# 15-721 DATABASE SYSTEMS

Lecture #10 – Query Execution & Scheduling

Andy Pavlo // Carnegie Mellon University // Spring 2016

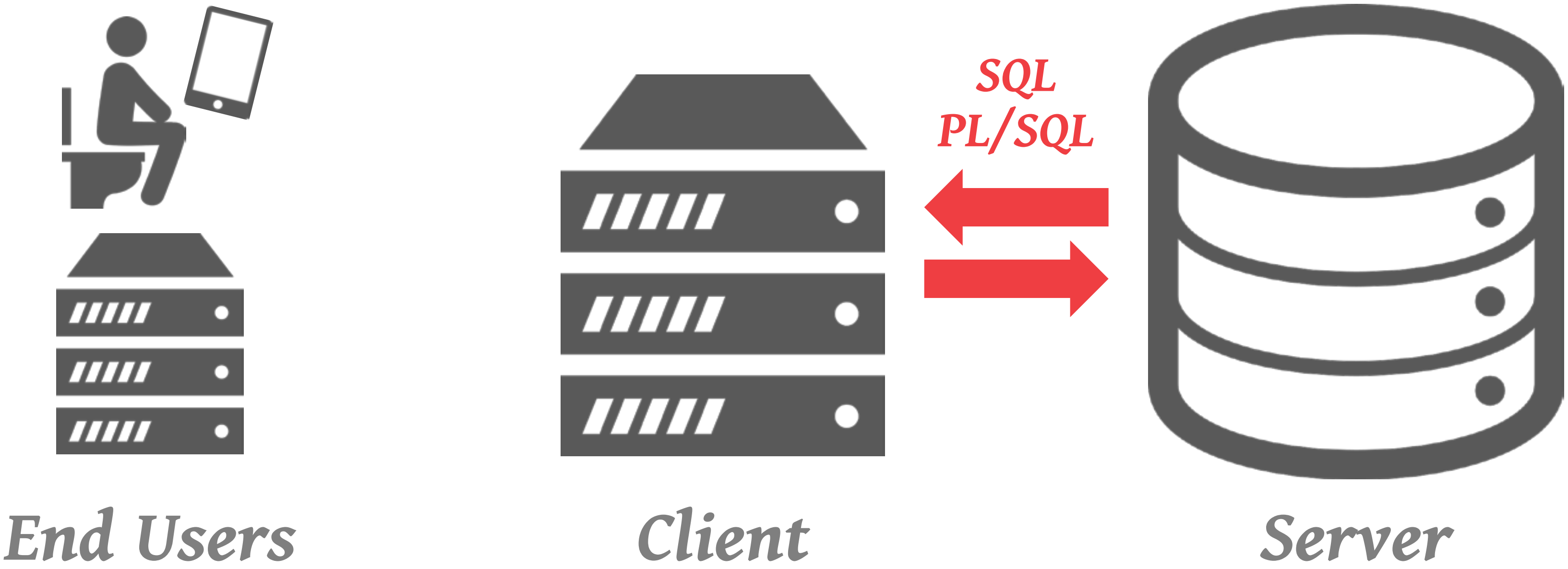
# TODAY'S AGENDA

Process Models
Query Parallelization
Data Placement
Scheduling
10 Crack Commandments

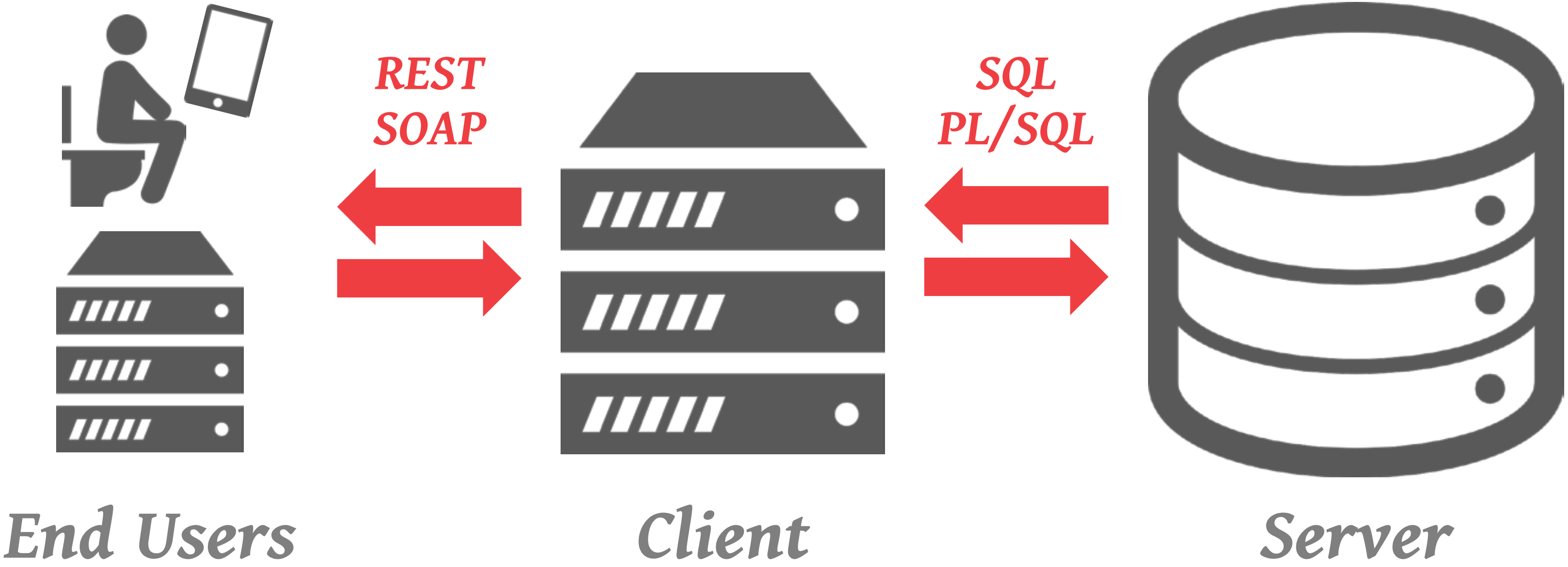
# MULTI-USER DATABASE APP STACK

*End Users* *Client* *Server*

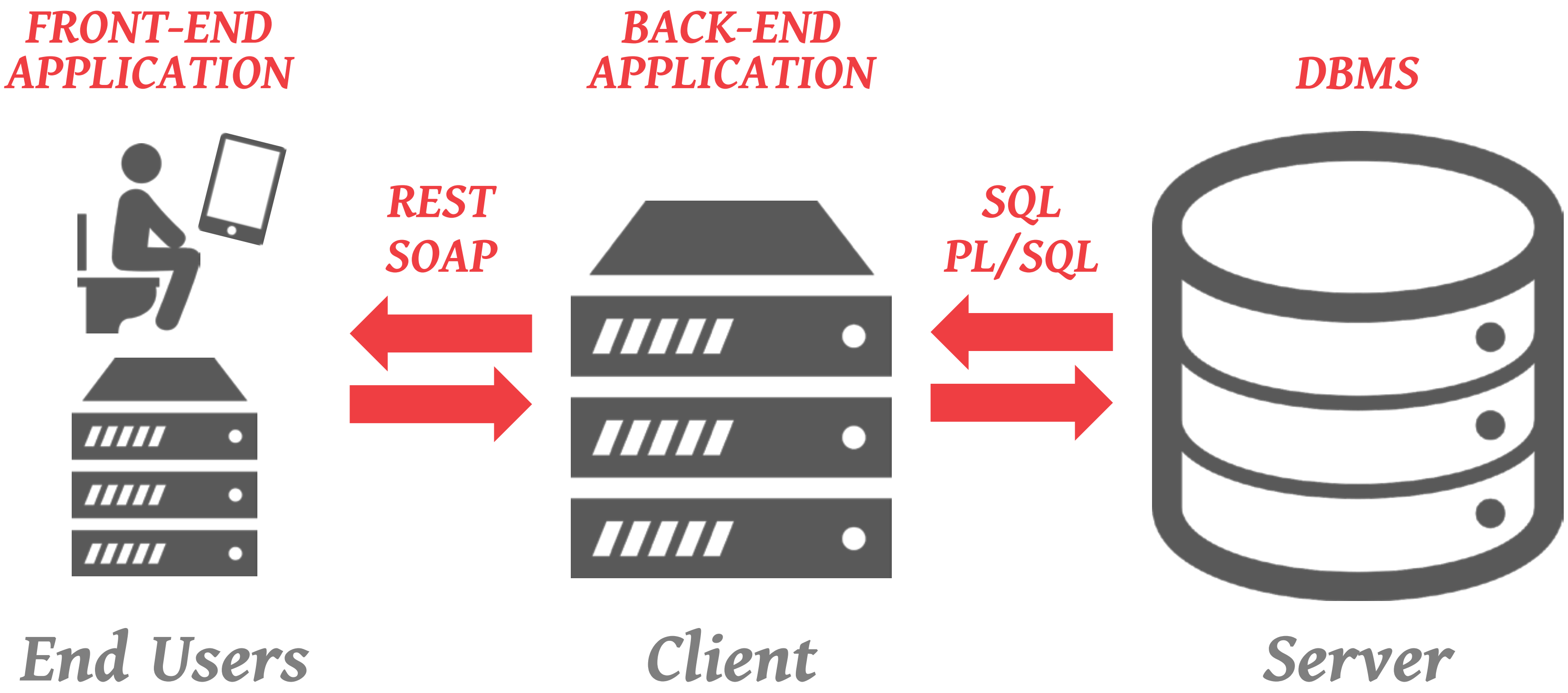
# MULTI-USER DATABASE APP STACK



# MULTI-USER DATABASE APP STACK



# MULTI-USER DATABASE APP STACK



# 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.

# 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.

ARCHITECTURE OF A DATABASE SYSTEM
*Foundations and Trends in Databases 2007*

# PROCESS MODELS

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

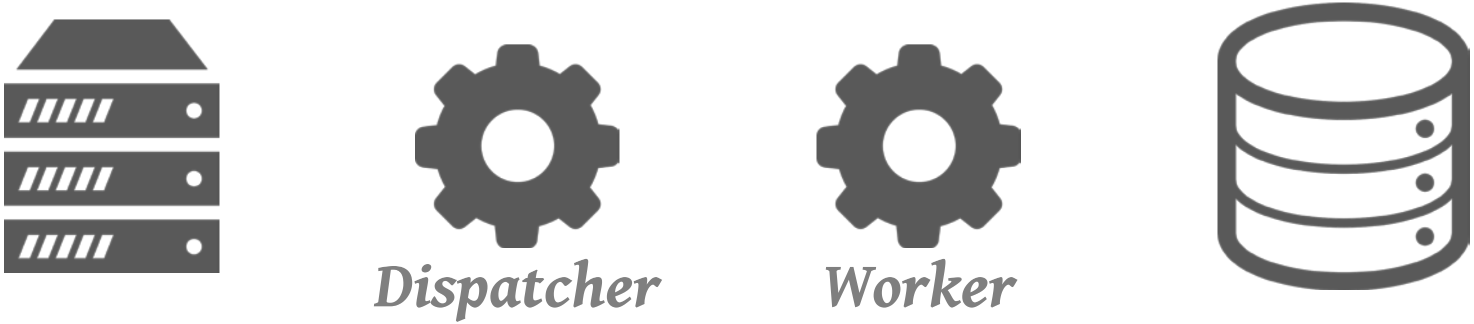
**Approach #2: Process Pool**

**Approach #3: Thread per DBMS Worker**

# PROCESS PER WORKER

Each worker is a separate OS process.

→ Relies on OS scheduler.
→ Use shared-memory for global data structures.
→ Examples: IBM DB2, Postgres, Oracle



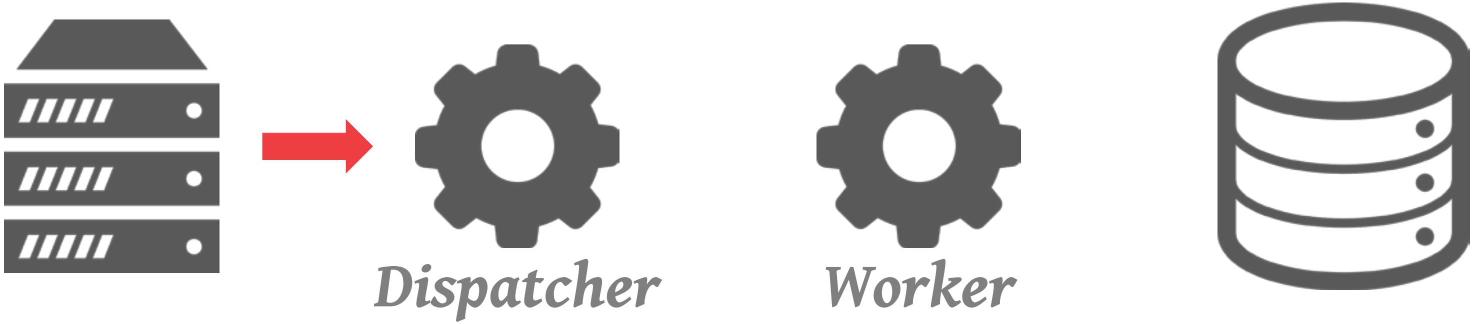
# PROCESS PER WORKER

Each worker is a separate OS process.

→ Relies on OS scheduler.

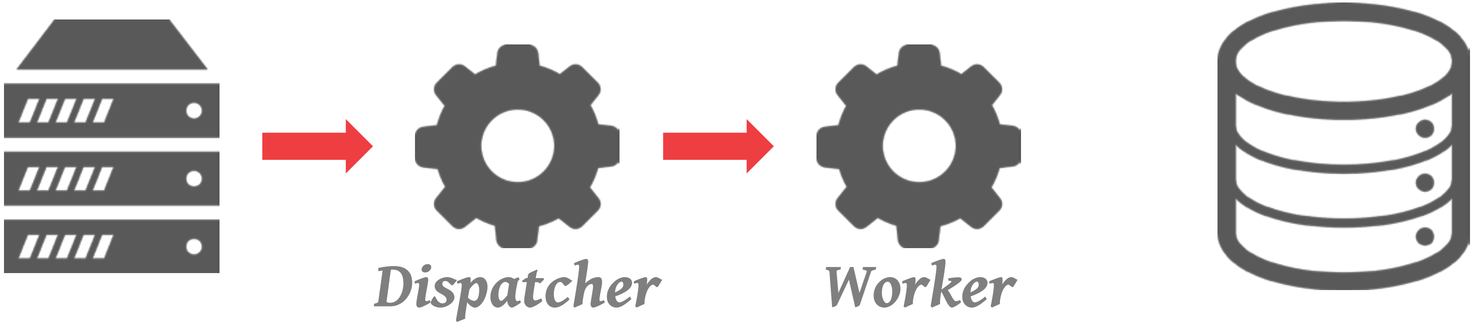
→ Use shared-memory for global data structures.

→ Examples: IBM DB2, Postgres, Oracle



# PROCESS PER WORKER

Each worker is a separate OS process.
→ Relies on OS scheduler.
→ Use shared-memory for global data structures.
→ Examples: IBM DB2, Postgres, Oracle



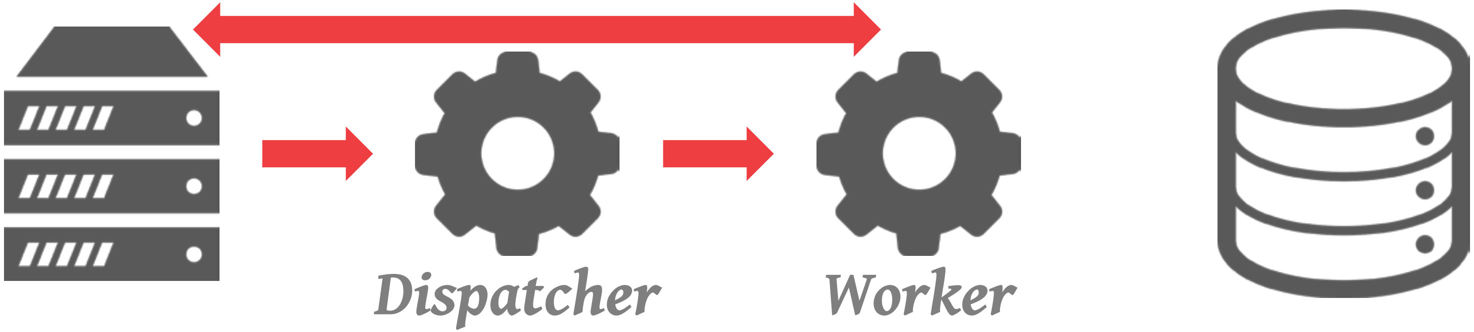
# PROCESS PER WORKER

Each worker is a separate OS process.

→ Relies on OS scheduler.

→ Use shared-memory for global data structures.

