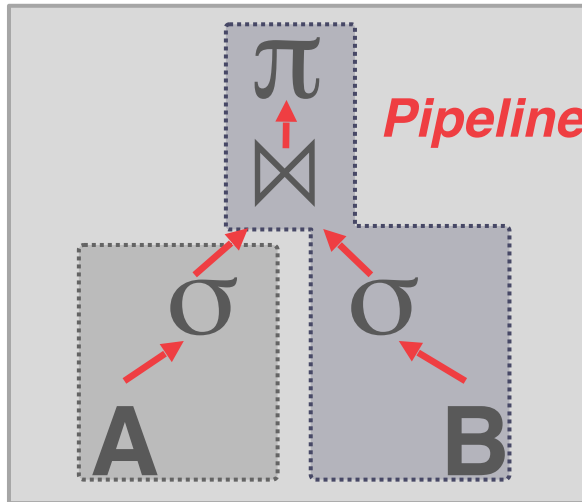


Global Task Queue



HyPer – Execution Example

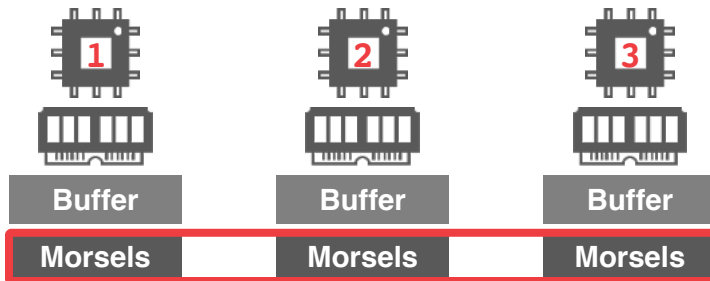
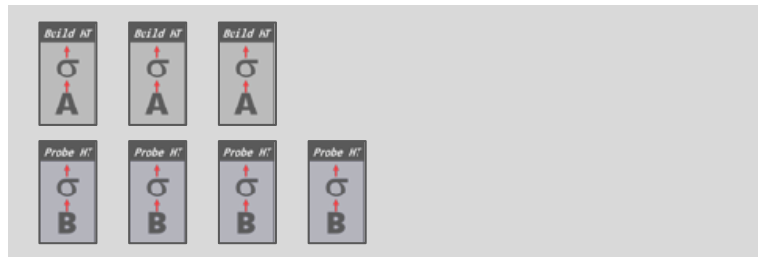
```
SELECT A.id, B.value  
FROM A JOIN B  
ON A.id = B.id  
WHERE A.value < 99  
AND B.value > 100
```



Pipeline #1

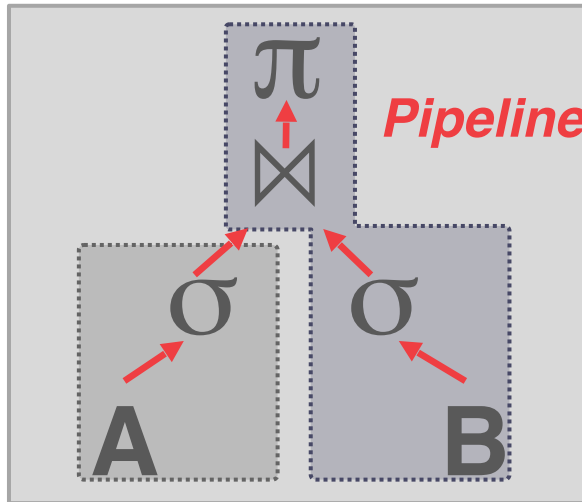
Pipeline #2

Global Task Queue



HyPer – Execution Example

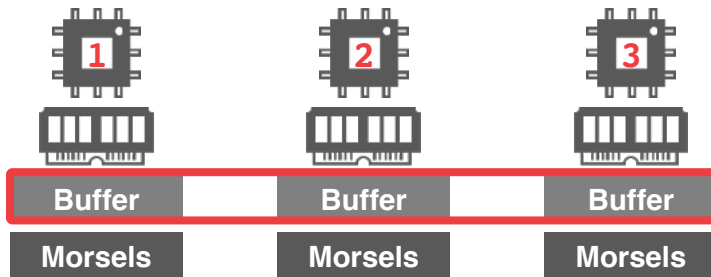
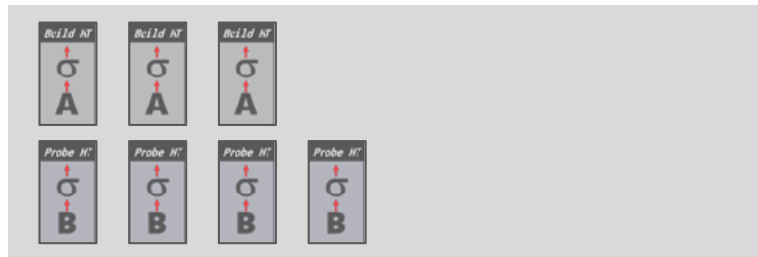
```
SELECT A.id, B.value  
FROM A JOIN B  
ON A.id = B.id  
WHERE A.value < 99  
AND B.value > 100
```



