ADVANCED
DATABASE
SYSTEMS

Query Scheduling & Coordination

08

Andy Pavlo
CMU 15-721
Spring 2024

Carnegie Mellon University

LAST CLASS

Query Code Generation & Compilation
→ **Approach #1: Transpilation / Source-to-Source**
→ **Approach #2: JIT Compilation**

Most DBMSs choose **vectorization** with SIMD over **JIT compilation** to accelerate query execution due to engineering reasons.

QUERY EXECUTION

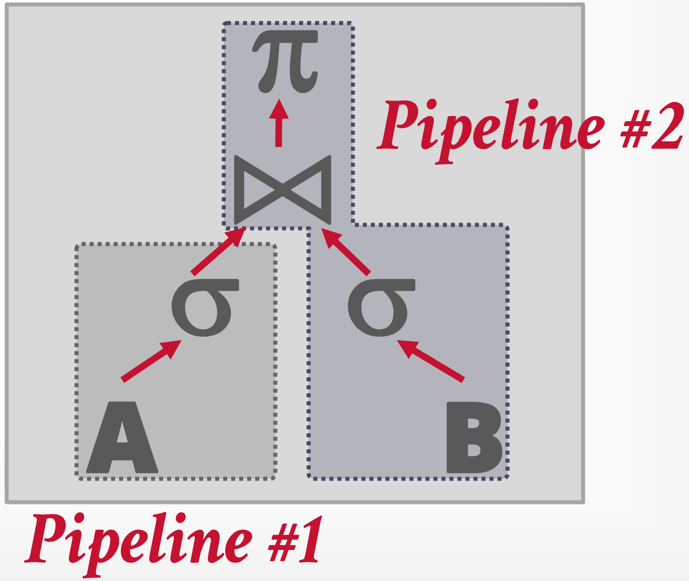
A query plan is a DAG of **operators**.

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

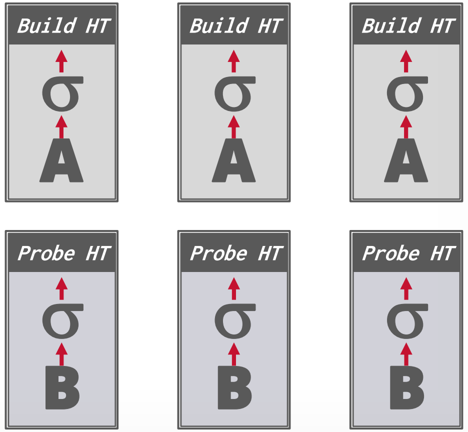
A query plan is a DAG of **operators**.

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.

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



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.

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.

TODAY'S AGENDA

Worker Allocation

Data Placement

Scheduler Implementations

Distributed Query Scheduling

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.

ARCHITECTURE OF A DATABASE SYSTEM
FOUNDATIONS AND TRENDS IN DATABASES 2007

WORKER ALLOCATION

Approach #1: One Worker per CPU Core

→ The DBMS pins a worker to a single core via OS syscalls.
→ See `sched_setaffinity`

Approach #2: Multiple Workers per CPU 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 must be aware of the location of data and each node's memory layout.

- → Attached Cached vs. Nearby Cache vs. Remote Storage
- → Uniform vs. Non-Uniform Memory Access

PARTITIONING VS. PLACEMENT

A **partitioning** scheme is used to split the database based on some policy and target objective.
→ Round-robin (e.g., at the file level)
→ Attribute Ranges
→ Hashing
→ Partial/Full Replication

A **placement** scheme then tells the DBMS where to put those partitions.
→ Round-robin
→ Interleave across nodes / workers

OBSERVATION

We have the following so far:
- → Task Assignment Model
- → Data Placement Policy

But how does the DBMS decide which tasks to create 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 <u>not</u> 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.

Slower tasks (**<u>stragglers</u>**) will hurt query latency because dependent tasks must wait until that pipeline completes.

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.

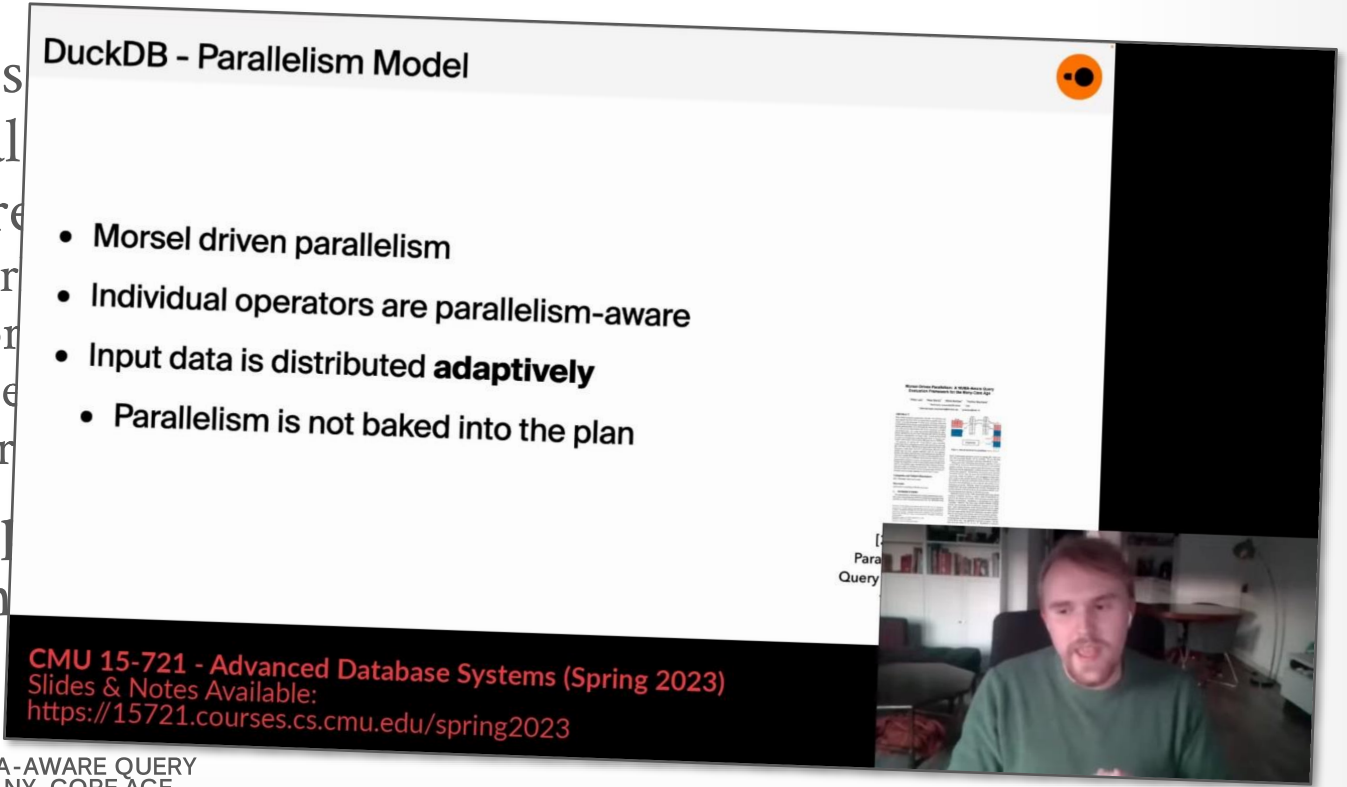
Supports parallel, NUMA-aware operator implementations.