→ Examples: IBM DB2, Postgres, Oracle



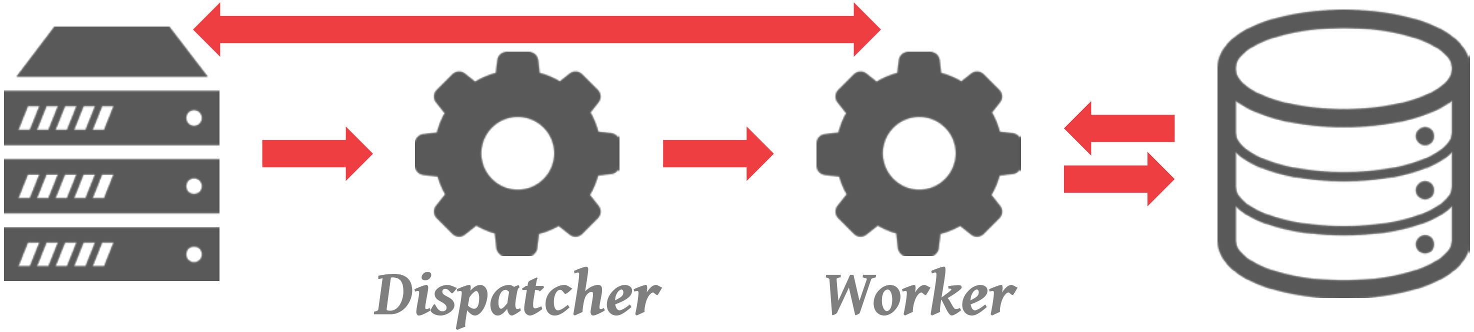
# PROCESS PER WORKER

Each worker is a separate OS process.

→ Relies on OS scheduler.

→ Use shared-memory for global data structures.

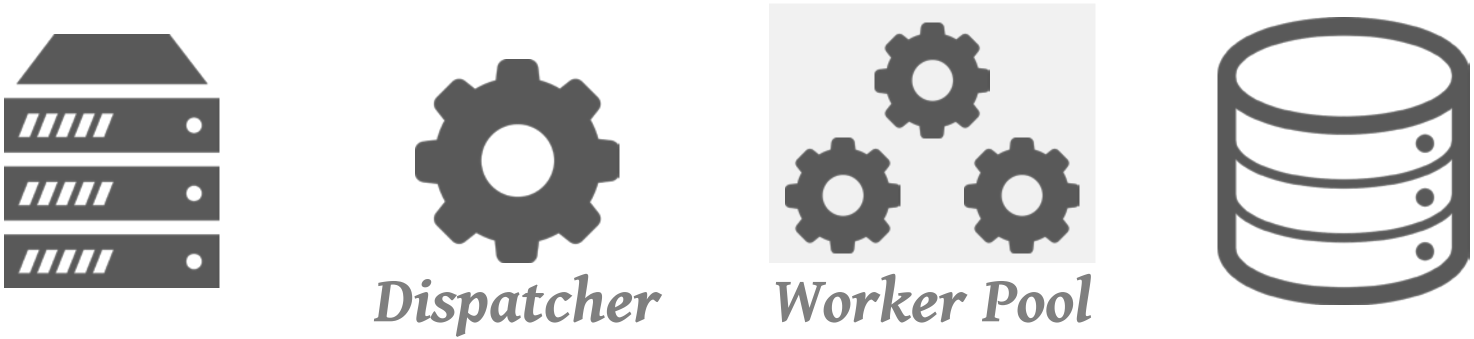
→ Examples: IBM DB2, Postgres, Oracle



# PROCESS POOL

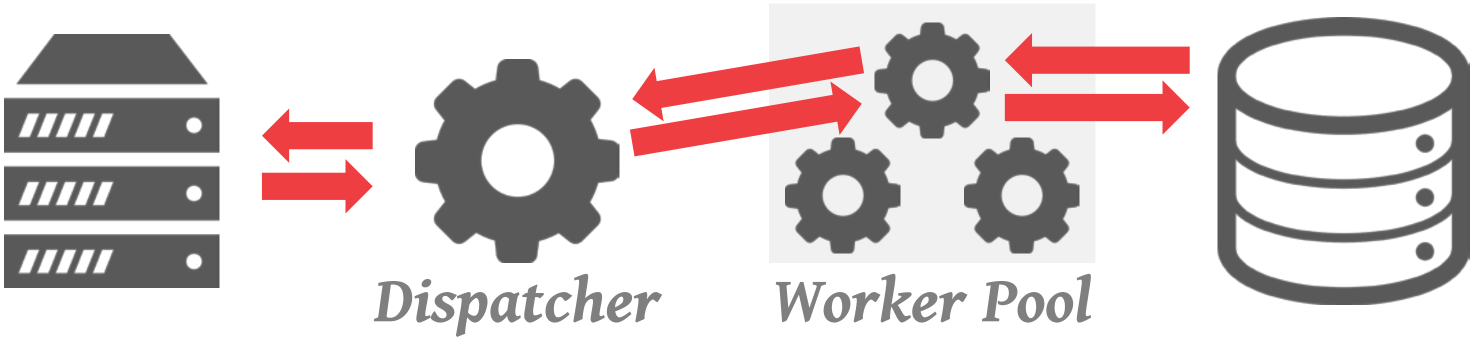
A worker uses any process that is free in a pool

→ Still relies on OS scheduler and shared memory.
→ Bad for CPU cache locality.
→ Examples: IBM DB2



# PROCESS POOL

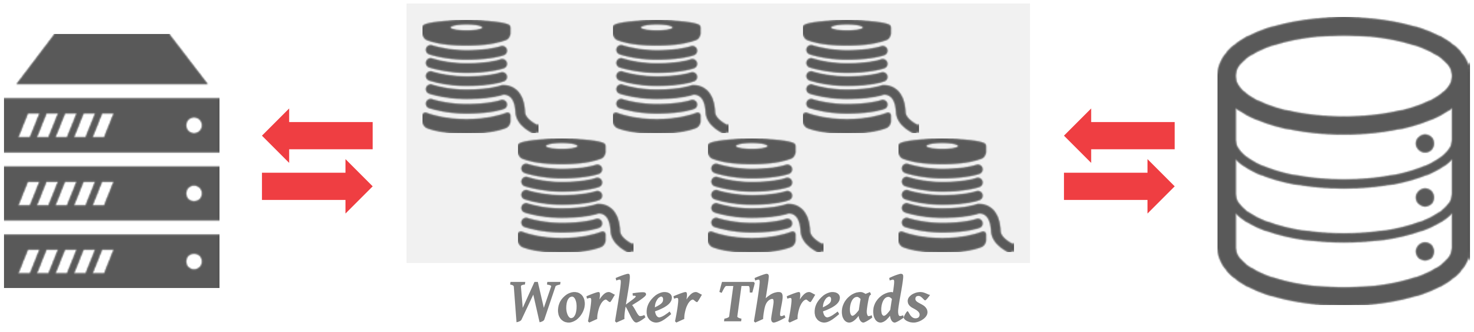
A worker uses any process that is free in a pool
→ Still relies on OS scheduler and shared memory.
→ Bad for CPU cache locality.
→ Examples: IBM DB2



# THREAD PER WORKER

Single process with multiple worker threads.

→ DBMS has to manage its own scheduling.
→ May or may not use a dispatcher thread.
→ Examples: IBM DB2, MSSQL, MySQL, Oracle (Newer)



# PROCESS MODELS

Using a multi-threaded architecture has several advantages:
→ Less overhead per context switch.
→ Don't have to manage shared memory.

The thread per worker model does not mean that you have intra-query parallelism.

I am not aware of any new DBMS built in the last 7-8 years that doesn't use threads.

# SCHEDULING

For each query plan, the DBMS has to 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.

# 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

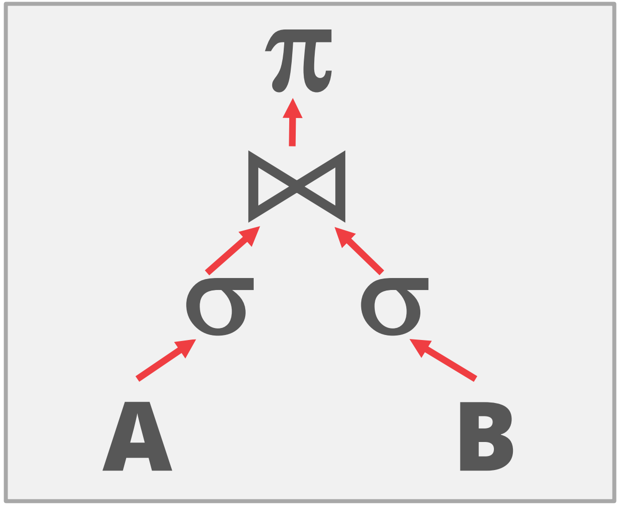
Improve the performance of a single query by executing its operators in parallel.

**Approach #1: Intra-Operator (Horizontal)**
→ Operators are decomposed into independent instances that perform the same function on different subsets of data.

**Approach #2: Inter-Operator (Vertical)**
→ Operations are overlapped in order to pipeline data from one stage to the next without materialization.

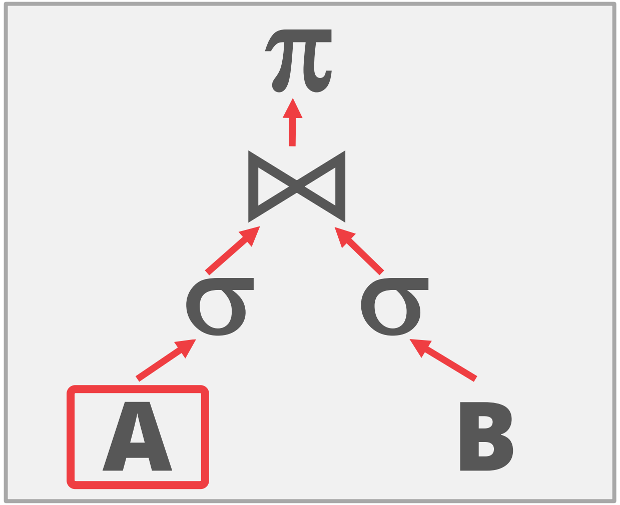
# INTRA-OPERATOR PARALLELISM

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



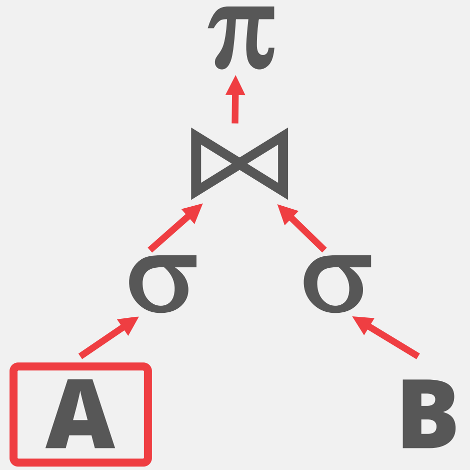
# INTRA-OPERATOR PARALLELISM

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



# INTRA-OPERATOR PARALLELISM

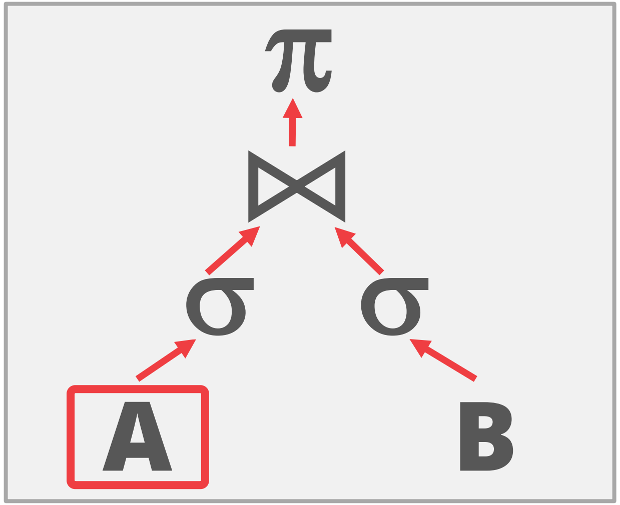
```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

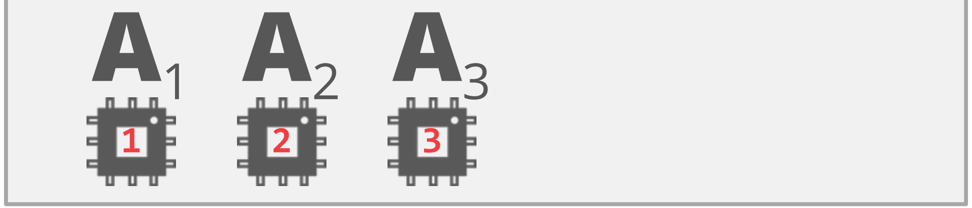



$A_1$ $A_2$ $A_3$

# INTRA-OPERATOR PARALLELISM

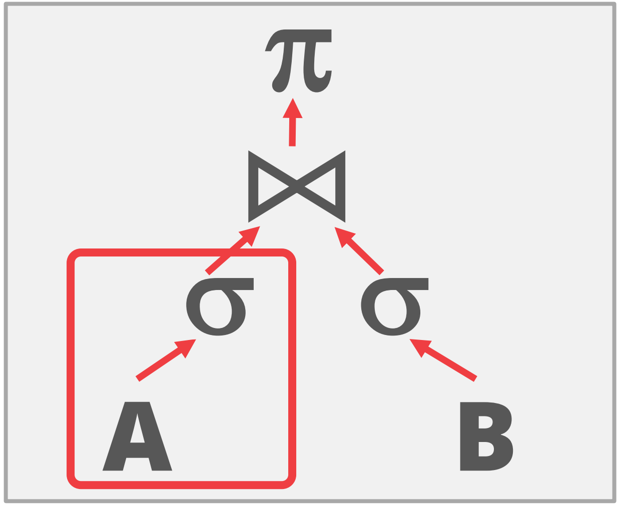
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

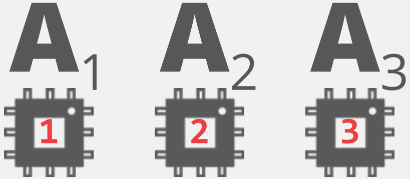


# INTRA-OPERATOR PARALLELISM

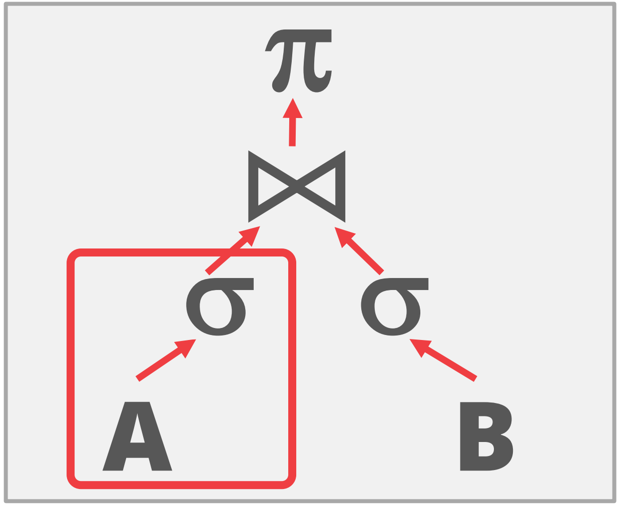
```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

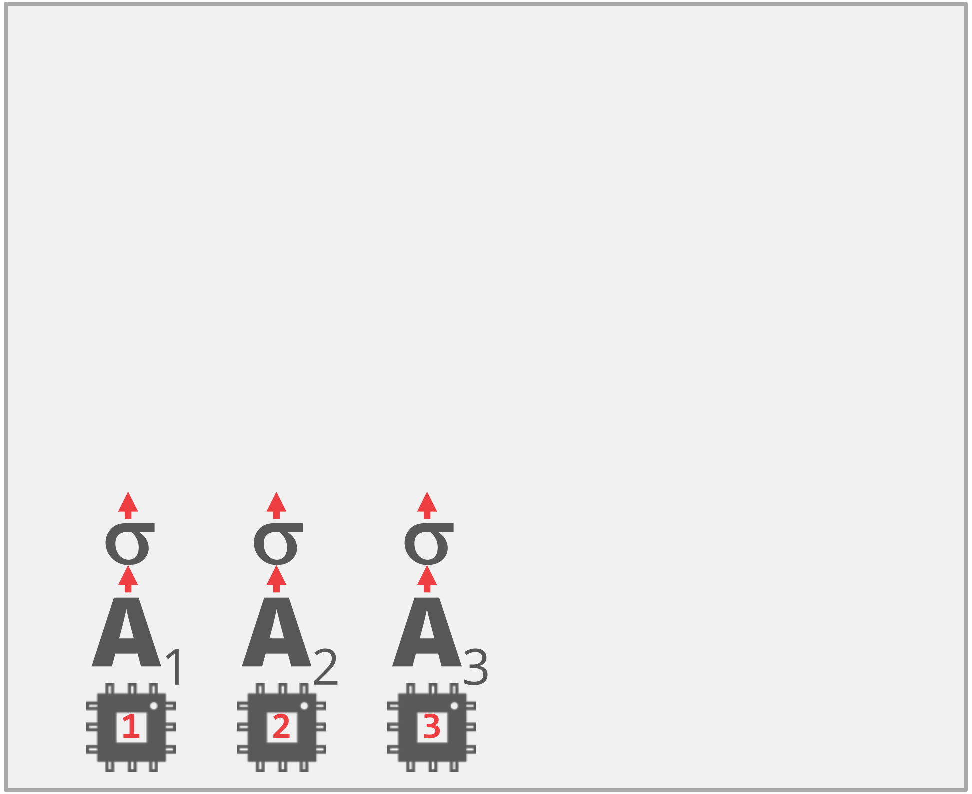


# INTRA-OPERATOR PARALLELISM

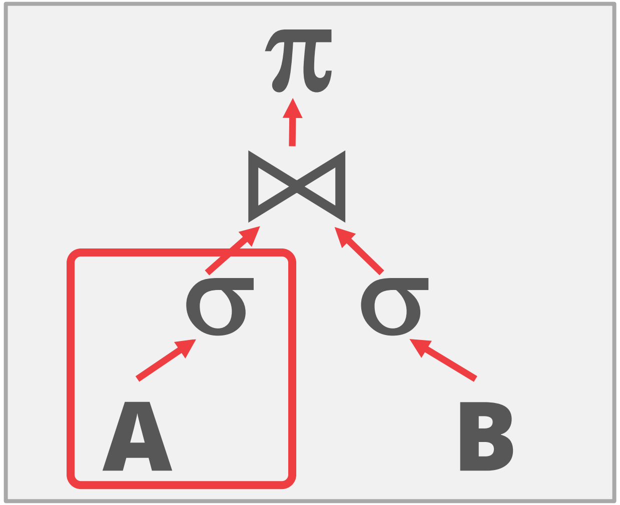
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