Pipeline #1

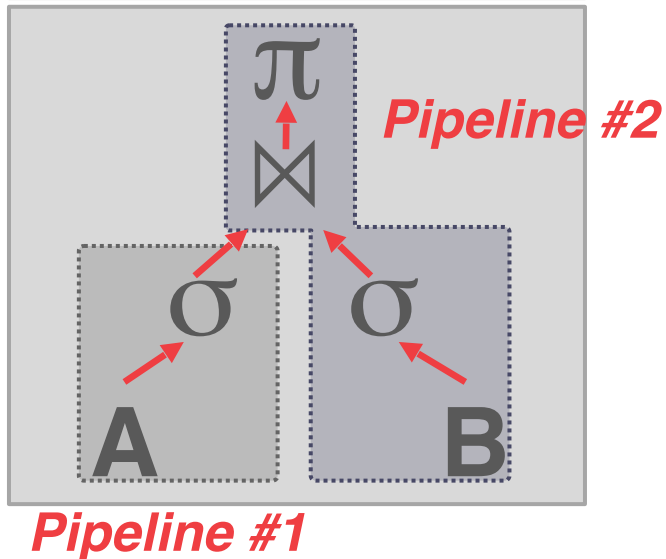
Pipeline #2

Global Task Queue

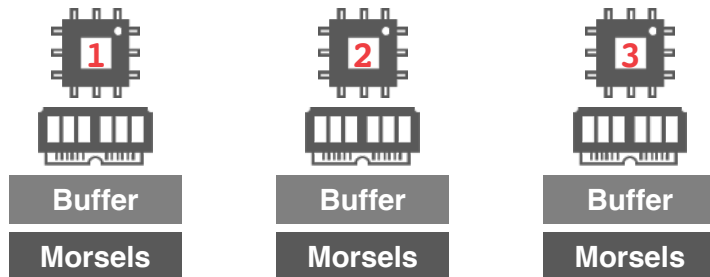
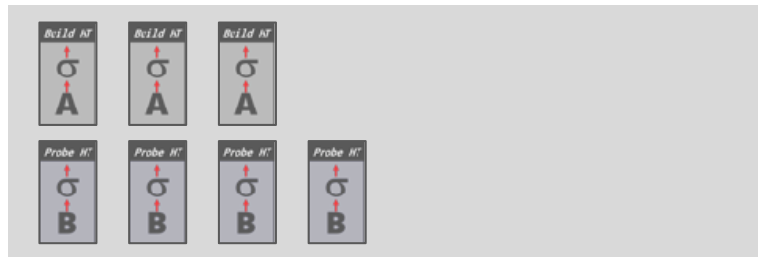


HyPer – Execution Example

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

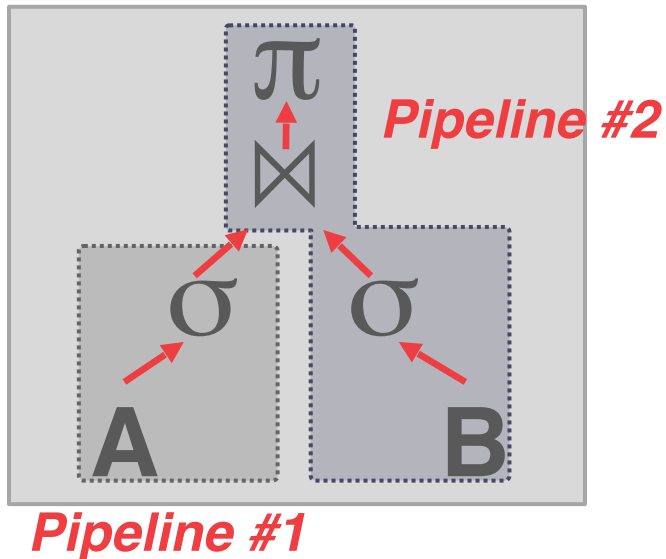


Global Task Queue

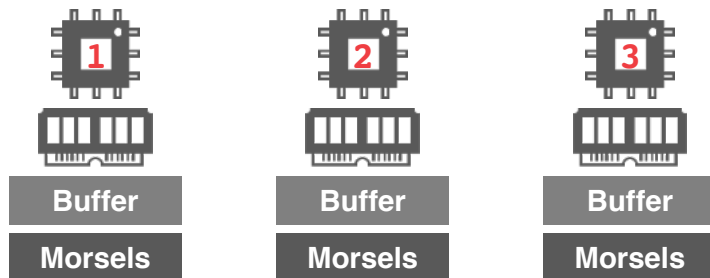
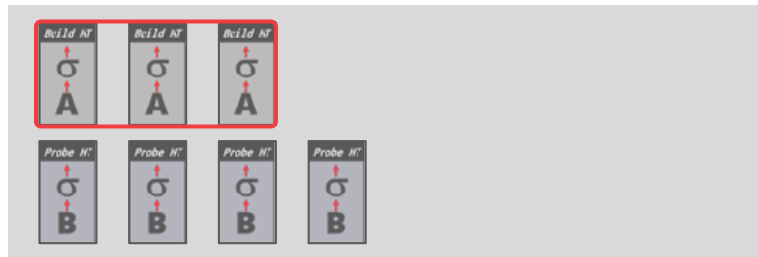