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

MORSEL-DRIVEN SCHEDULING

Dynamic s
horizontal
across core
→ One wor
→ One mor
→ Pull-base
→ Round-r

Supports
implemen



DuckDB - Parallelism Model

- Morsel driven parallelism
- Individual operators are parallelism-aware
- Input data is distributed **adaptively**
  - Parallelism is not baked into the plan

CMU 15-721 - Advanced Database Systems (Spring 2023)
Slides & Notes Available:
https://15721.courses.cs.cmu.edu/spring2023

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

HYPER: ARCHITECTURE

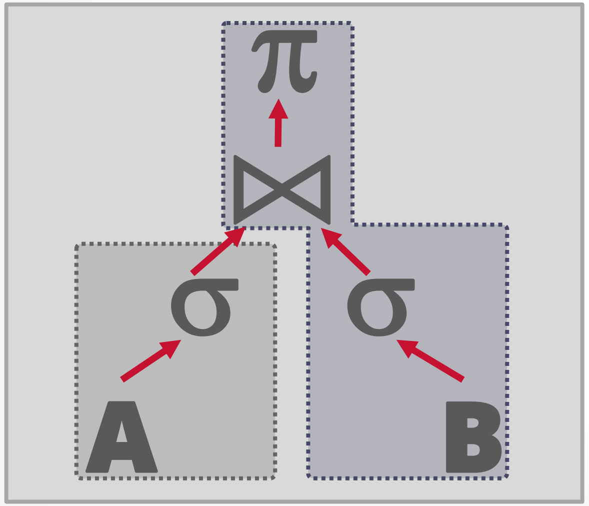
No separate dispatcher thread.

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

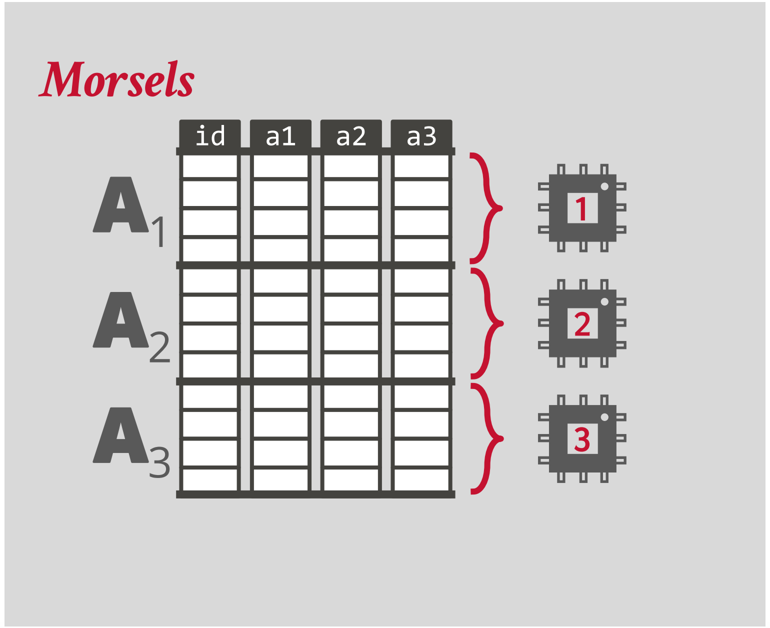
→ Each worker tries to select tasks that will execute on morsels that are local to it.
→ If there are no local tasks, then the worker just pulls the next task from the global work queue.