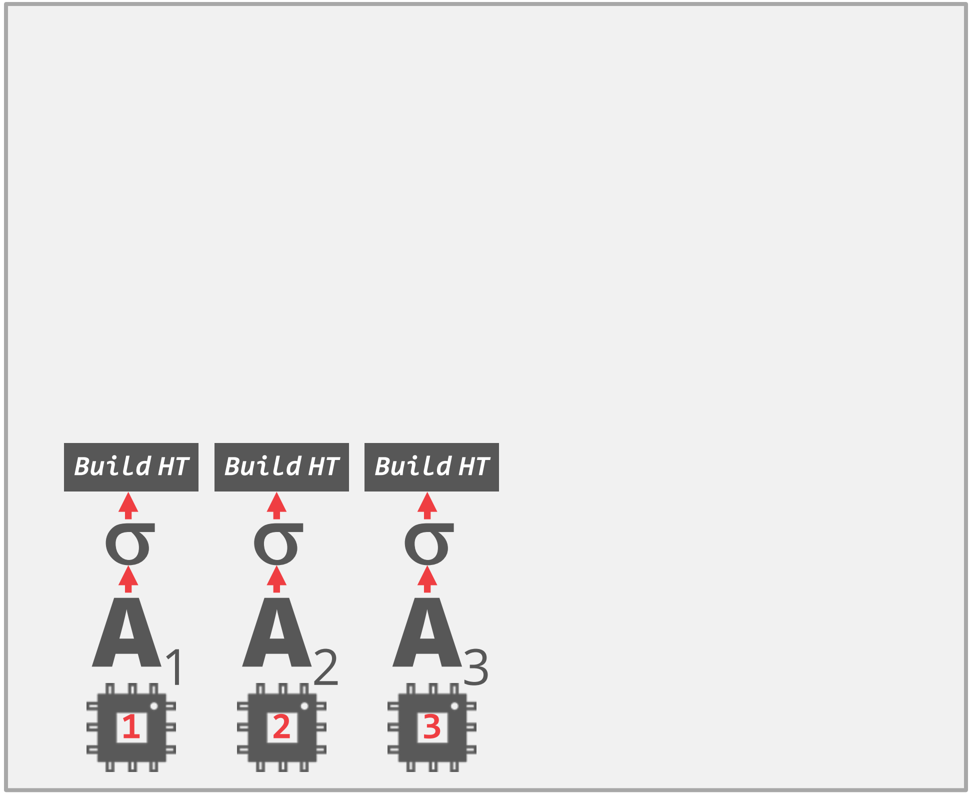


# INTRA-OPERATOR PARALLELISM

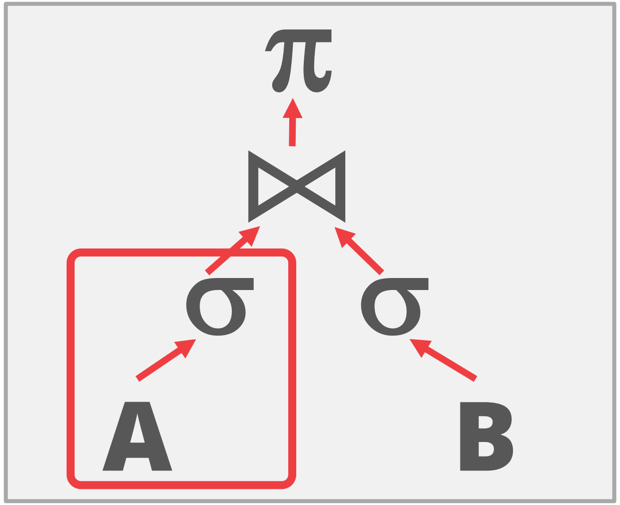
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

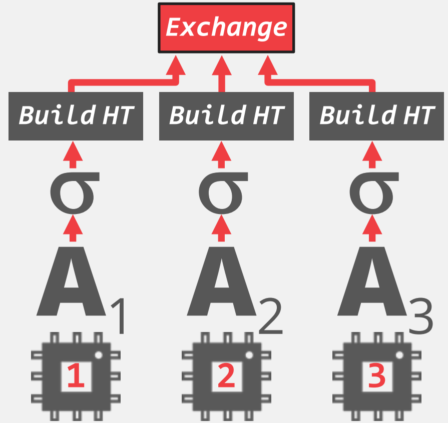


# INTRA-OPERATOR PARALLELISM

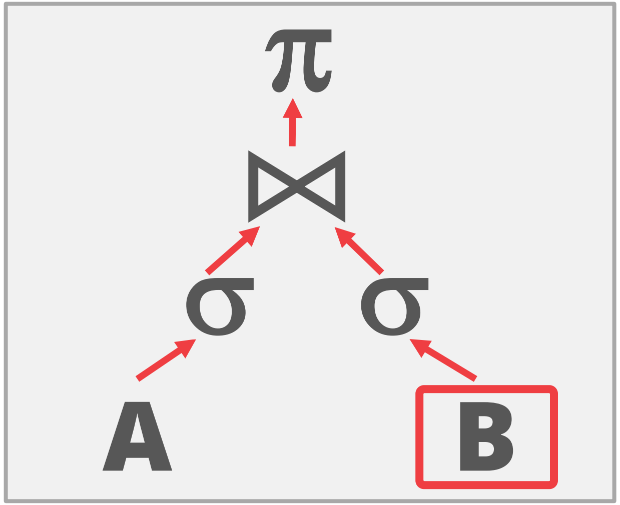
```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

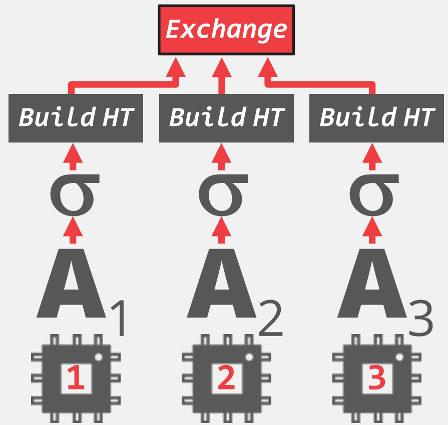


# INTRA-OPERATOR PARALLELISM

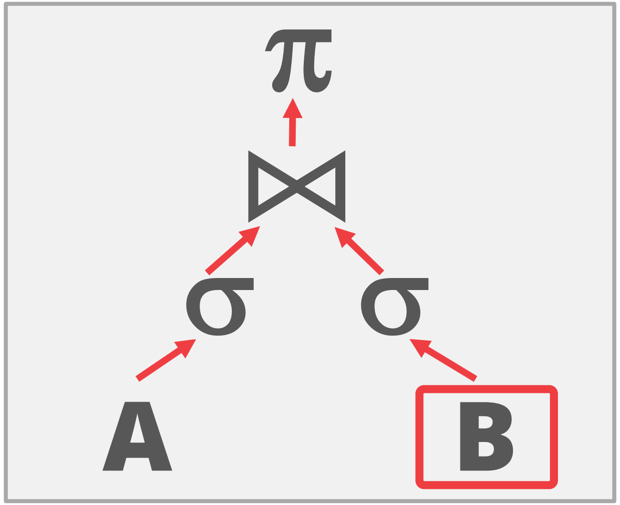
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

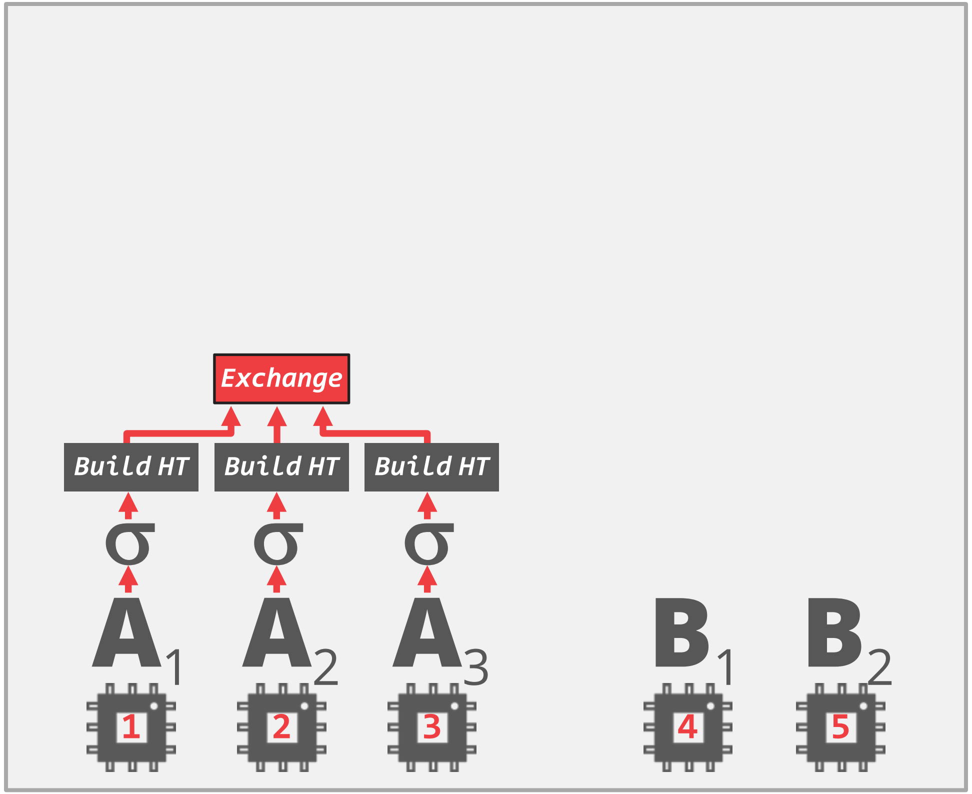


# INTRA-OPERATOR PARALLELISM

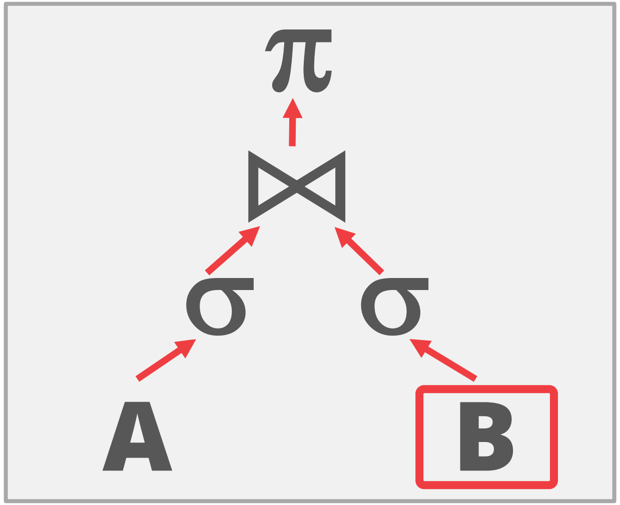
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

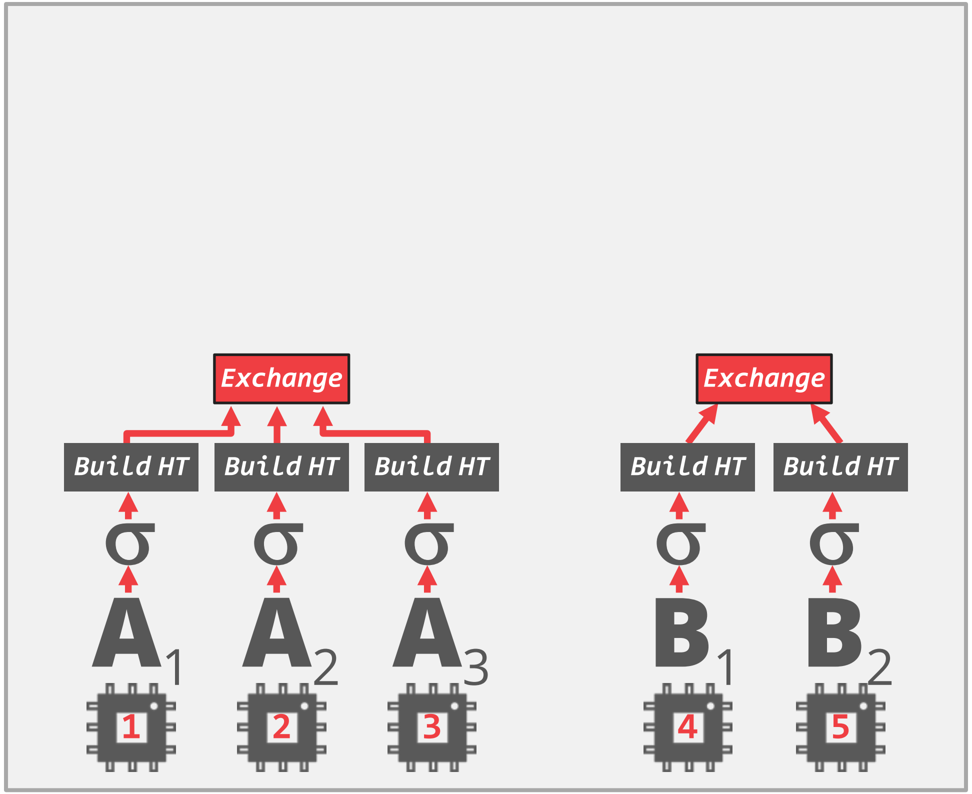


# INTRA-OPERATOR PARALLELISM

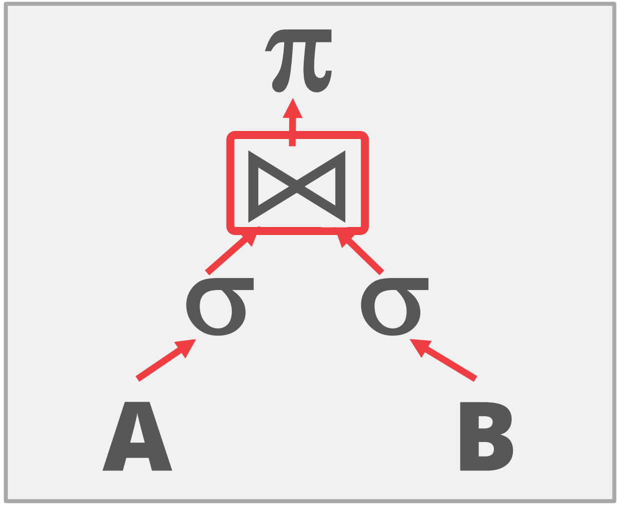
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

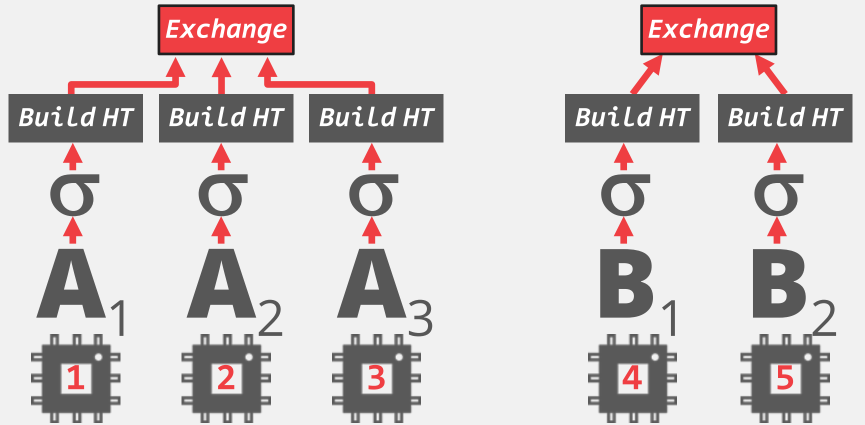


# INTRA-OPERATOR PARALLELISM

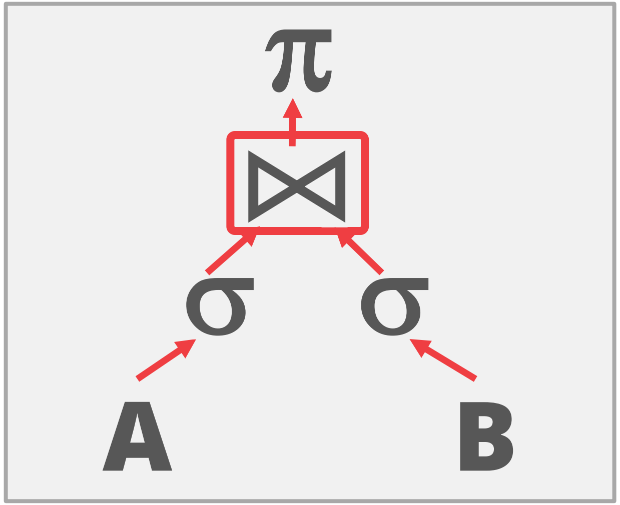
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

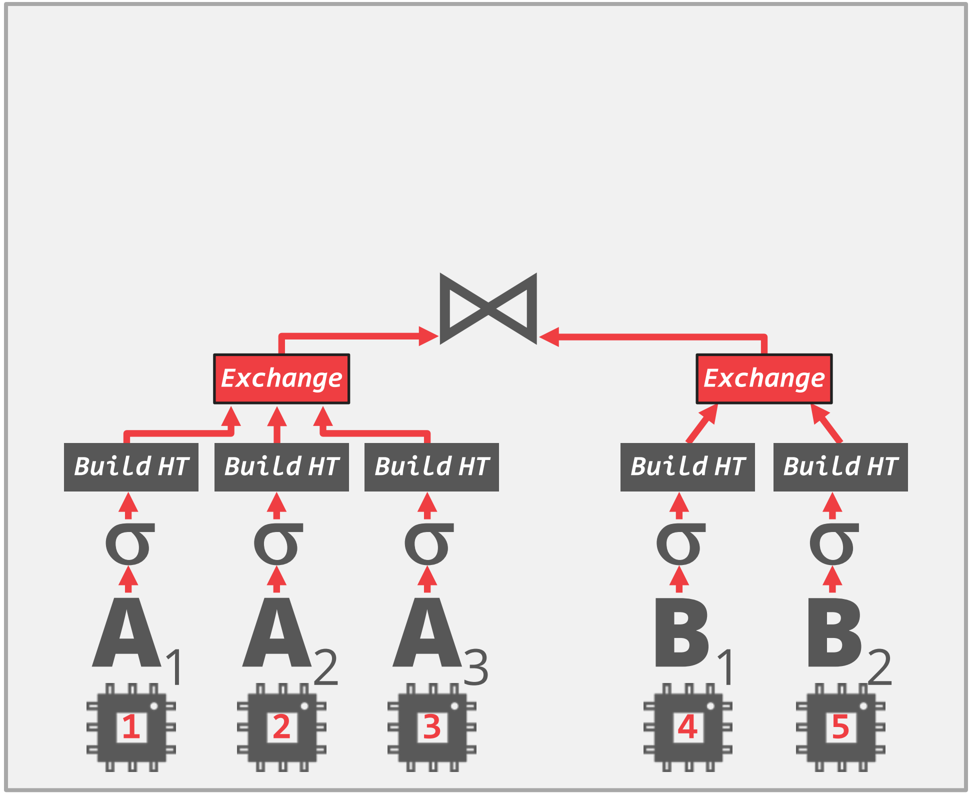


# INTRA-OPERATOR PARALLELISM

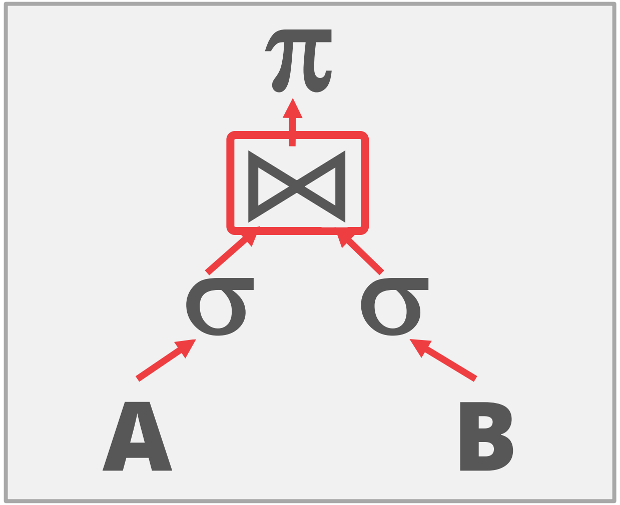
```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