HyPer – Execution Example

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

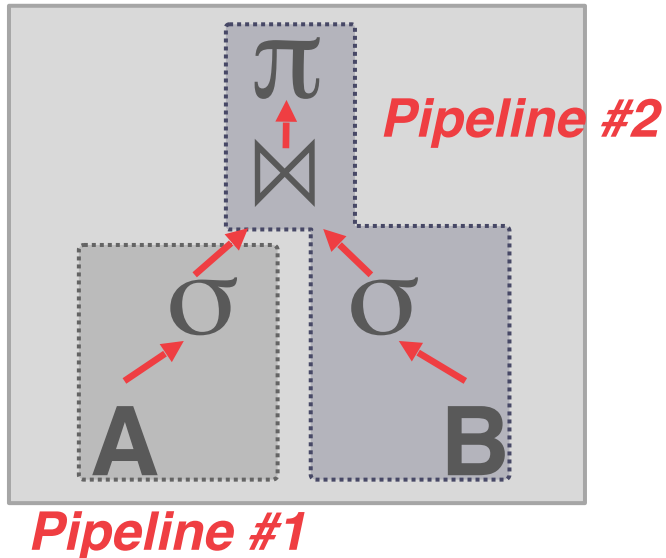


Global Task Queue

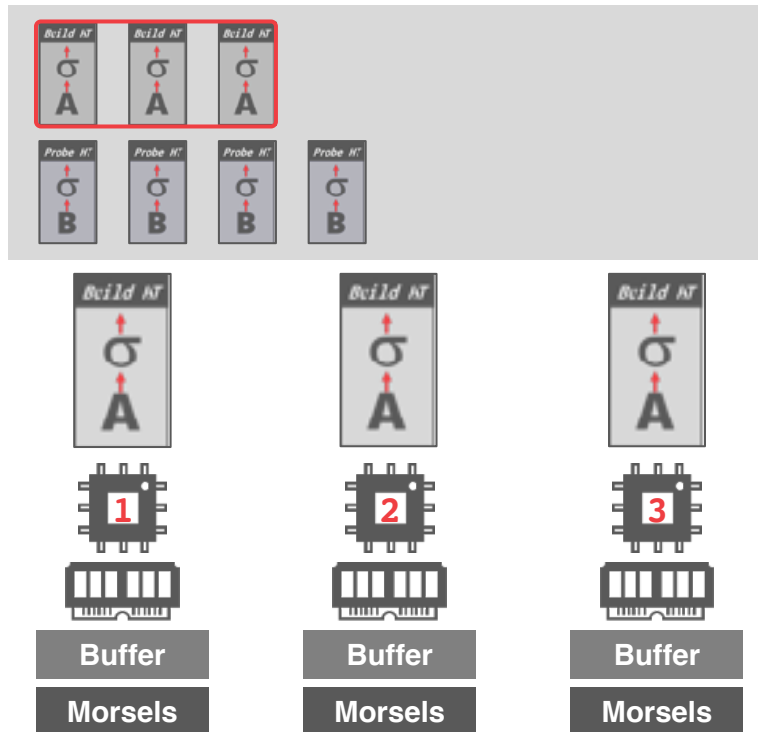


HyPer – Execution Example

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

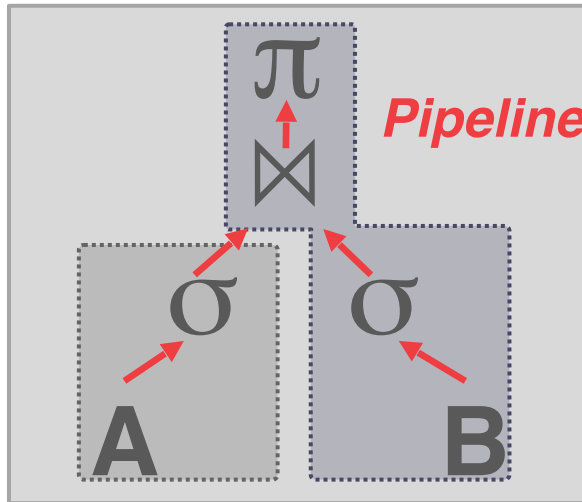


Global Task Queue



HyPer – Execution Example

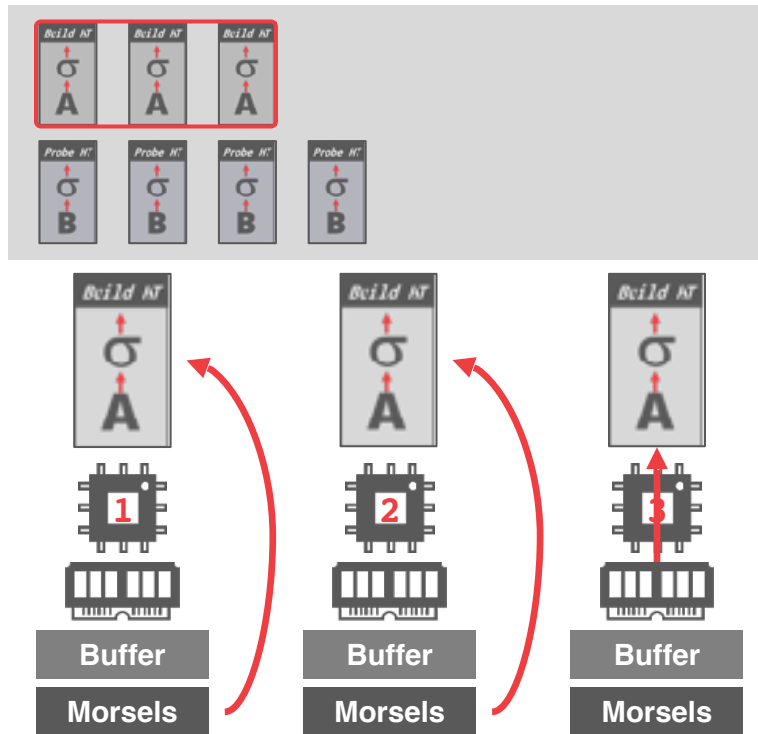
```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```



Pipeline #1

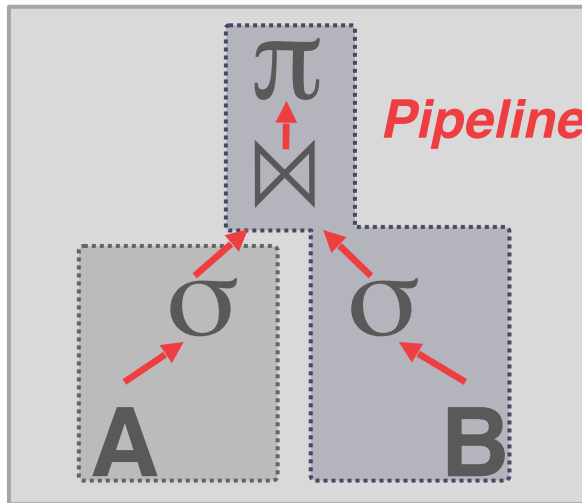
Pipeline #2

Global Task Queue



HyPer – Execution Example

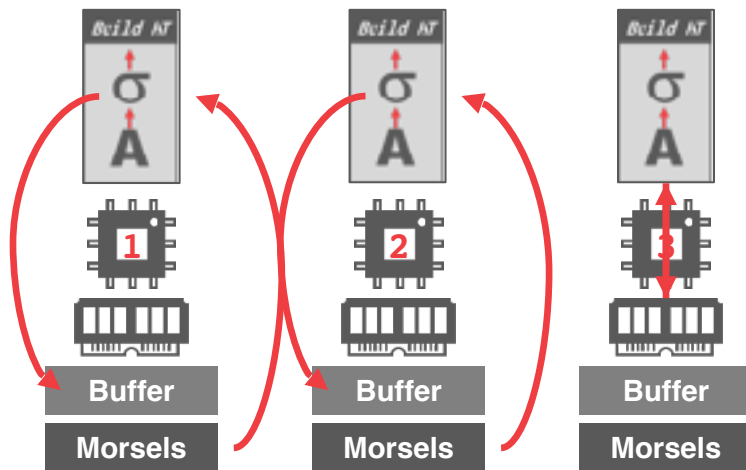
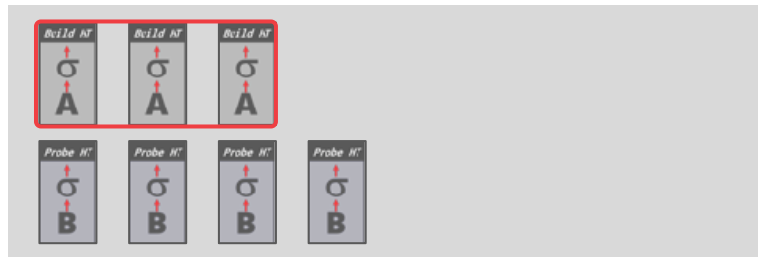
```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```