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
```

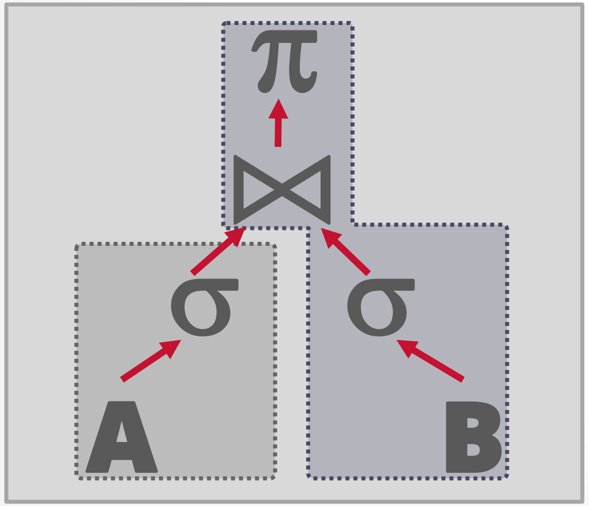


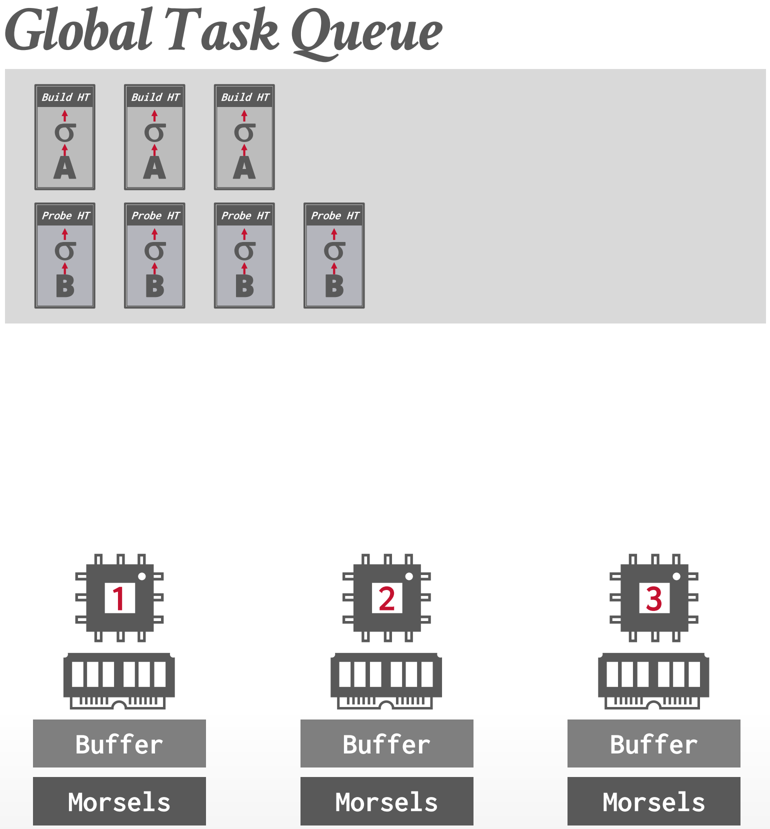
Data Table



HYPER: EXECUTION EXAMPLE

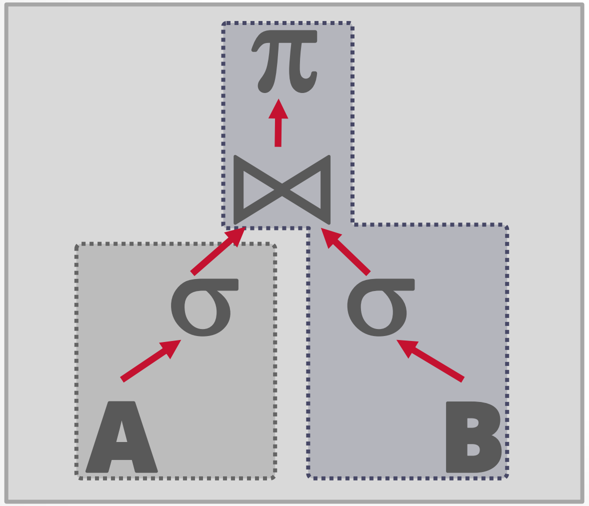
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

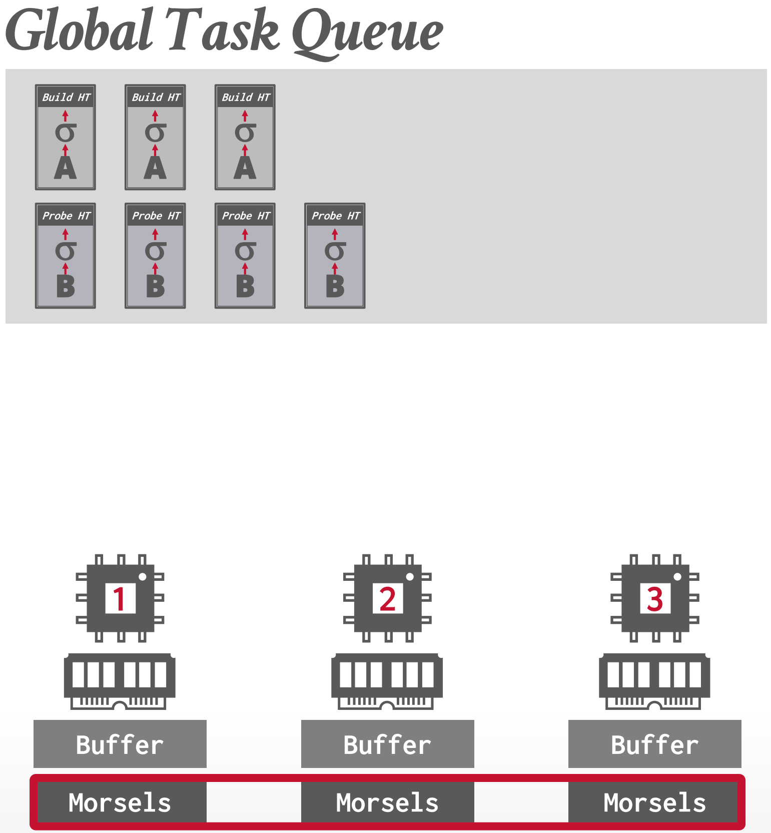




HYPER: EXECUTION EXAMPLE

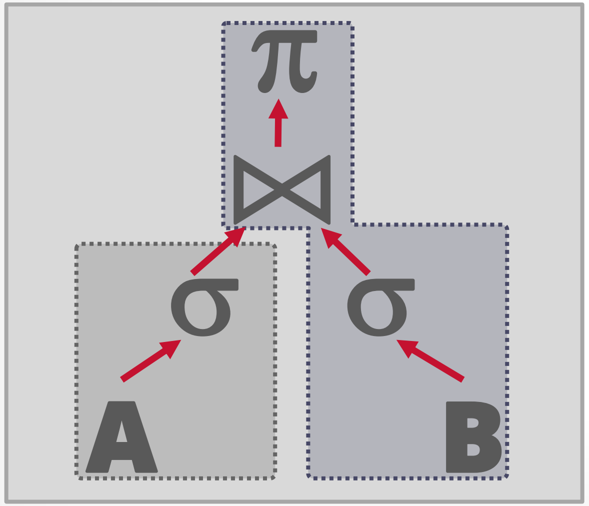
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



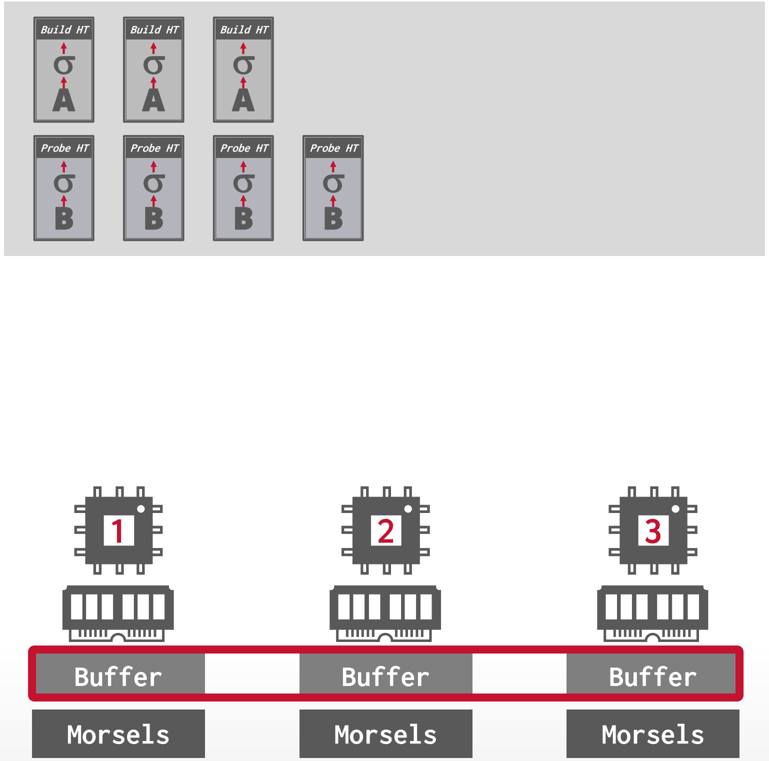


HYPER: EXECUTION EXAMPLE