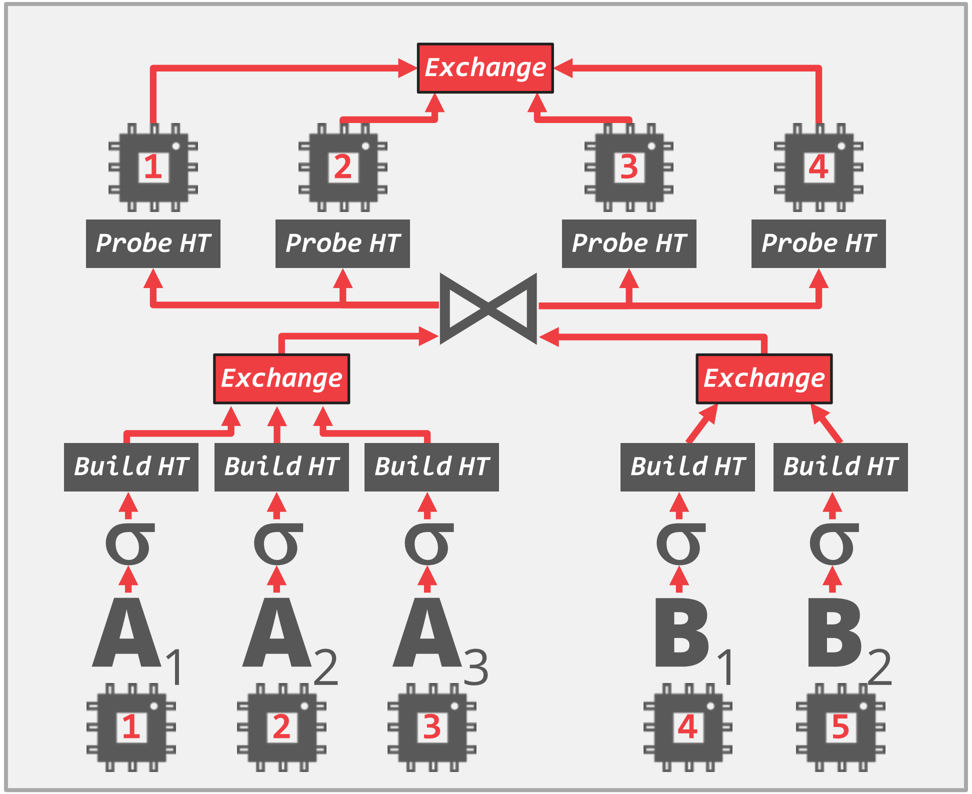


# INTRA-OPERATOR PARALLELISM

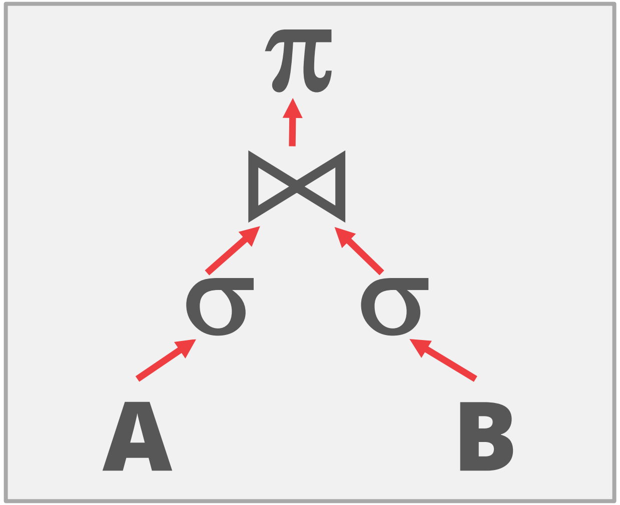
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



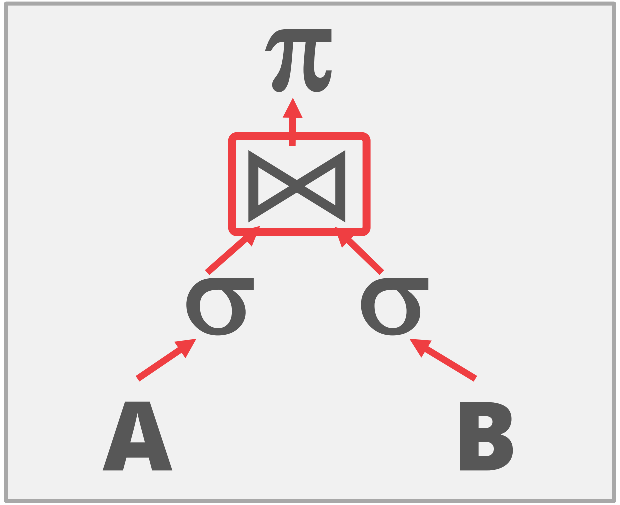
# INTER-OPERATOR PARALLELISM

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



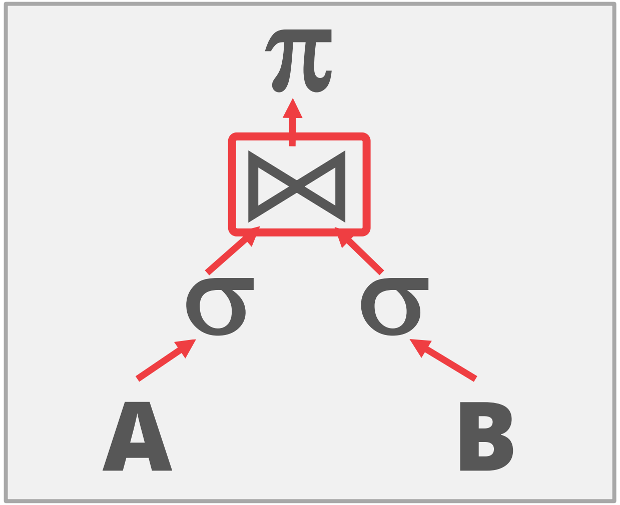
# INTER-OPERATOR PARALLELISM

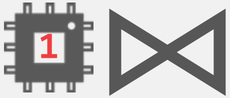
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



# INTER-OPERATOR PARALLELISM

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

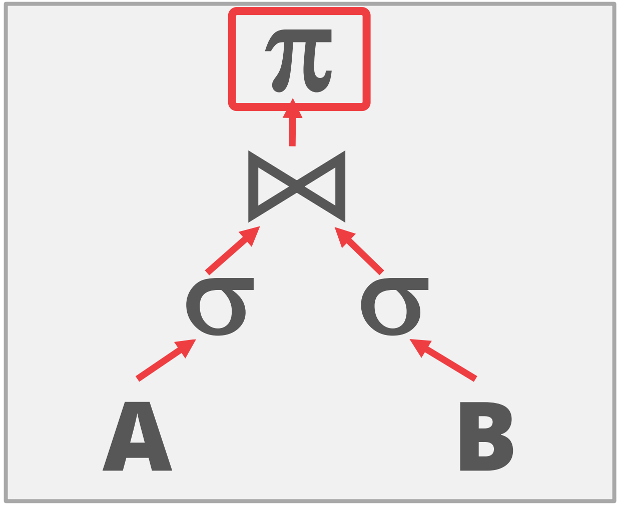


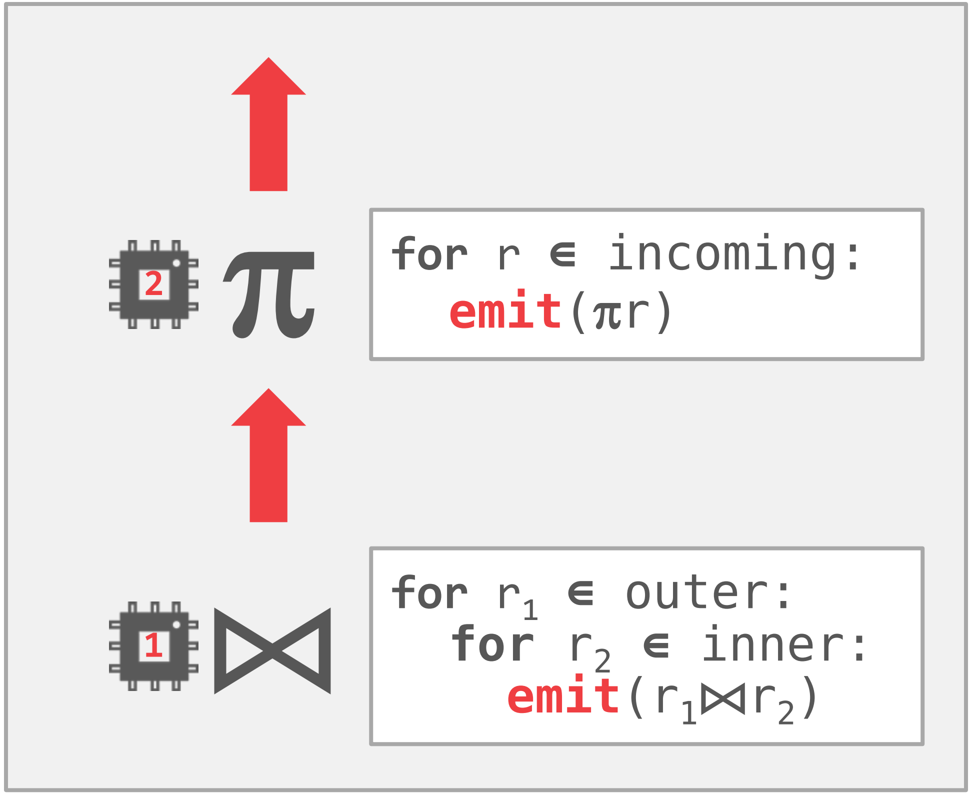




```
for r1 ∊ outer:
  for r2 ∊ inner:
    emit(r1⋈r2)
```

# INTER-OPERATOR PARALLELISM

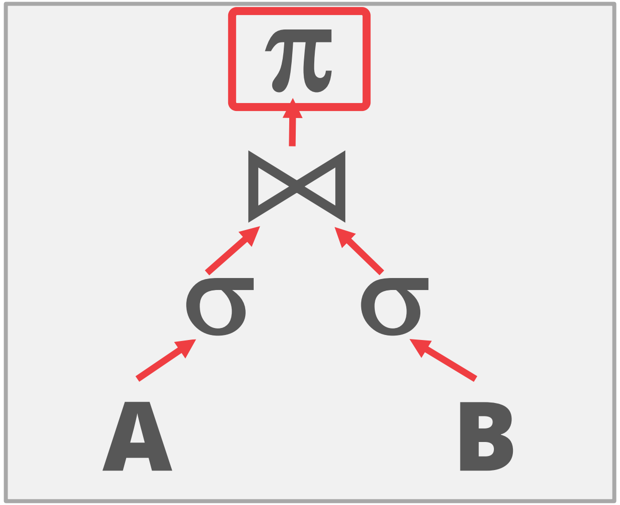
```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```

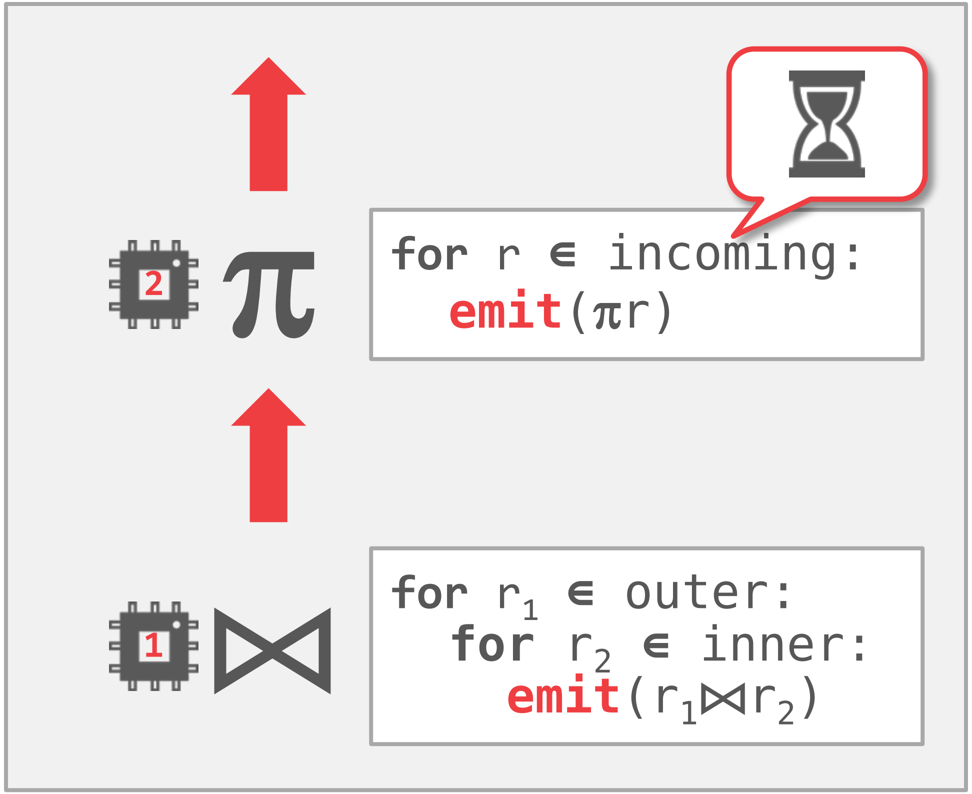


# INTER-OPERATOR PARALLELISM

```
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.