Pipeline #1

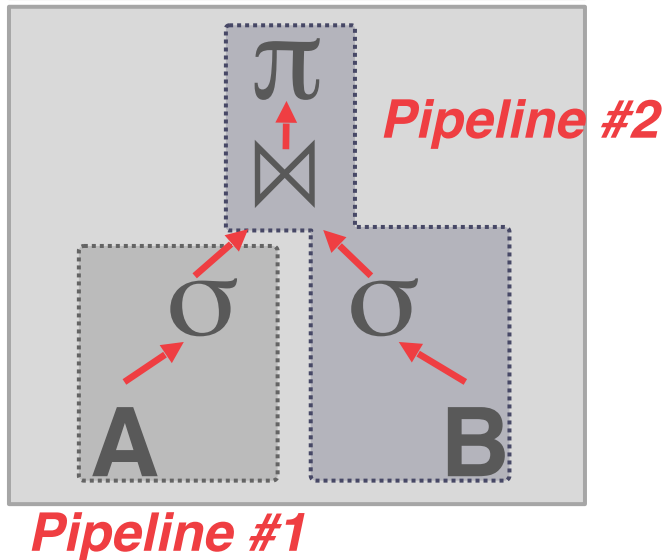
Pipeline #2

Global Task Queue

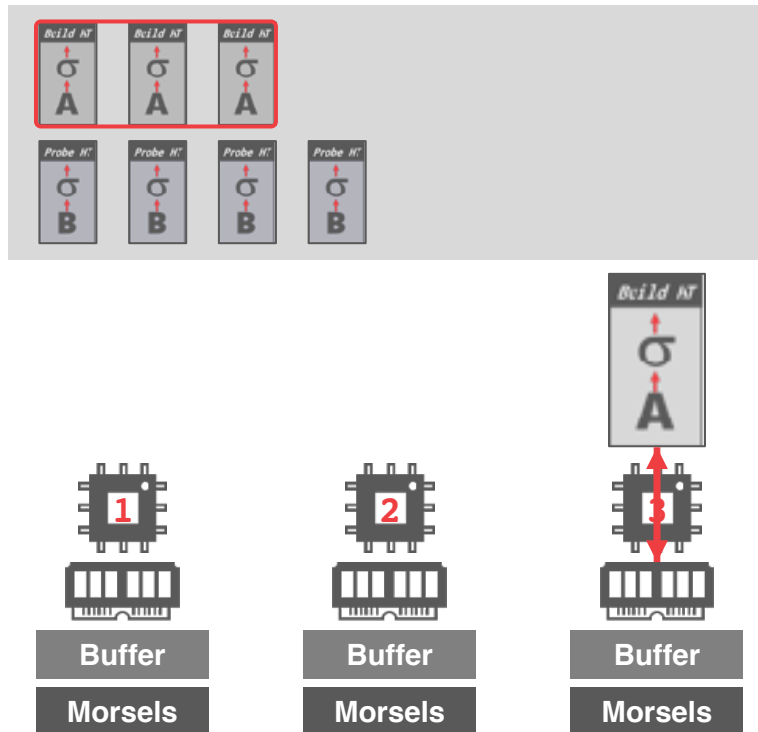


HyPer – Execution Example

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

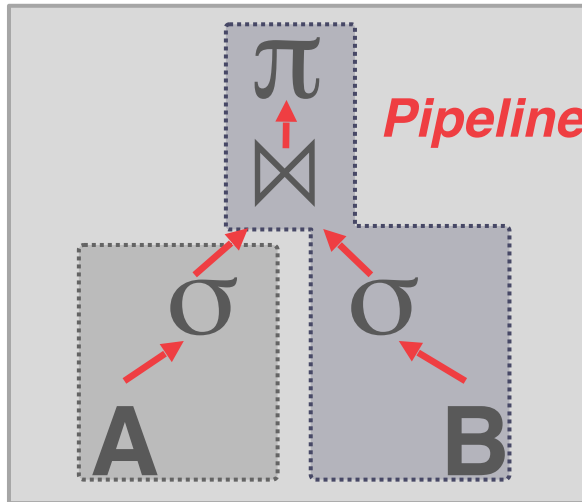


Global Task Queue



HyPer – Execution Example

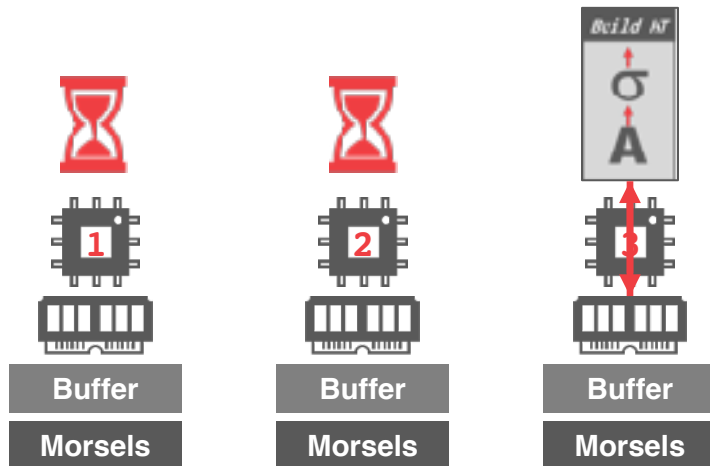
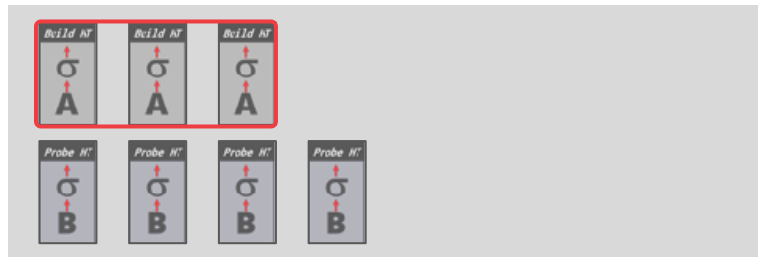
```
SELECT A.id, B.value
FROM A JOIN B
ON A.id = B.id
WHERE A.value < 99
AND B.value > 100
```



Pipeline #1

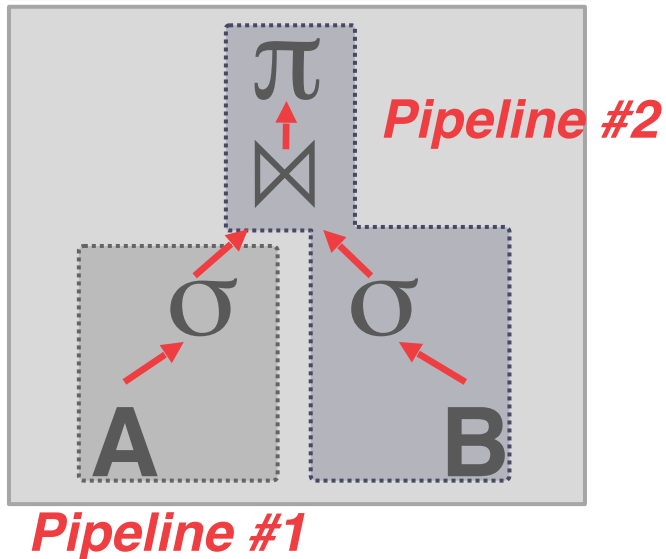
Pipeline #2

Global Task Queue

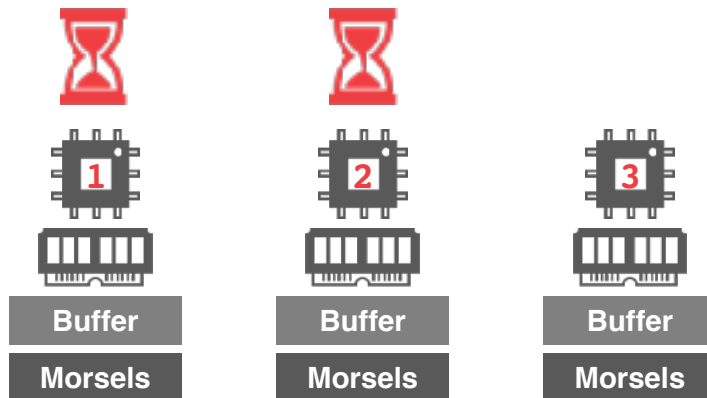
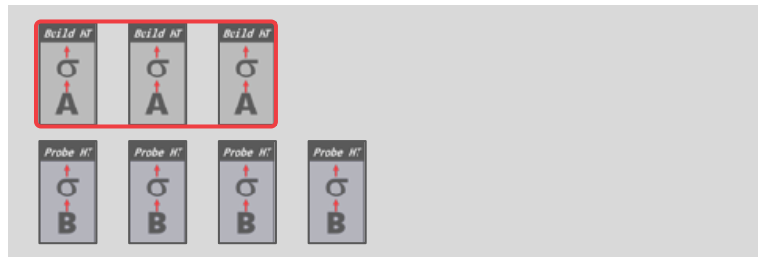