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

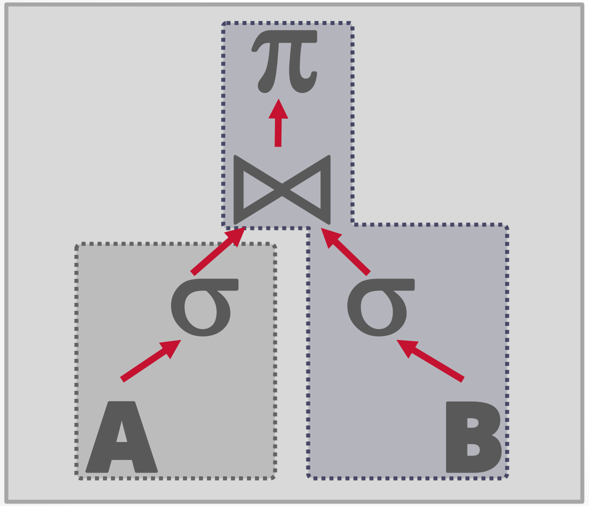


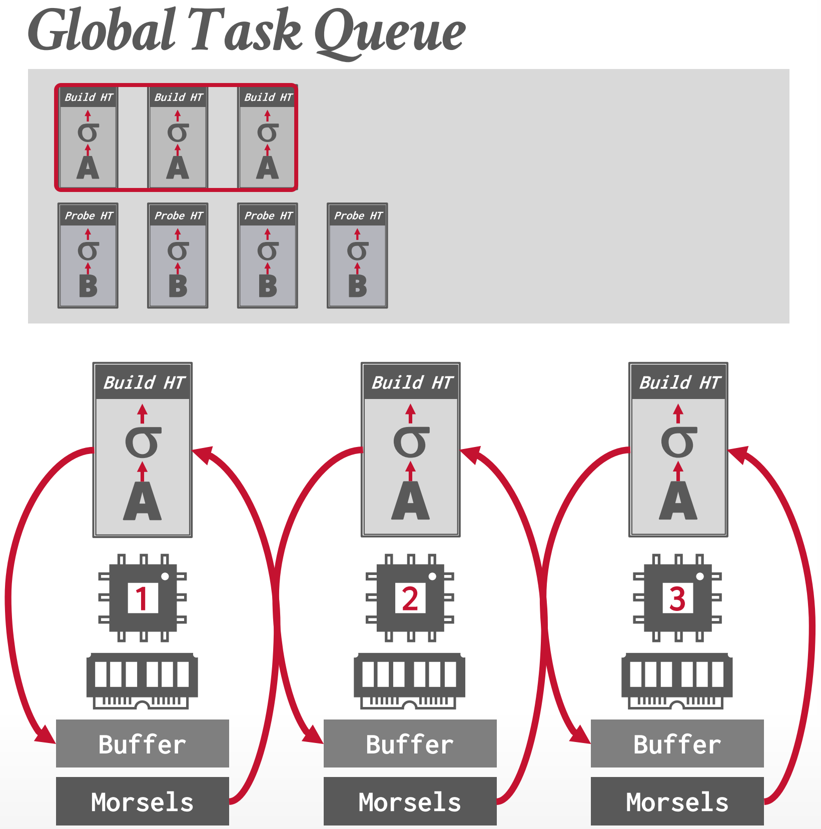
Global Task Queue



HYPER: EXECUTION EXAMPLE

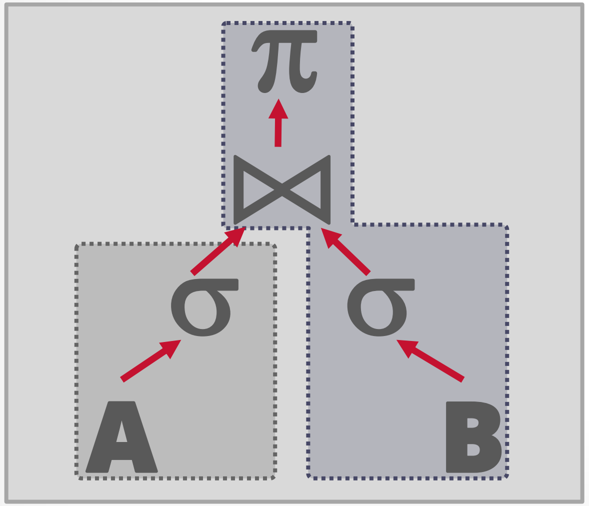
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

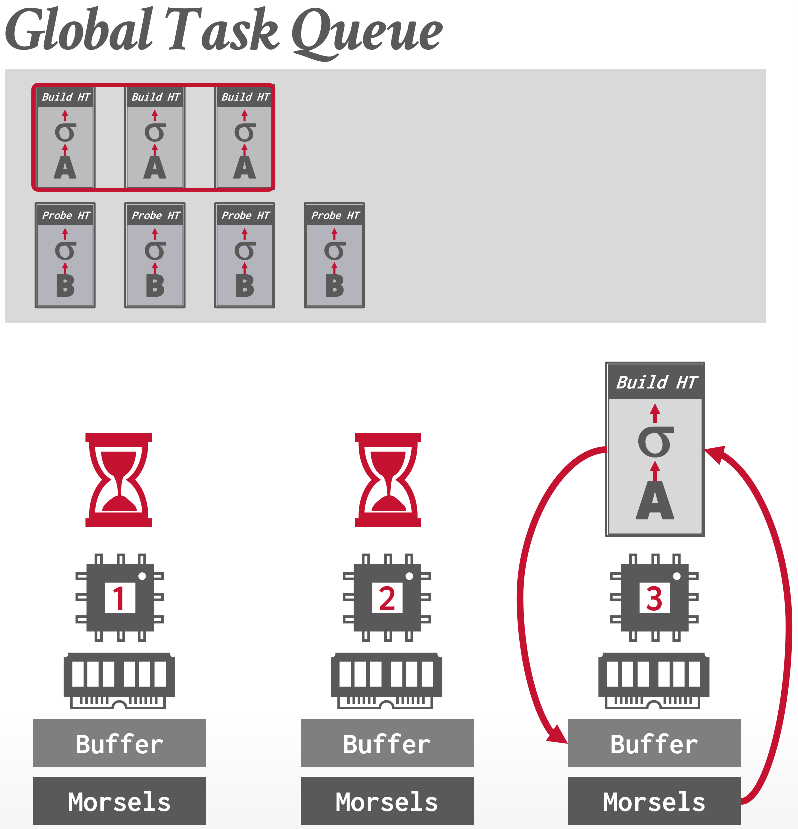




HYPER: EXECUTION EXAMPLE

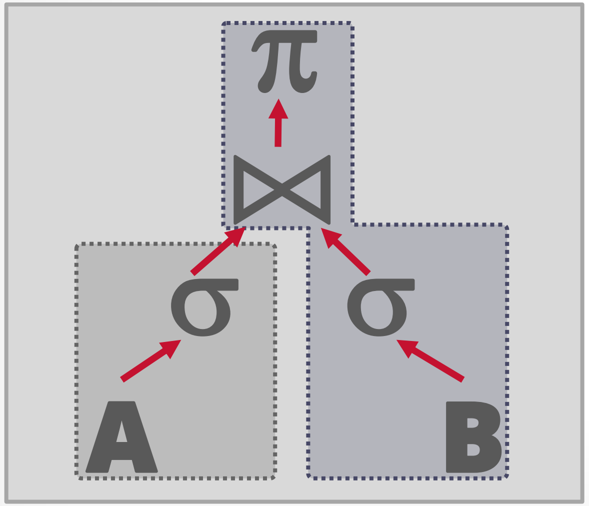
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

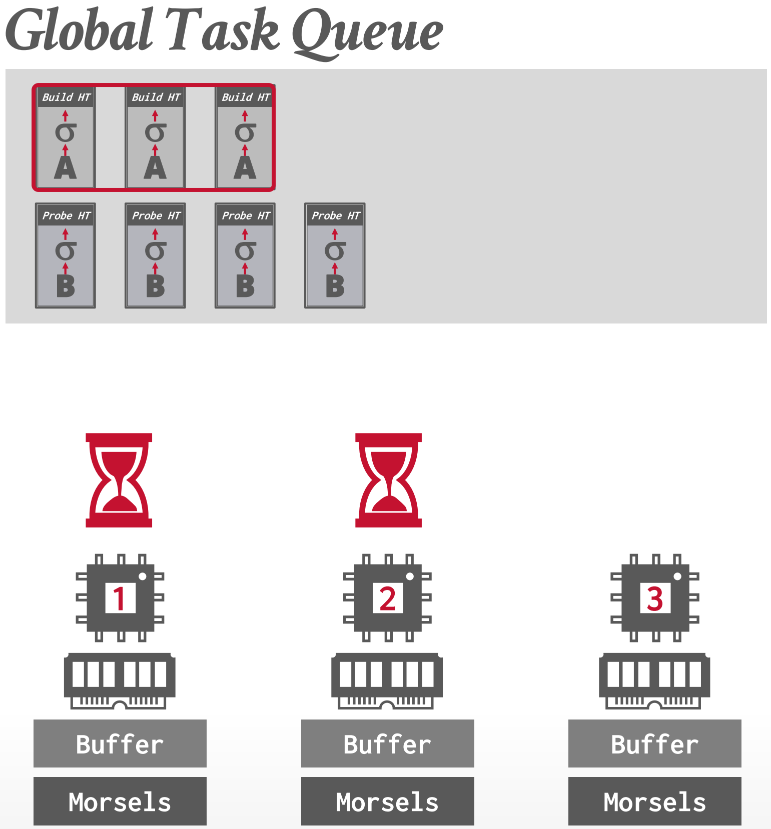




HYPER: EXECUTION EXAMPLE

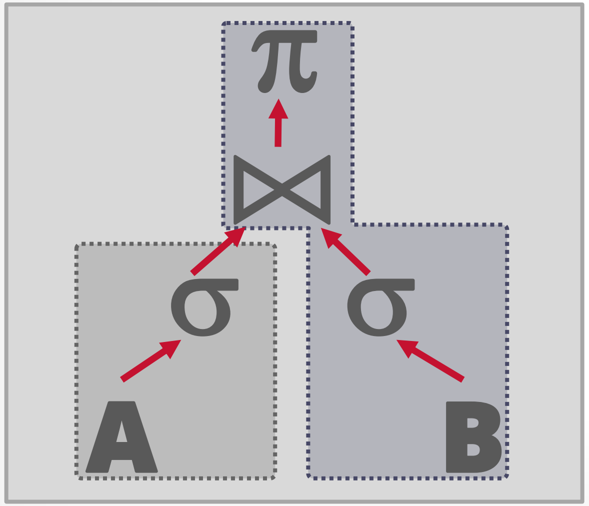