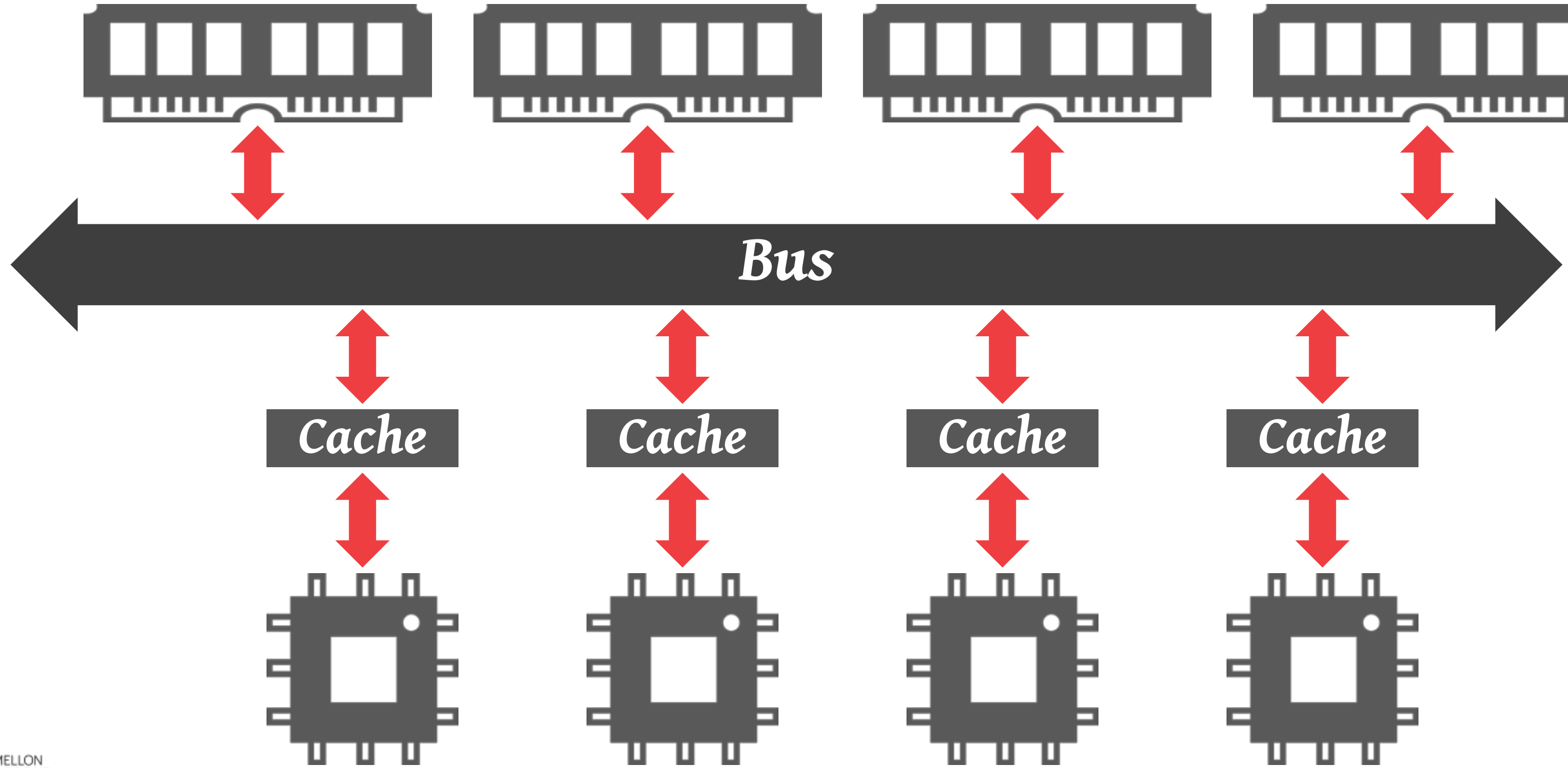
# OBSERVATION

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

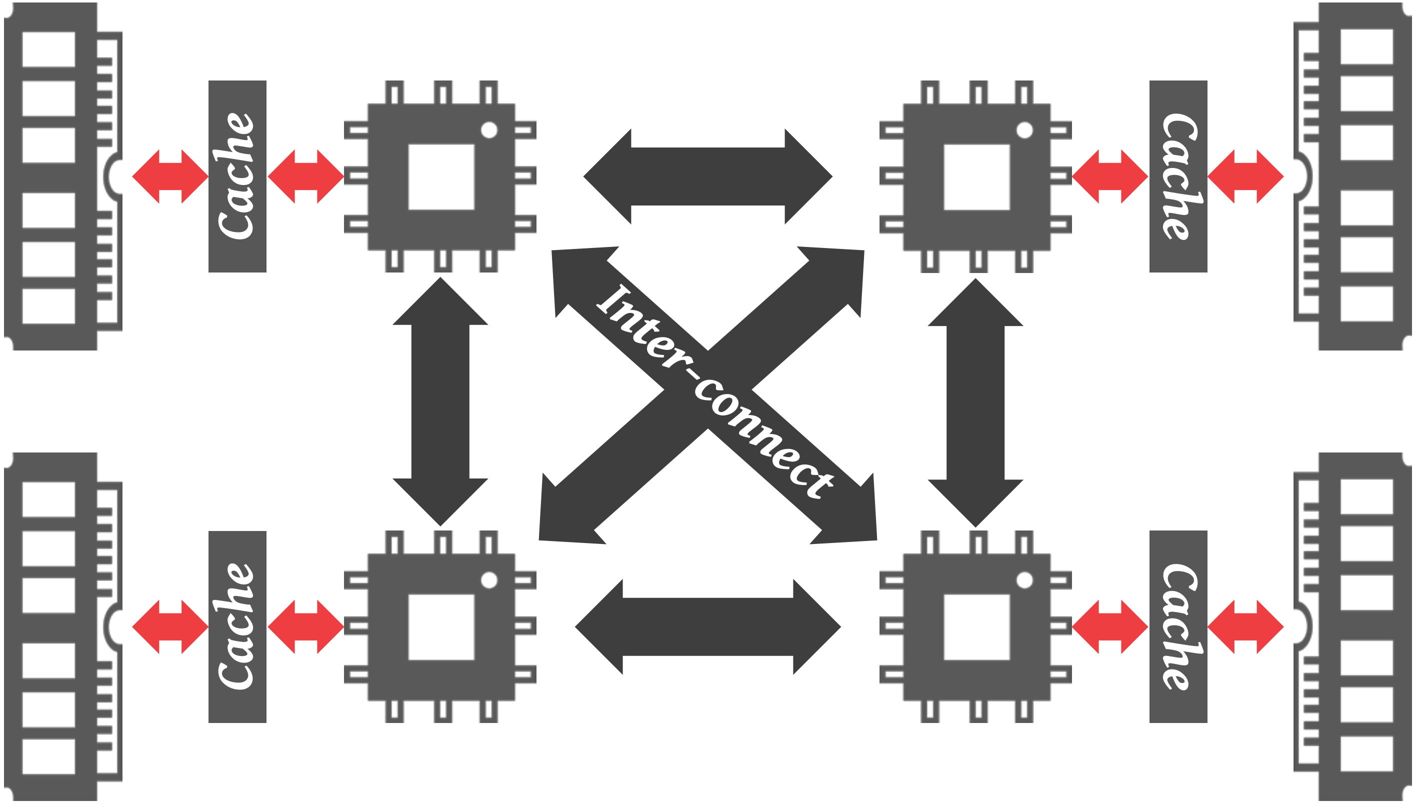
The DBMS's scheduler has to be aware of it's underlying hardware's memory layout.

→ 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`**

# MEMORY ALLOCATION

What happens when the DBMS calls `malloc`?

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

Actually, almost nothing:

# MEMORY ALLOCATION

What happens when the DBMS calls `malloc`?

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

Actually, 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.

# MEMORY ALLOCATION LOCATION

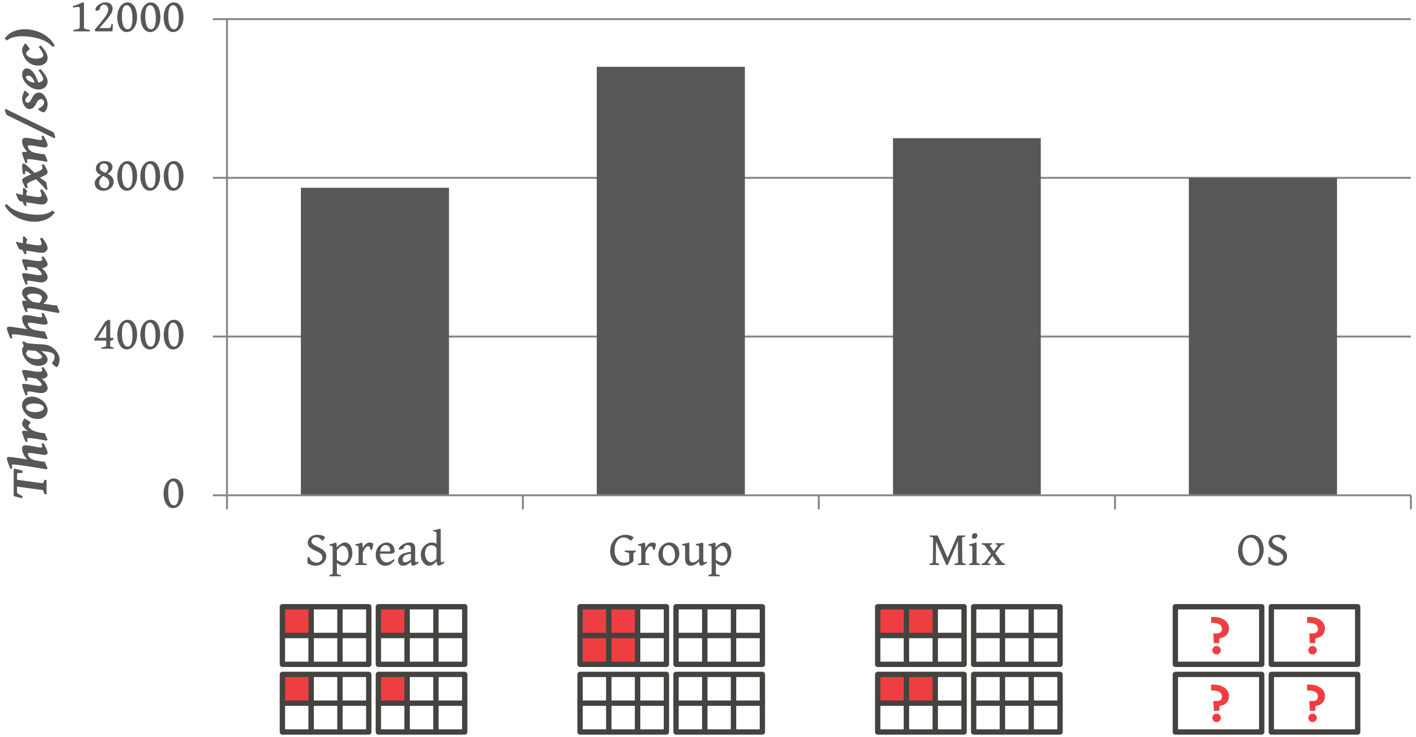
Now after a page fault, where does the OS allocate physical memory in a NUMA system?

**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.

# DATA PLACEMENT

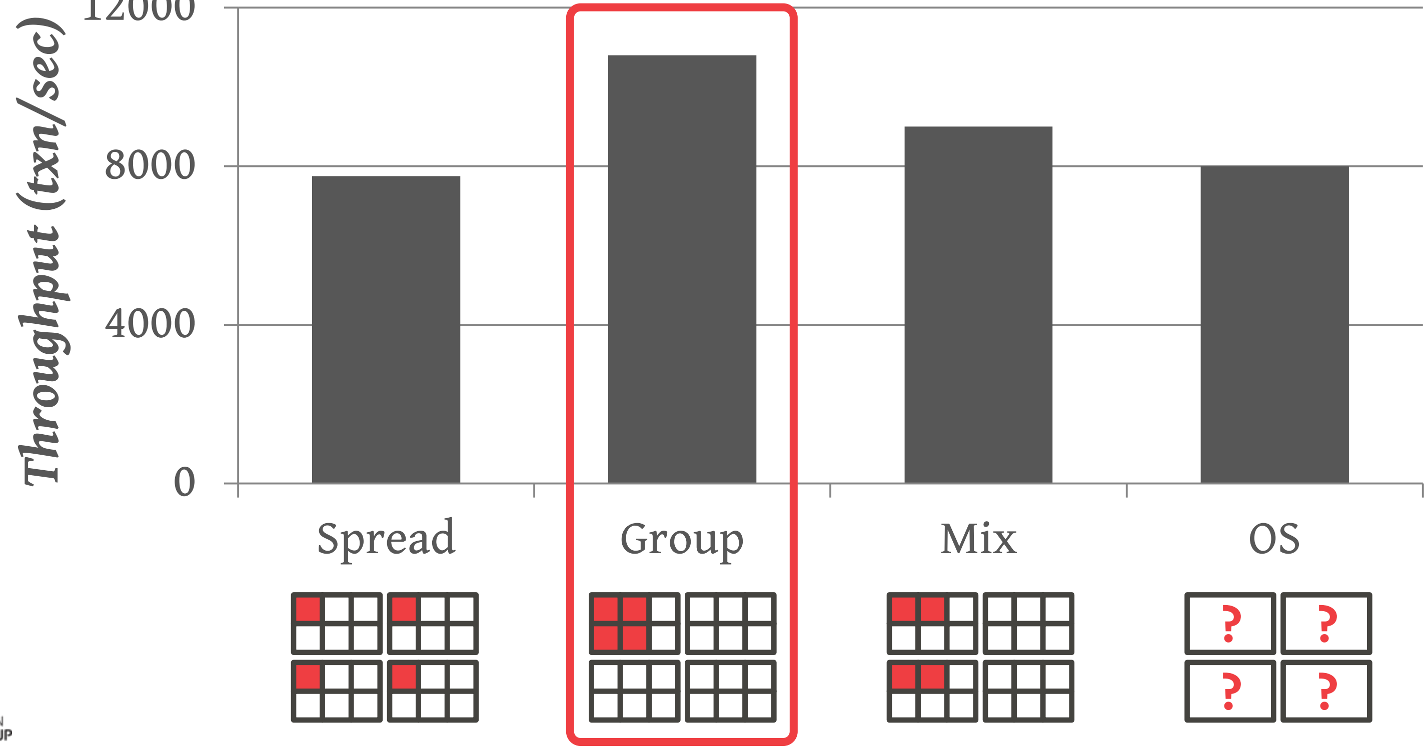
Workload: TPC-C Payment using 4 Workers
Processor: NUMA with 4 sockets (6 cores each)




Source: Danica Porobic

# DATA PLACEMENT

Workload: TPC-C Payment using 4 Workers
Processor: NUMA with 4 sockets (6 cores each)




Source: Danica Porobic

# 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:

→ Process Model
→ Worker Allocation Model
→ Task Assignment Model
→ Data Placement Policy
→ Solid appreciation for the CMU-DB fam.

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 when it generates the plan.

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

→ The easiest approach is to just use the same # of tasks as the # of cores.

# MORSEL-DRIVEN SCHEDULING

Dynamic scheduling of tasks that operate over horizontal partitions called “morsels” that are distributed across cores.

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

Supports parallel, NUMA-aware operator implementations.

MORSEL-DRIVEN PARALLELISM: A NUMA-AWARE QUERY EVALUATION FRAMEWORK FOR THE MANY-CORE AGE
*SIGMOD 2014*

# HYPER: ARCHITECTURE

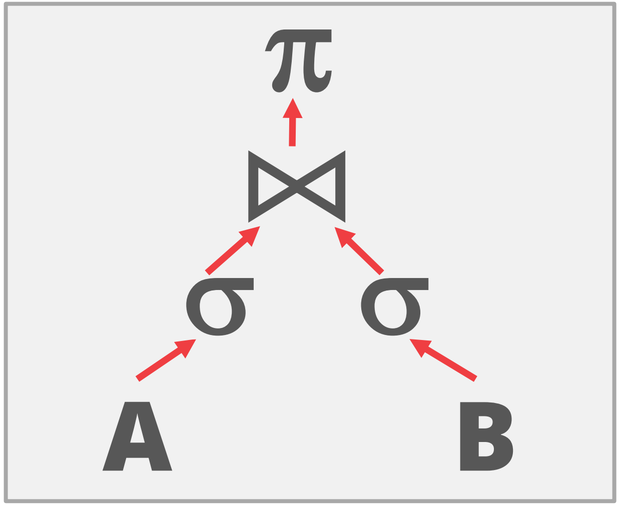
No separate dispatcher thread.