HyPer – Execution Example

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

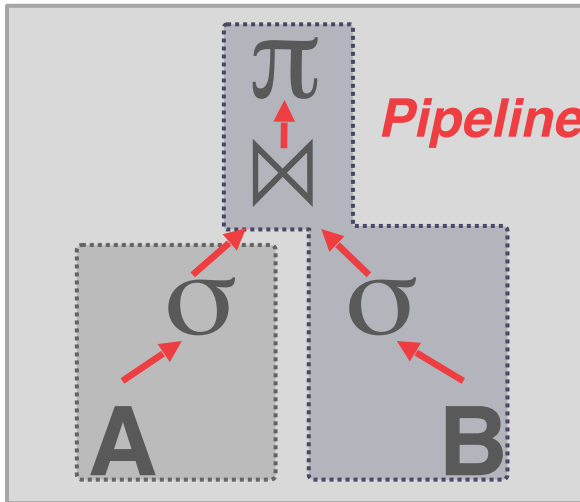


Global Task Queue



HyPer – Execution Example

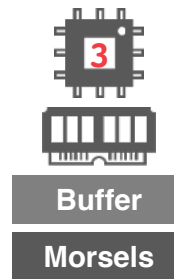
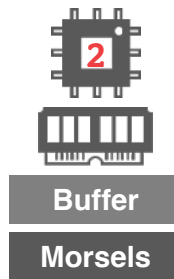
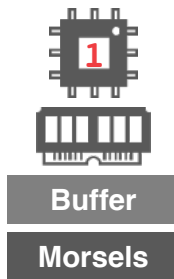
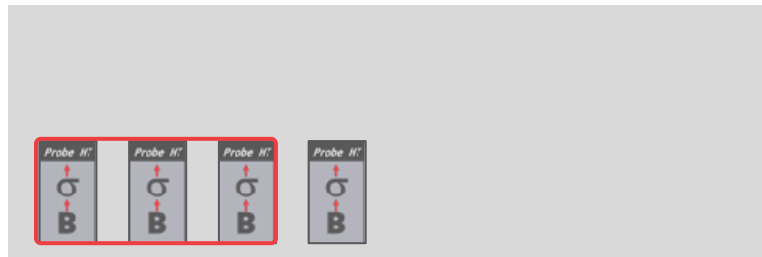
```
SELECT A.id, B.value  
FROM A JOIN B  
ON A.id = B.id  
WHERE A.value < 99  
AND B.value > 100
```