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```



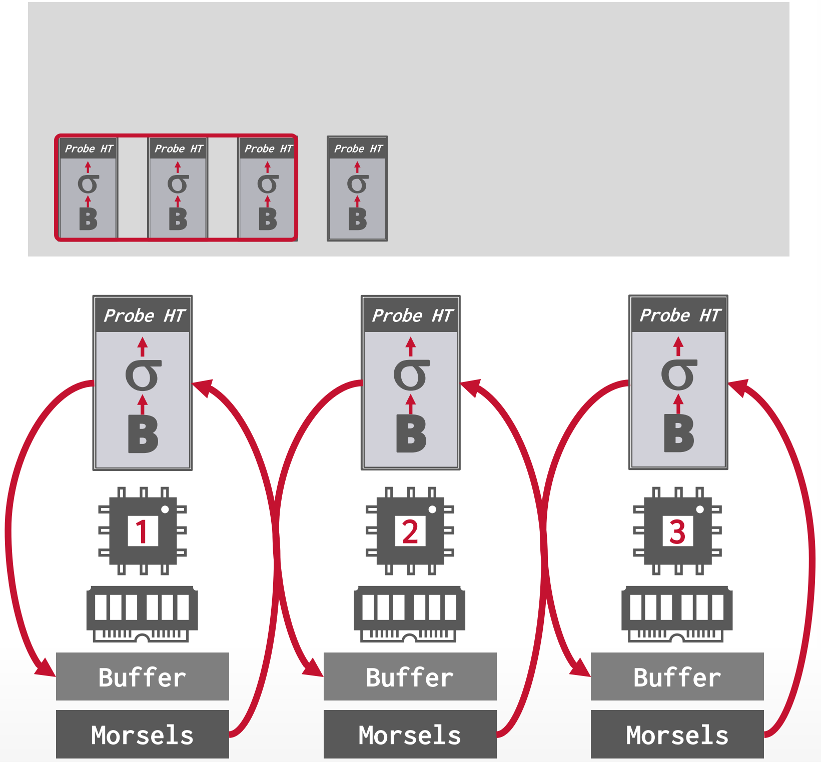


HYPER: EXECUTION EXAMPLE

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

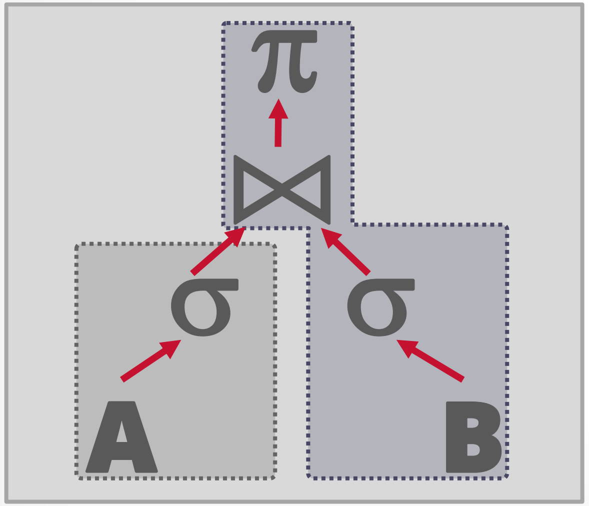


Global Task Queue

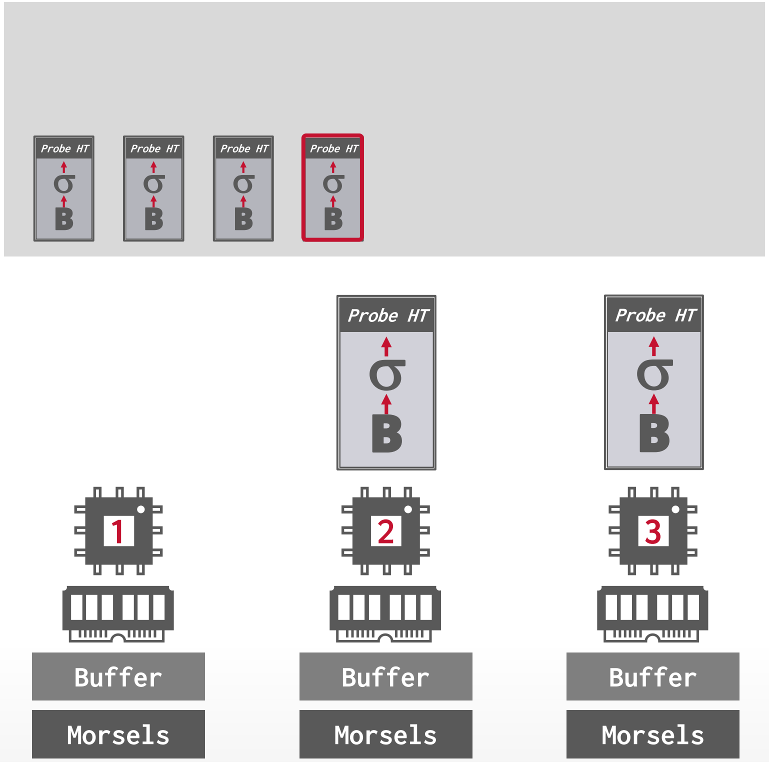


HYPER: EXECUTION EXAMPLE

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

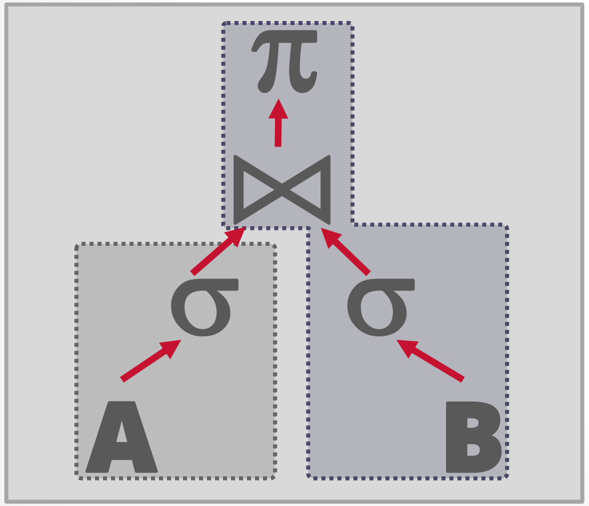


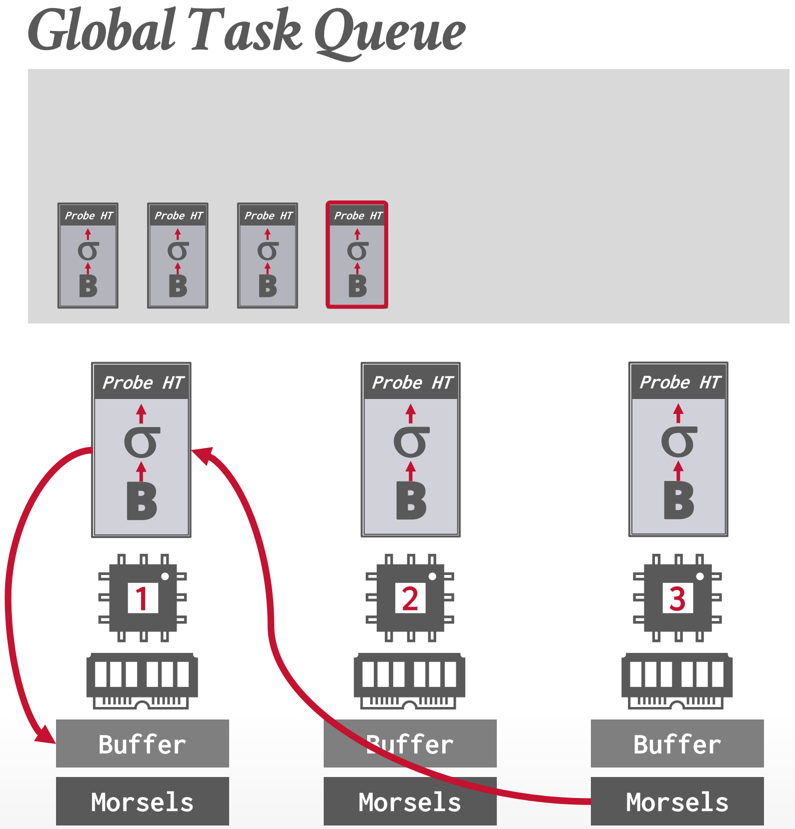
Global Task Queue



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
```