The threads perform cooperative scheduling for each query plan.

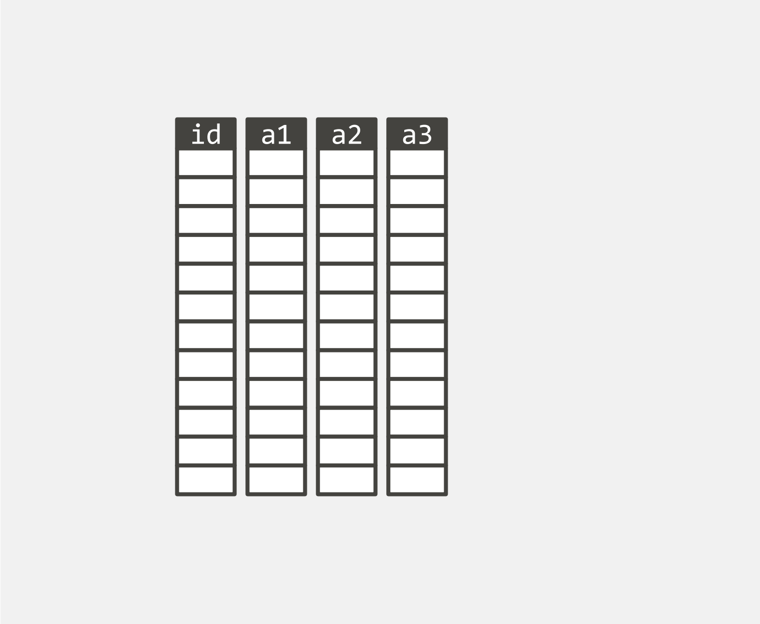
→ Each worker has a queue of tasks that will execute on morsels that are local to it.
→ It pulls the next task from a global work queue.