Pipeline #1

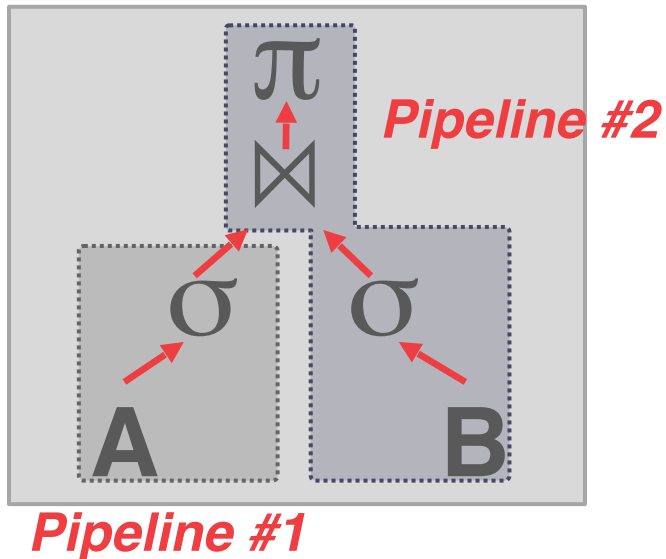
Pipeline #2

Global Task Queue

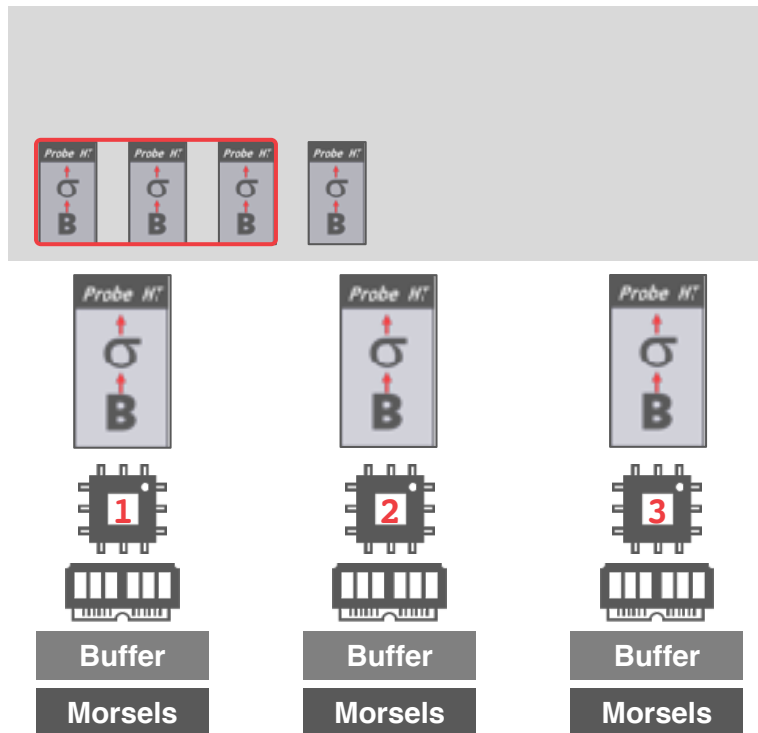


HyPer – Execution Example

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

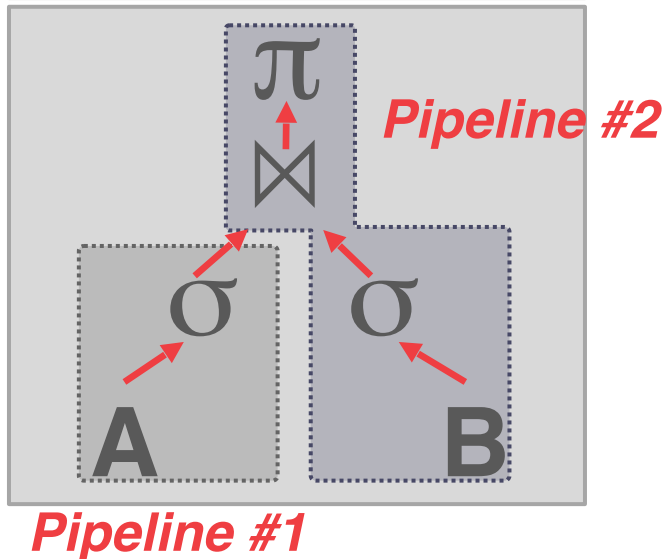


Global Task Queue

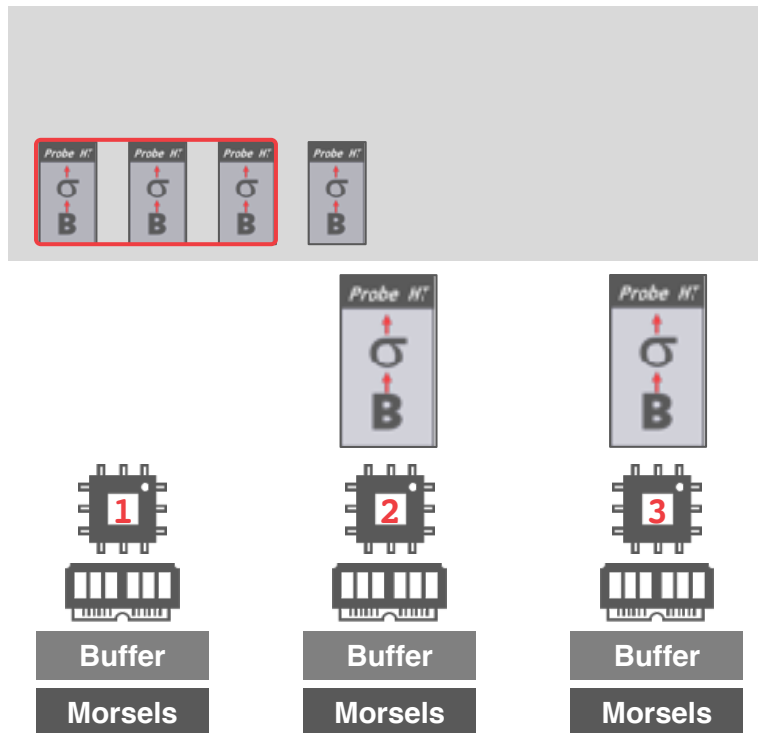


HyPer – Execution Example

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

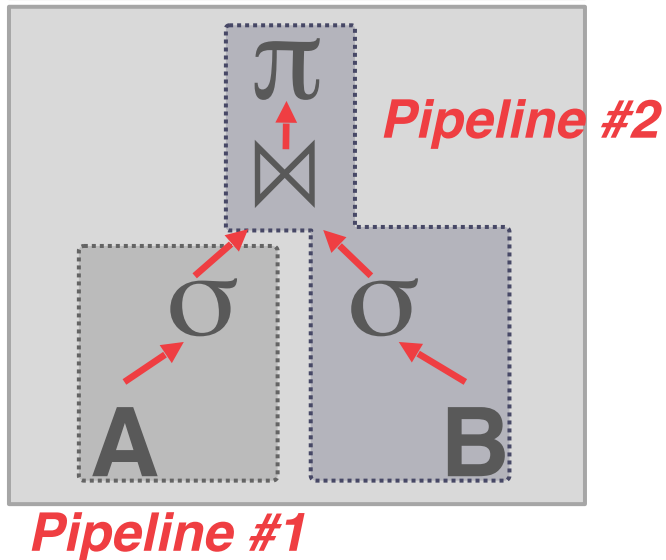


Global Task Queue