MORSEL-DRIVEN 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.

OBSERVATION

Tasks have different execution costs per tuple, but HyPer treats each task having the same cost.
→ Example: Simple Selection vs. String Matching

HyPer also has no notion of execution priorities.
→ The DBMS executes all query tasks with the same priority.
→ Short-running queries get blocked behind long-running queries.

UMBRA: MORSEL SCHEDULING 2.0

Rather than scheduling tasks based chunks on data (morsels), the DBMS schedules tasks based on time.

→ Tasks are not created statically at runtime.
→ System exponentially grow morsel sizes per task set until an individual task takes 1ms to execute.

Automatic priority decay for query tasks.

→ Ensures that short-running queries finish quickly and long-running queries are not starved for resources.
→ Modern implementation of stride scheduling.

SELF-TUNING QUERY SCHEDULING FOR ANALYTICAL WORKLOADS
SIGMOD 2021

UMBRA: MORSEL SCHEDULING 2.0

Rather than scheduling tasks based chunks on data (morsels), the DBMS schedules tasks based on time.

→ Tasks are not created statically at runtime.
→ System exponentially grow morsel sizes per task set until an individual task takes 1ms to execute.

Automatic priority decay for query tasks.

→ Ensures that short-running queries finish quickly and long-running queries are not starved for resources.
→ Modern implementation of stride scheduling.