# HYPER: DATA PARTITIONING

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

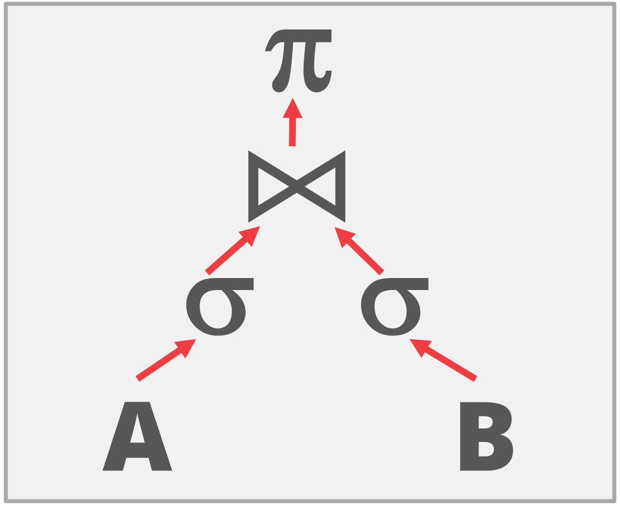



*Data Table*

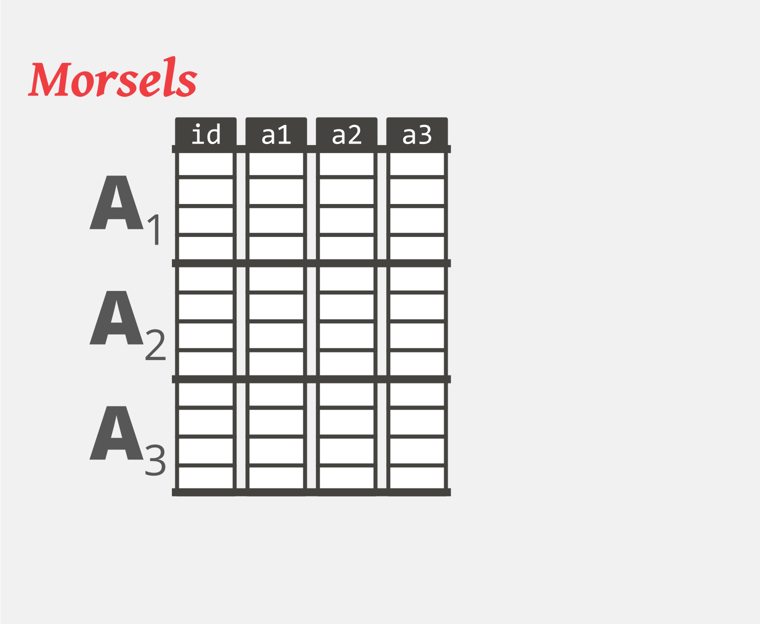


# HYPER: DATA PARTITIONING

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

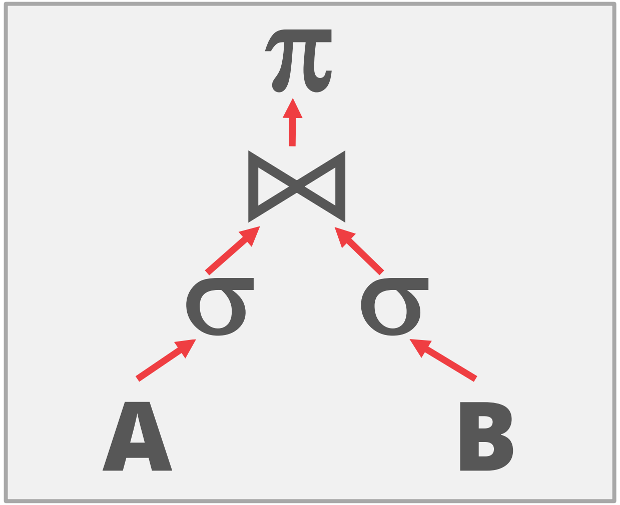



*Data Table*

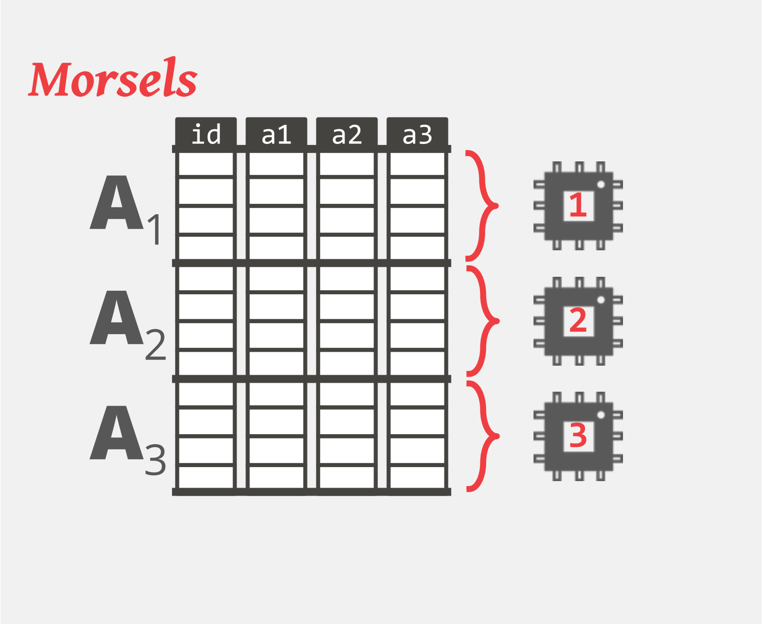


# HYPER: DATA PARTITIONING

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

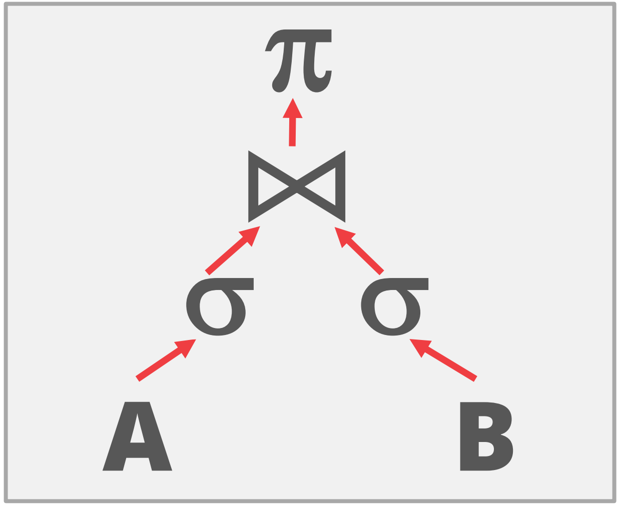



*Data Table*

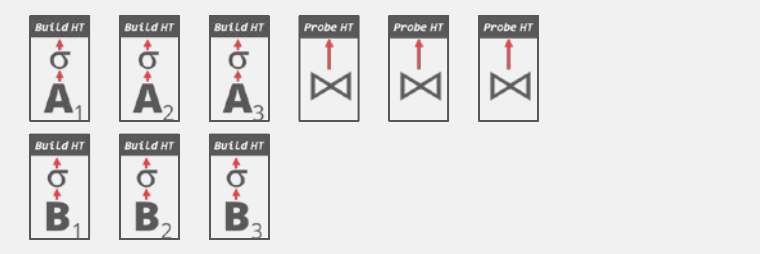


# HYPER: EXECUTION EXAMPLE

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

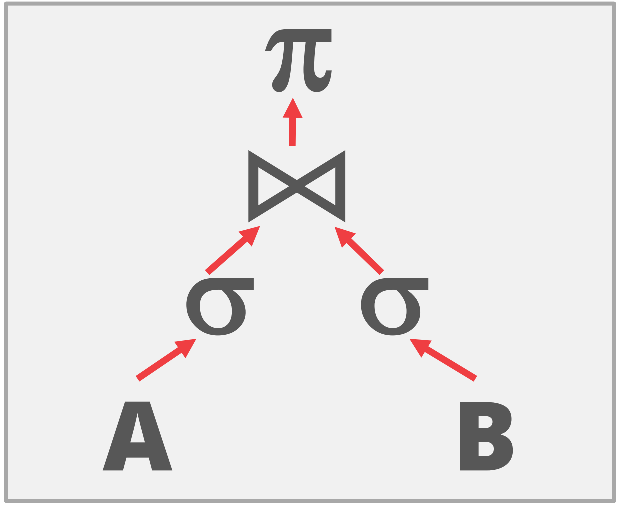



*Task Queues*

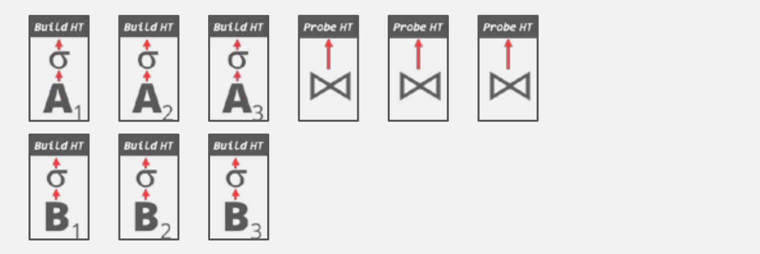


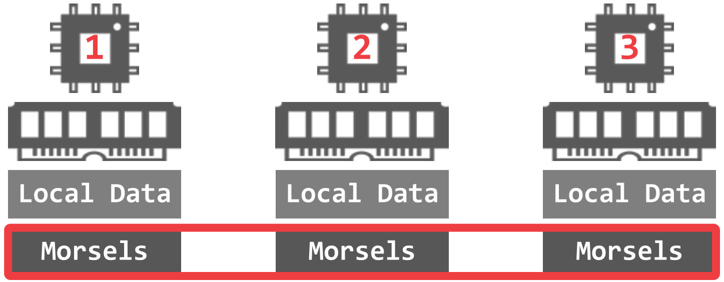

# HYPER: EXECUTION EXAMPLE

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



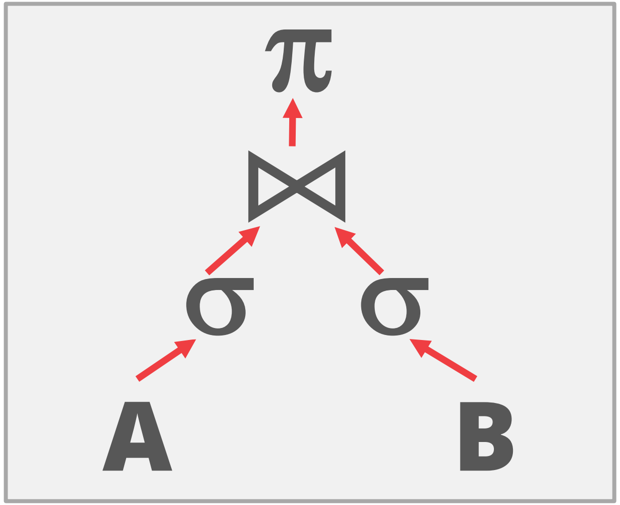

## *Task Queues*



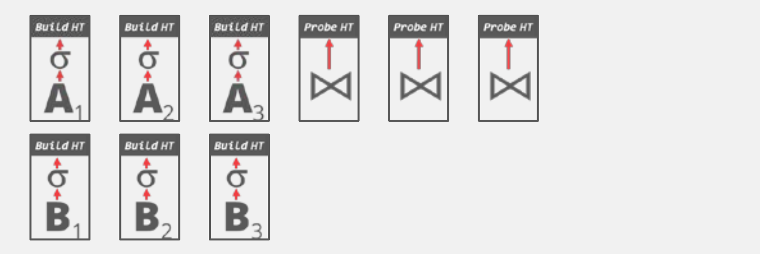



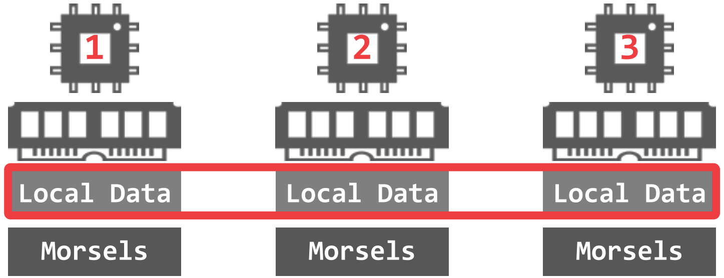
# HYPER: EXECUTION EXAMPLE

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



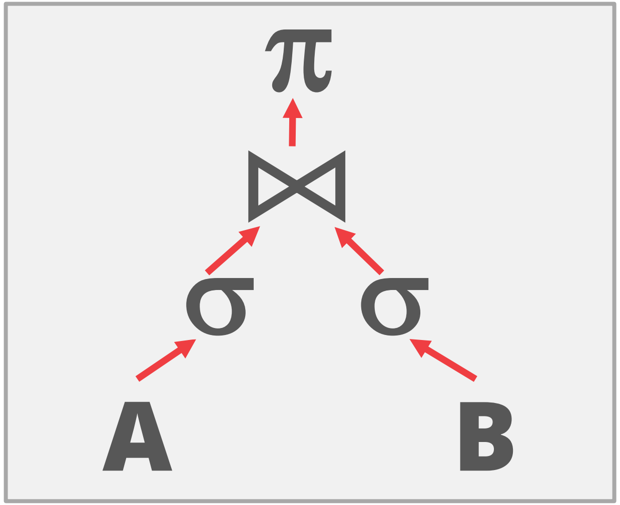

*Task Queues*



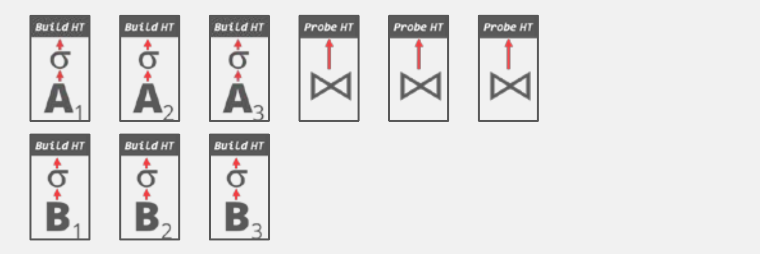



# HYPER: EXECUTION EXAMPLE

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

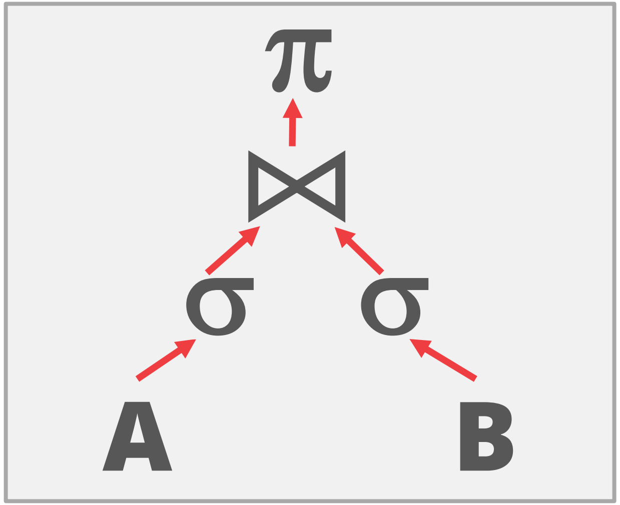



*Task Queues*

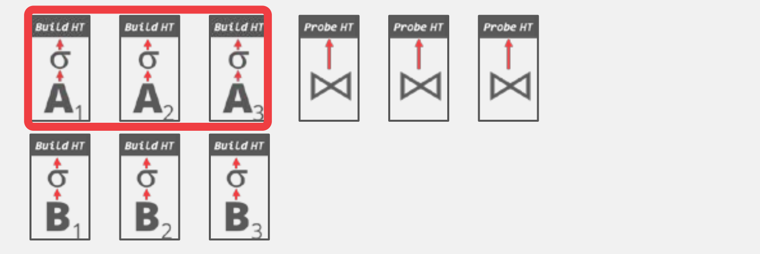



# HYPER: EXECUTION EXAMPLE

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

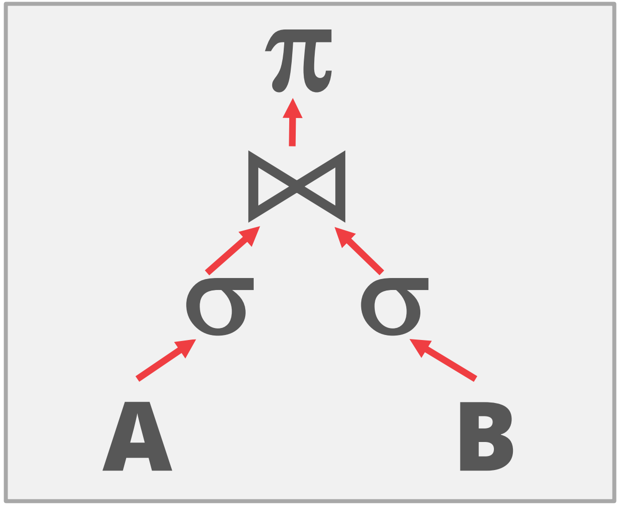



*Task Queues*

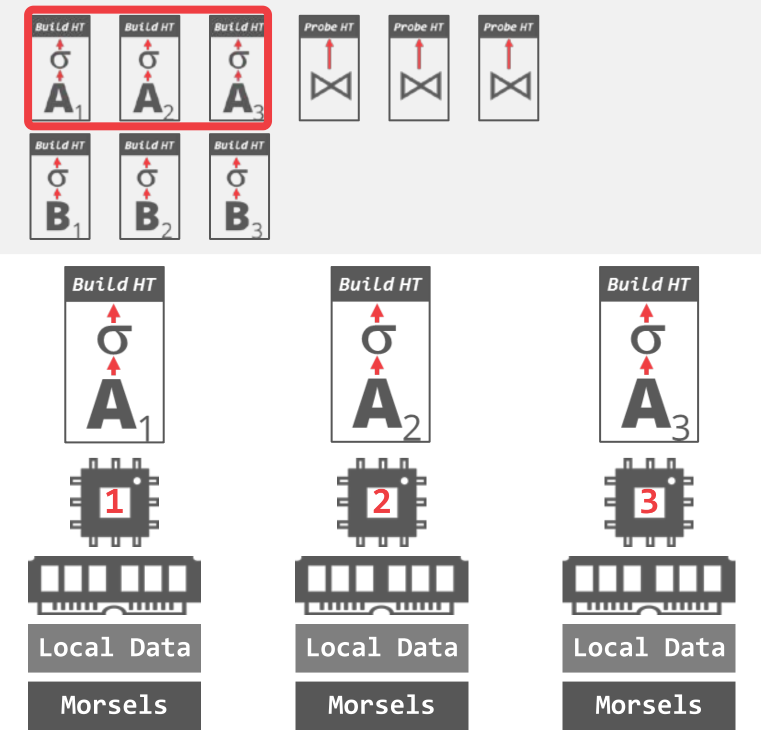



# HYPER: EXECUTION EXAMPLE

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

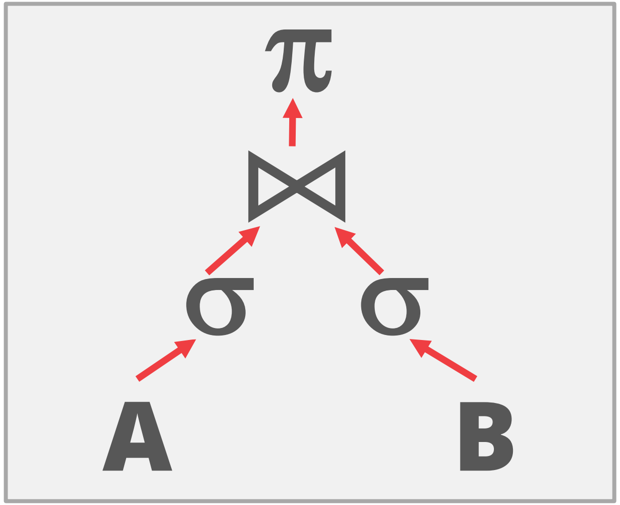


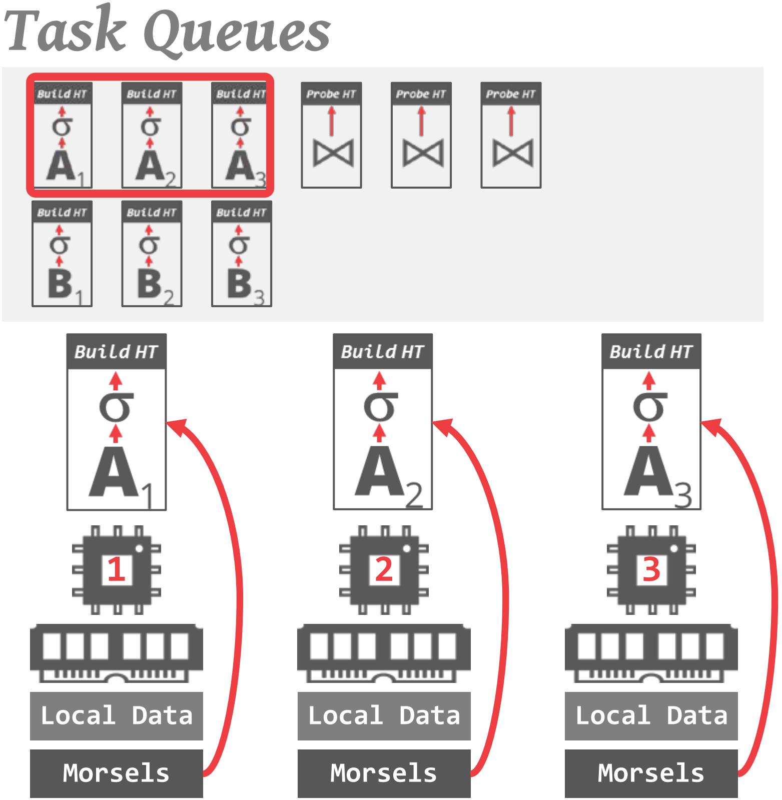

*Task Queues*



# HYPER: EXECUTION EXAMPLE

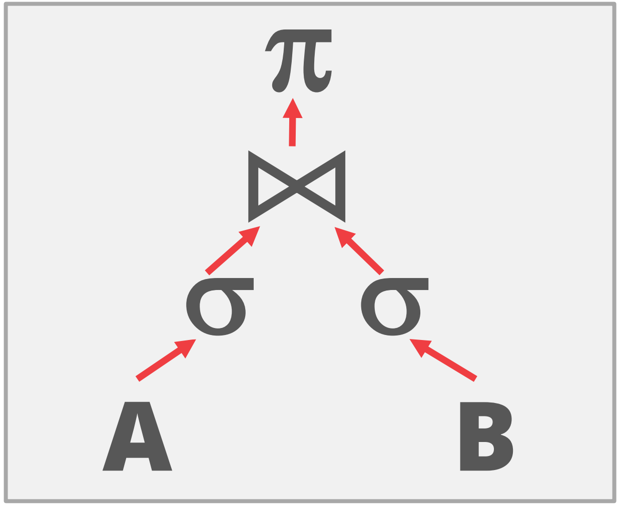
```sql
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



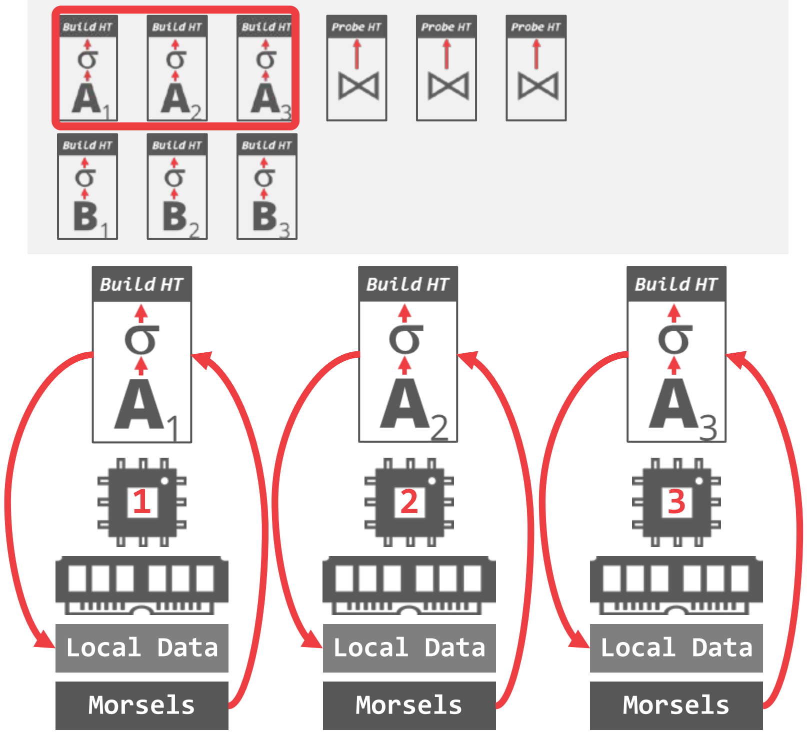



# HYPER: EXECUTION EXAMPLE

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

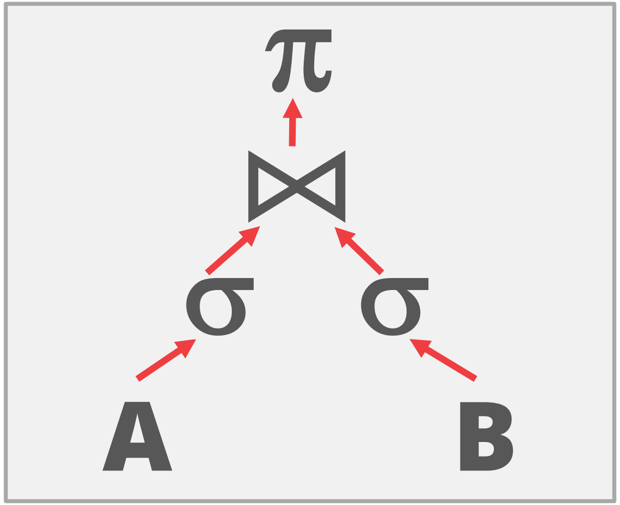



*Task Queues*

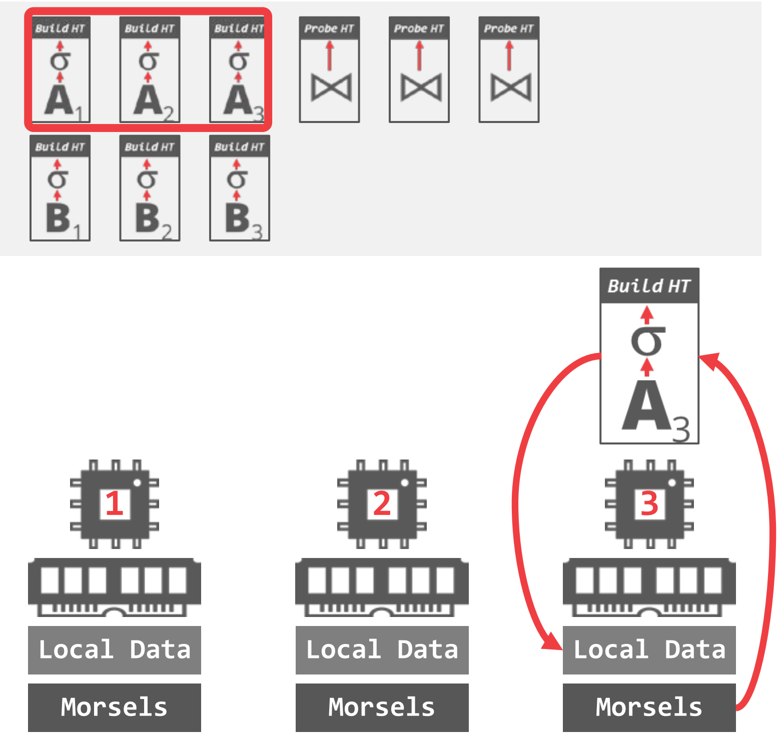


# HYPER: EXECUTION EXAMPLE

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

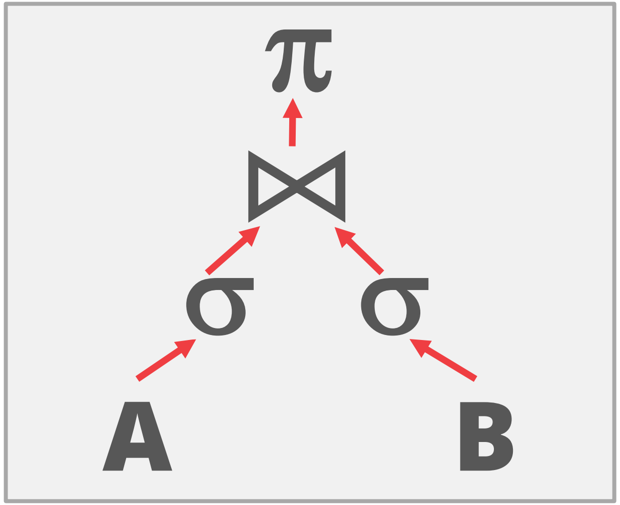


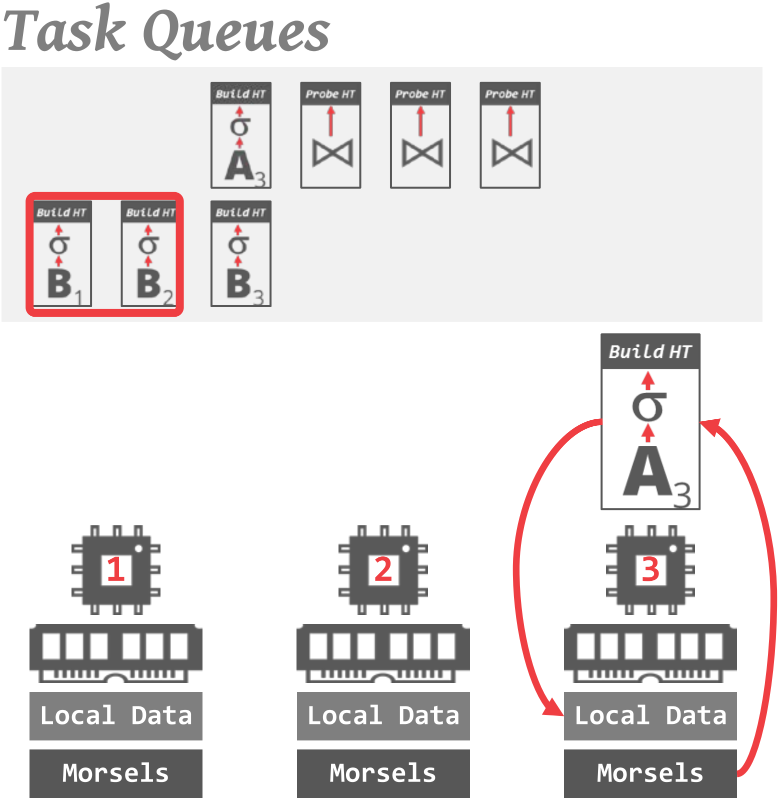

*Task Queues*



# HYPER: EXECUTION EXAMPLE

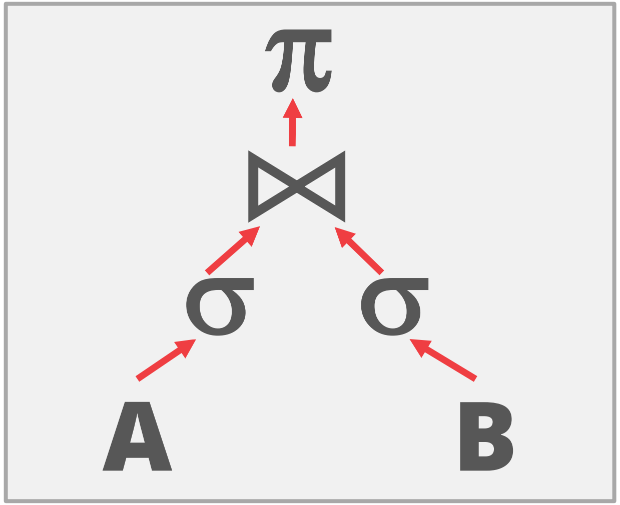
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



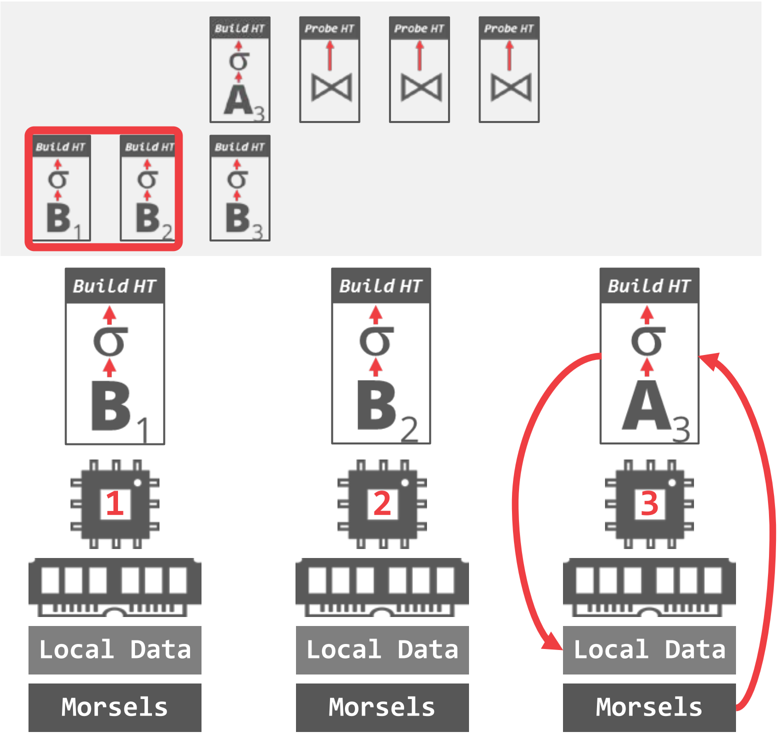



# HYPER: EXECUTION EXAMPLE

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

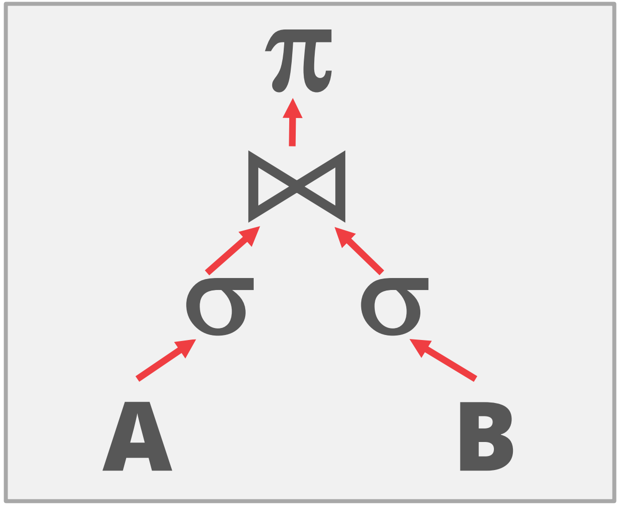



*Task Queues*

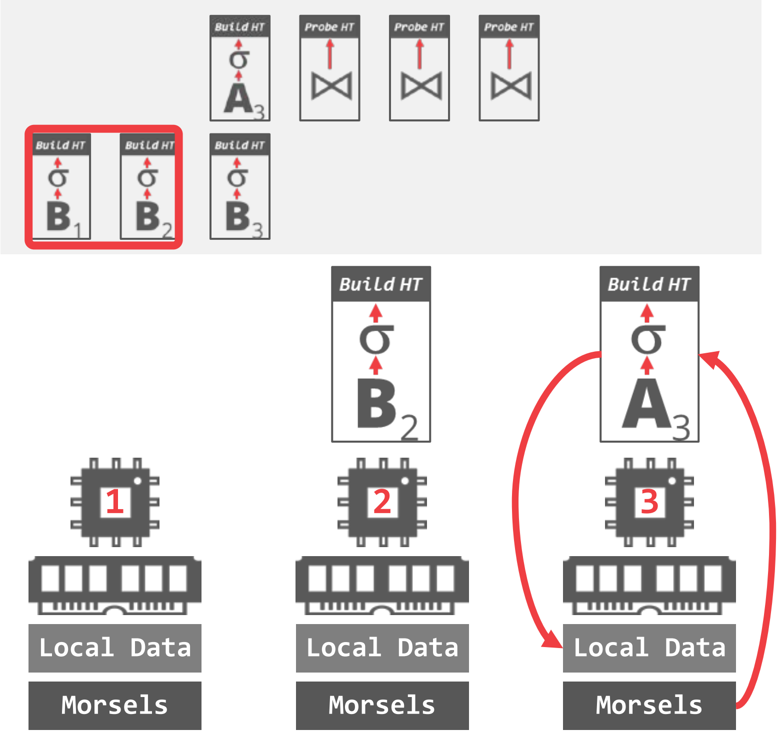


# HYPER: EXECUTION EXAMPLE

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

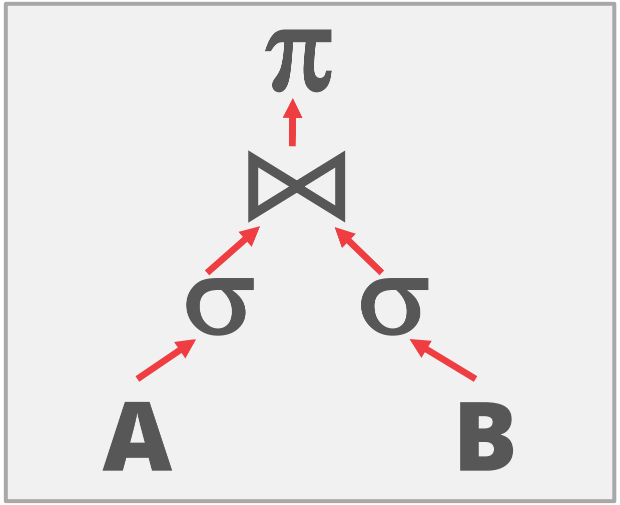



*Task Queues*

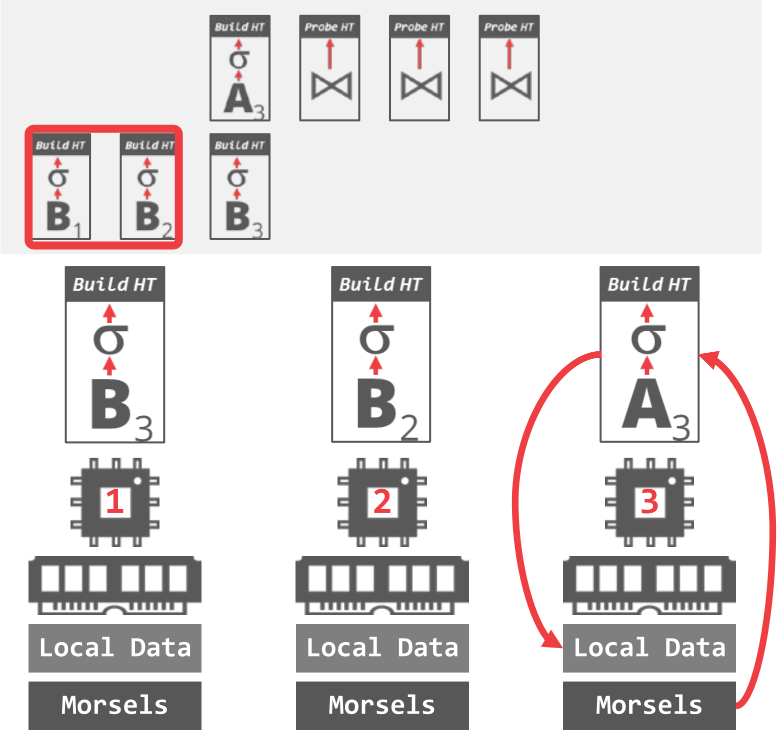


# HYPER: EXECUTION EXAMPLE

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

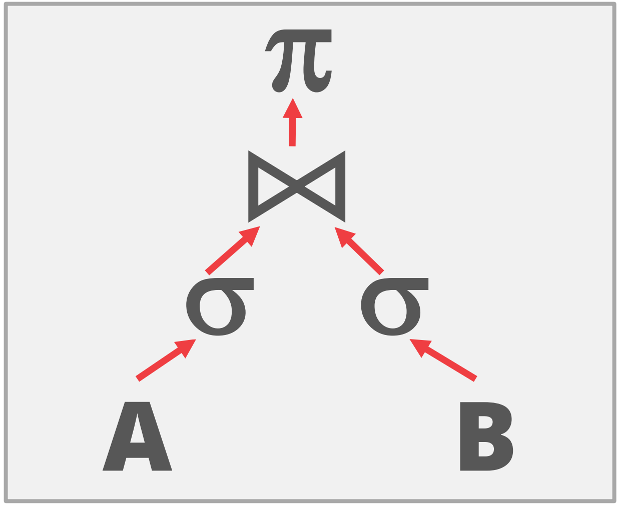