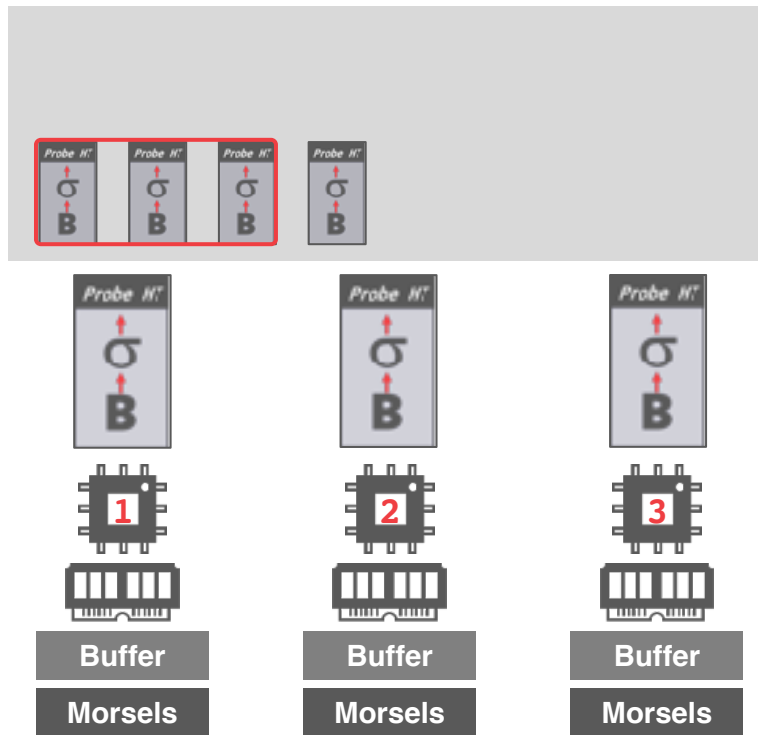


HyPer – Execution Example

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

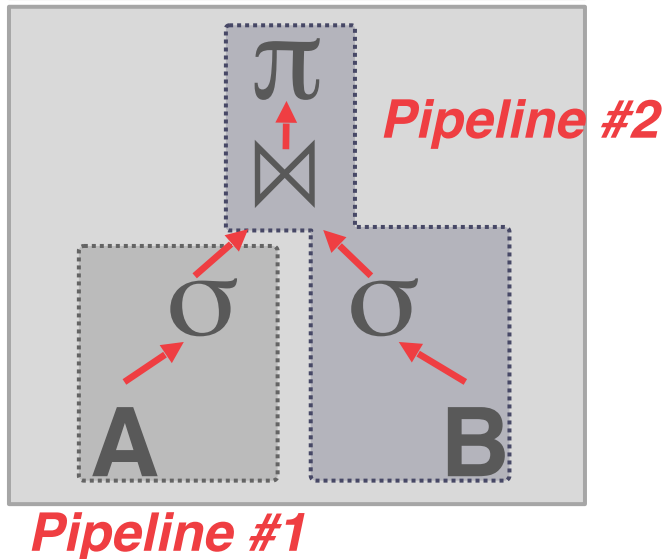


Global Task Queue

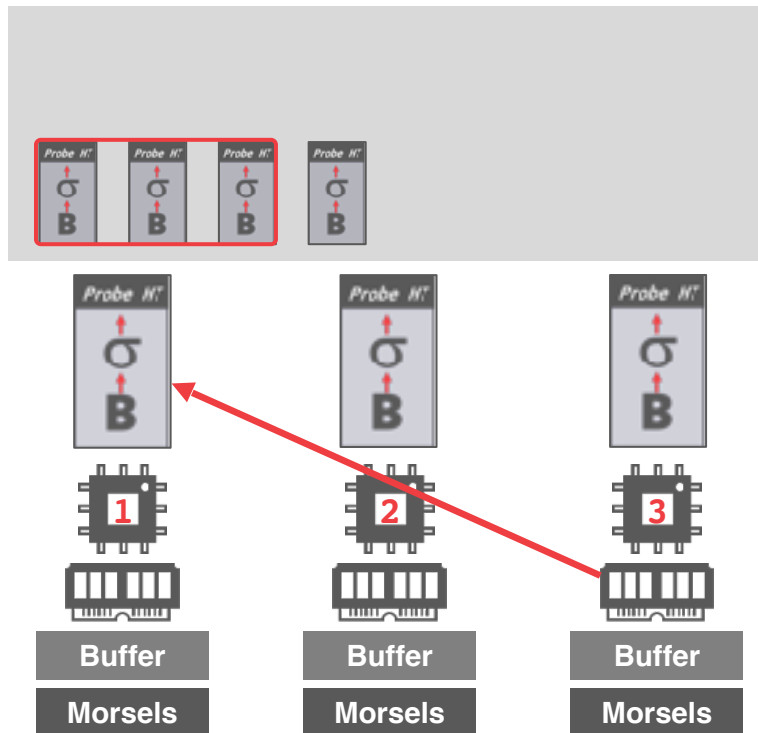


HyPer – Execution Example

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

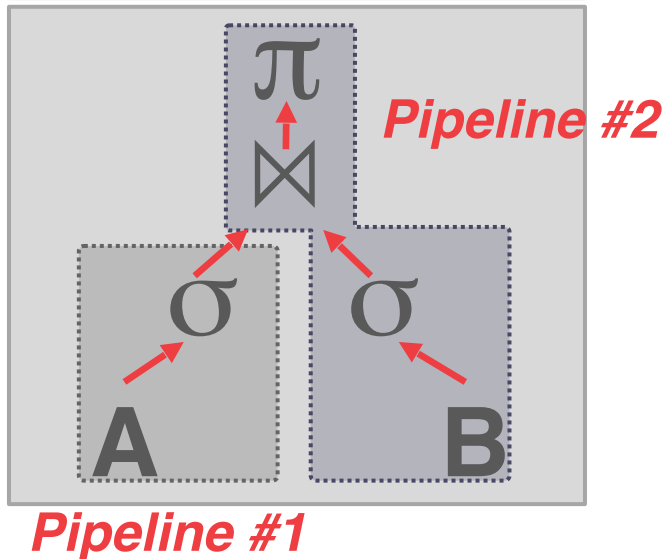


Global Task Queue

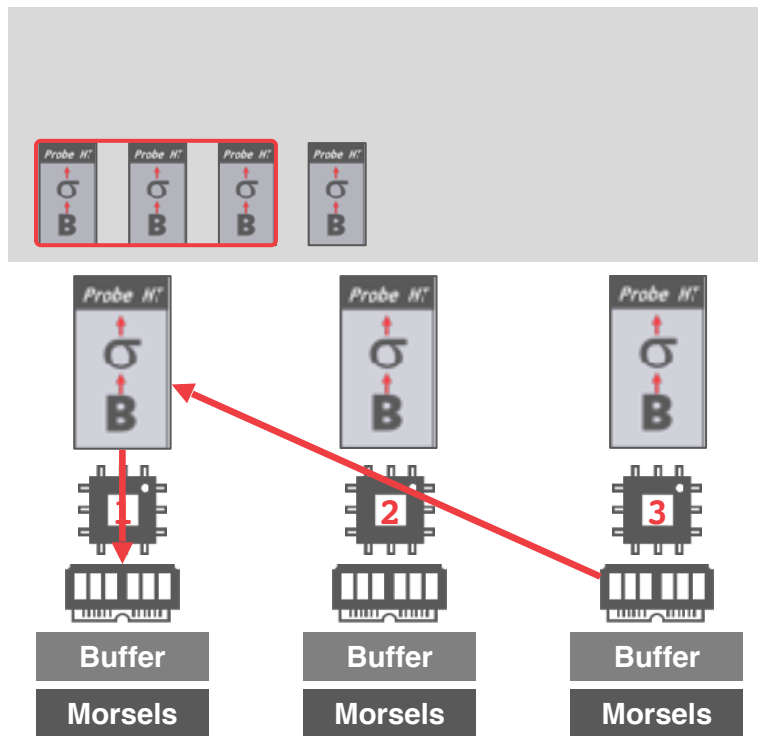


HyPer – Execution Example

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



Global Task Queue



Morsel-Drive Scheduling

Because there is only one worker per core and one morsel per task, HyPer must use work stealing because otherwise threads could sit idle waiting for stragglers.

The DBMS uses a lock-free hash table to maintain the global work queues.

Parting Thoughts

We discussed query processing models.

- Vectorized model is best for OLAP.
- Top-to-bottom (push) approach is probably better.

Next Class

Execution Optimizations