SELF-TUNING QUERY SCHEDULING FOR ANALYTICAL WORKLOADS
SIGMOD 2021

UMBRA: SCHEDULING STATE

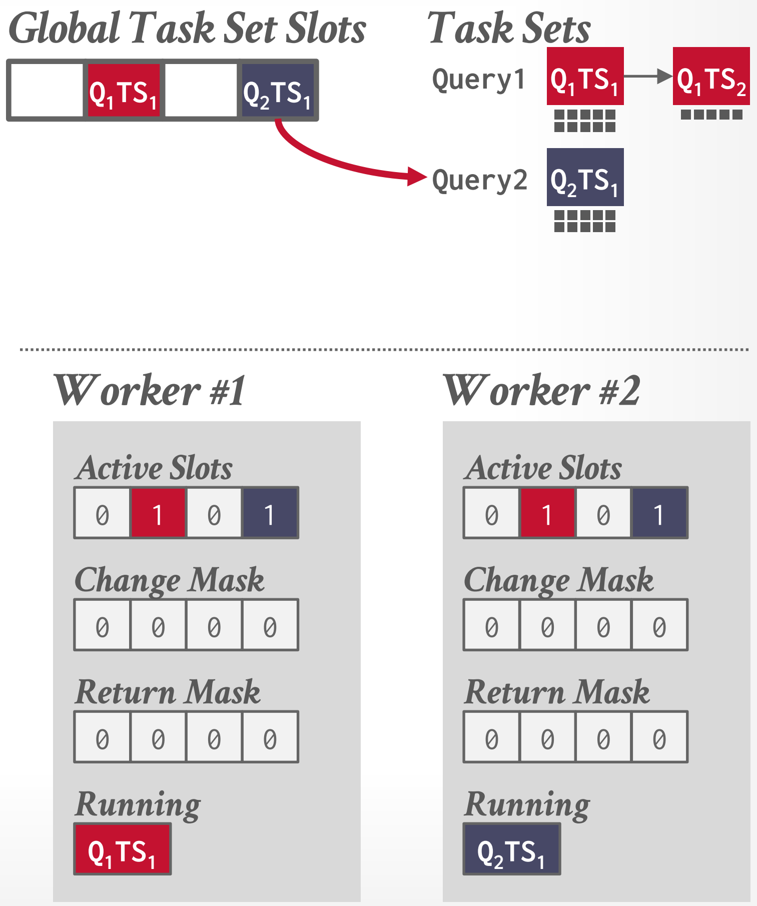
Each worker maintains its own thread-local meta-data about the available tasks to execute.

→ **Active Slots:** Which entries in the global slot array have active task sets available.

→ **Change Mask:** Indicates when a new task set is added to the global slot array.

→ **Return Mask:** Indicates when a worker completes a task set.

Workers perform CaS updates to TLS meta-data to broadcast changes.



UMBRA: SCHEDULING STATE

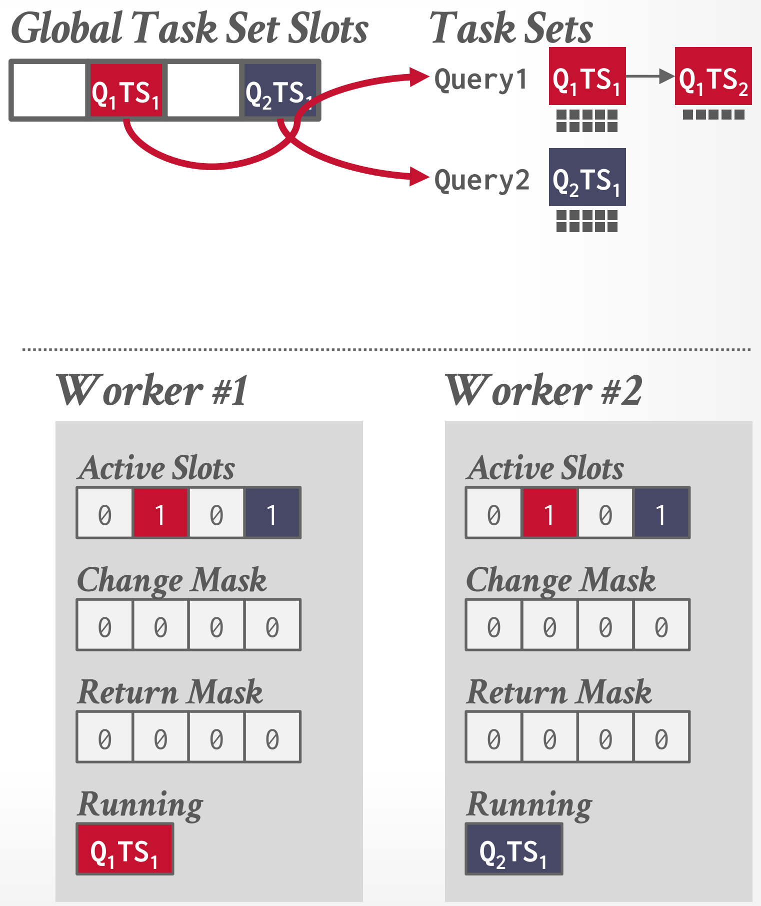
Each worker maintains its own thread-local meta-data about the available tasks to execute.

→ **Active Slots:** Which entries in the global slot array have active task sets available.

→ **Change Mask:** Indicates when a new task set is added to the global slot array.

→ **Return Mask:** Indicates when a worker completes a task set.

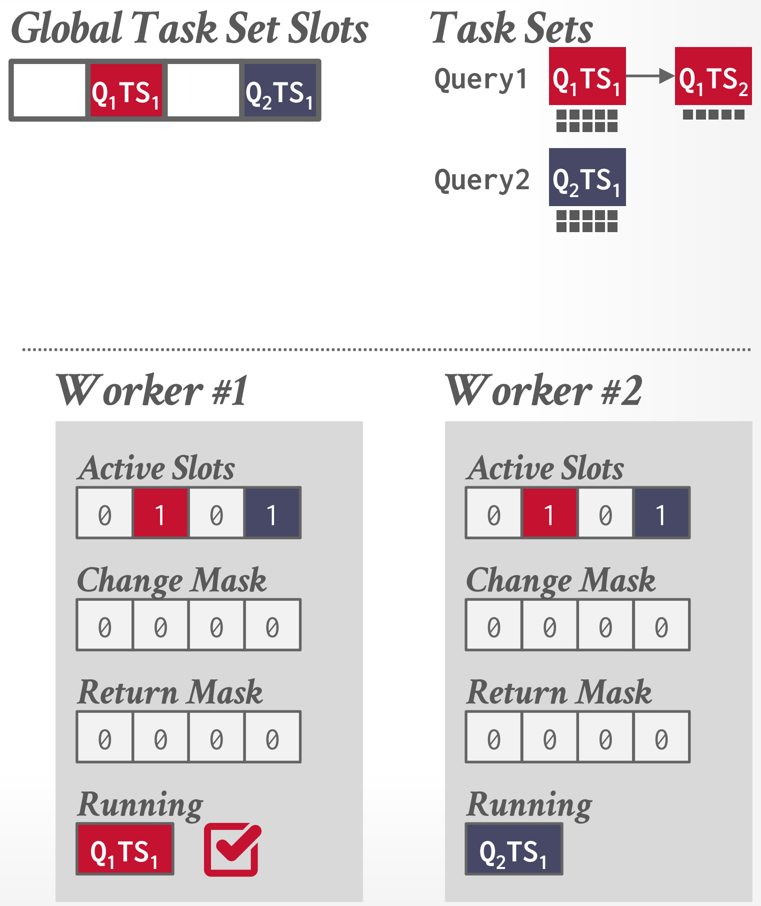
Workers perform CaS updates to TLS meta-data to broadcast changes.



UMBRA: SCHEDULING STATE

When a worker completes the last morsel for a query's active task set, it inserts the next task set into the global slot array and updates the **Return Mask** for all workers.