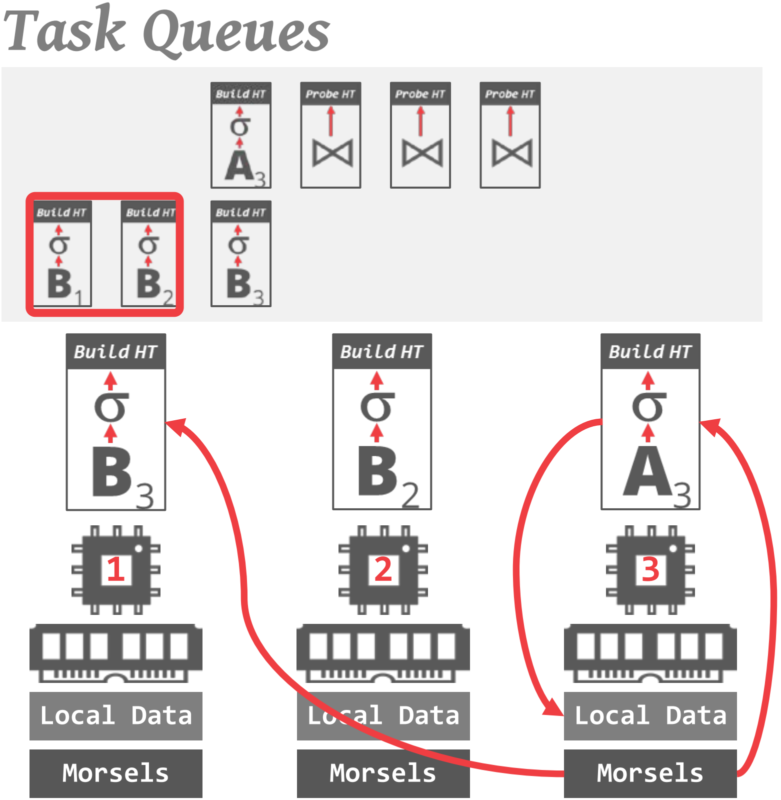

## *Task Queues*



# HYPER: EXECUTION EXAMPLE

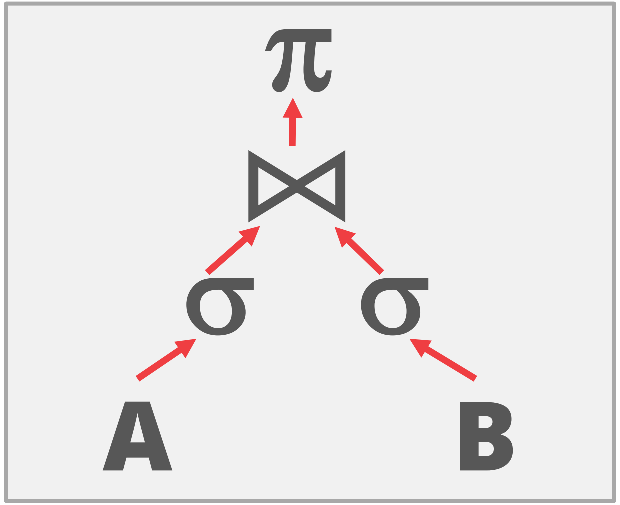
```
SELECT A.id, B.value
  FROM A, B
 WHERE A.id = B.id
   AND A.value < 99
   AND B.value > 100
```



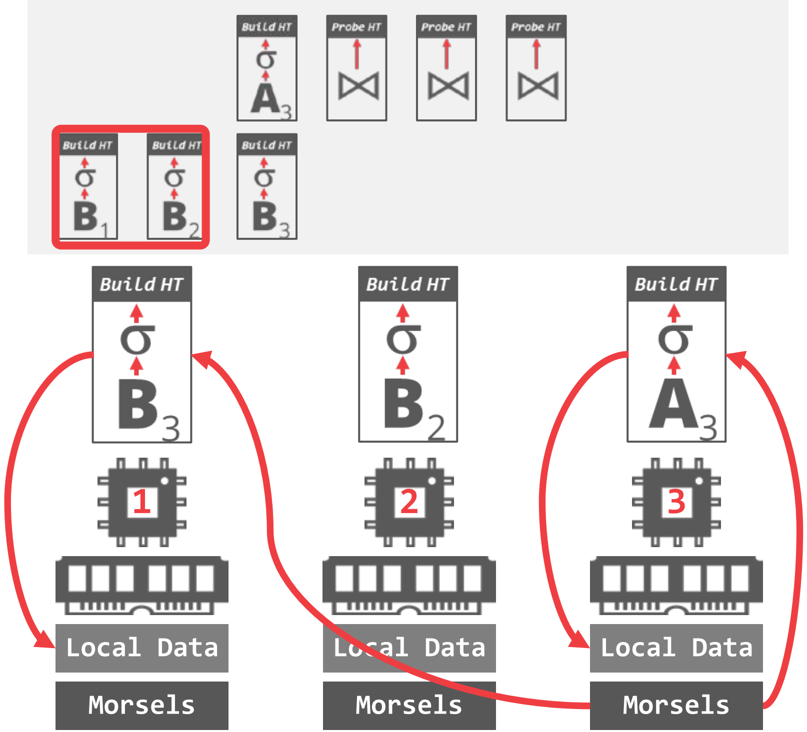



# HYPER: EXECUTION EXAMPLE

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




## *Task Queues*



# MORSEL-DRIVEN SCHEDULING

Because there is only one worker per core, they have to use work stealing because otherwise threads could sit idle waiting for stragglers.

Uses a lock-free hash table to maintain the global work queues.
→ We will discuss hash tables next class...

# HANA NUMA-AWARE SCHEDULER

Pull-based scheduling with multiple worker threads that are organized into groups (pools).

→ Each CPU can have multiple groups.

→ Each group has a soft and hard priority queue.

Uses a separate "watchdog" thread to check whether groups are saturated and can reassign tasks dynamically.

SCALING UP CONCURRENT MAIN-MEMORY COLUMN-STORE SCANS: TOWARDS ADAPTIVE NUMA-AWARE DATA AND TASK PLACEMENT
*VLDB 2015*

# HANA THREAD GROUPS

Each thread group has a soft and hard priority task queues.

→ Threads are allowed to steal tasks from other groups' soft queues.

Four different pools of thread per group:

→ **Working**: Actively executing a task.
→ **Inactive**: Blocked inside of the kernel due to a latch.
→ **Free**: Sleeps for a little, wake up to see whether there is a new task to execute.
→ **Parked**: Like free but doesn't wake up on its own.

# HANA NUMA-AWARE SCHEDULER

Can dynamically adjust thread pinning based on whether a task is CPU or memory bound.

Found that work stealing was not as beneficial for systems with a larger number of sockets.

Using thread groups allows cores to execute other tasks instead of just only queries.

# PARTING THOUGHTS

A DBMS is a beautiful, strong-willed independent piece of software.

But it has to make sure that it uses its underlying hardware correctly.

→ Data location is an important aspect of this.
→ Tracking memory location in a single-node DBMS is the same as tracking shards in a distributed DBMS

Don't let the OS ruin your life.

# 10 CRACK COMMANDMENTS

In 1997, Christopher Wallace wrote a prophetic list of rules to follow if you are hustling product out on the streets.

Wallace

Almost 20 years later, these rules are still apt for both trapping and databases.

*10 CRACK COMMANDMENTS*
*BAD BOY RECORDS 1997*

# 10 CRACK COMMANDMENTS

1. Never let people know how much money you have.
2. Never let people know your next move.
3. Never trust anybody.
4. Never sample your own supply.
5. Never sling where you live.
6. Never lend anybody credit.
7. Never mix your family with your business affairs.
8. Never keep a large amount of product on yourself.
9. Never talk to the police.
10. Never take a consignment if you do not have the clientele.

# NEXT CLASS

Parallel Hash Joins!

**Project #2 Checkpoint: Monday Feb 22**