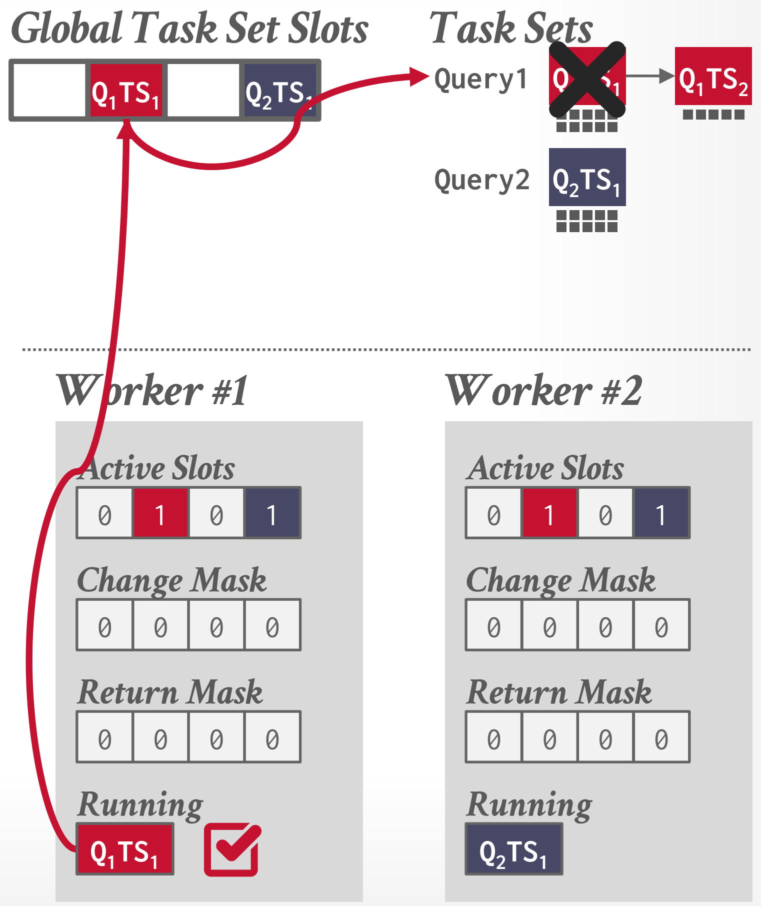
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



UMBRA: SCHEDULING STATE

When a worker completes the last morsel for a query's active task set, it inserts the next task set into the global slot array and updates the **Return Mask** for all workers.

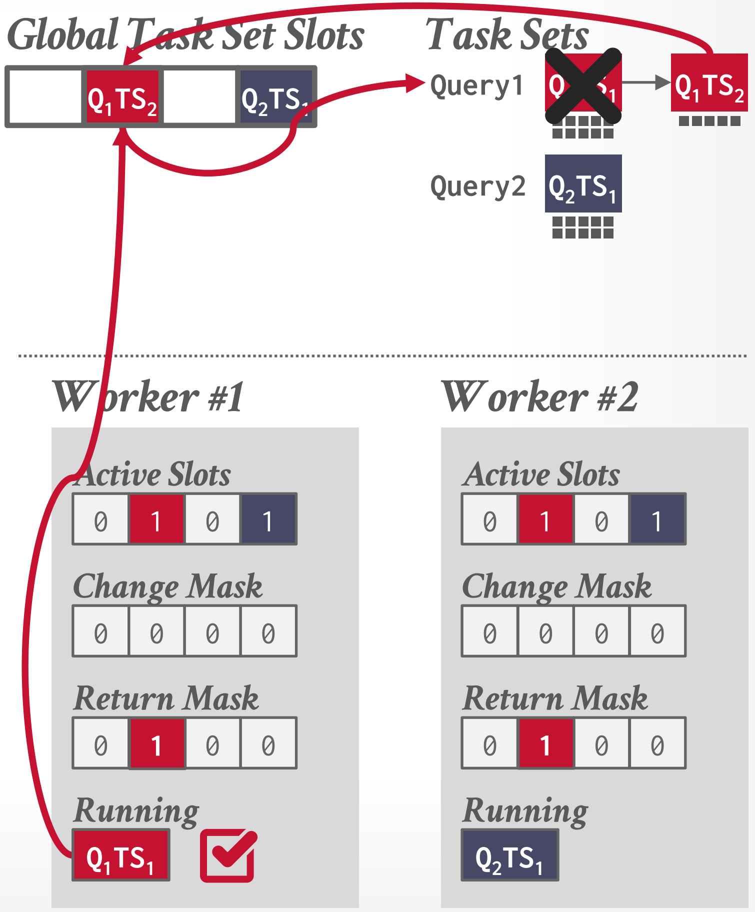
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



UMBRA: SCHEDULING STATE

When a worker completes the last morsel for a query's active task set, it inserts the next task set into the global slot array and updates the **Return Mask** for all workers.

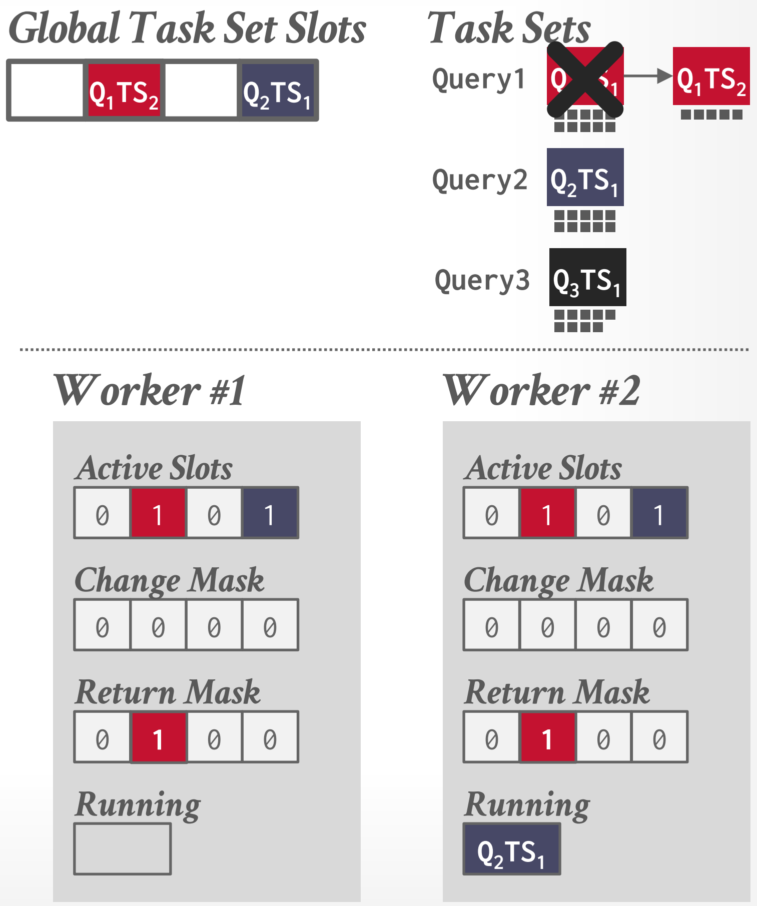
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



UMBRA: SCHEDULING STATE

When a worker completes the last morsel for a query's active task set, it inserts the next task set into the global slot array and updates the **Return Mask** for all workers.

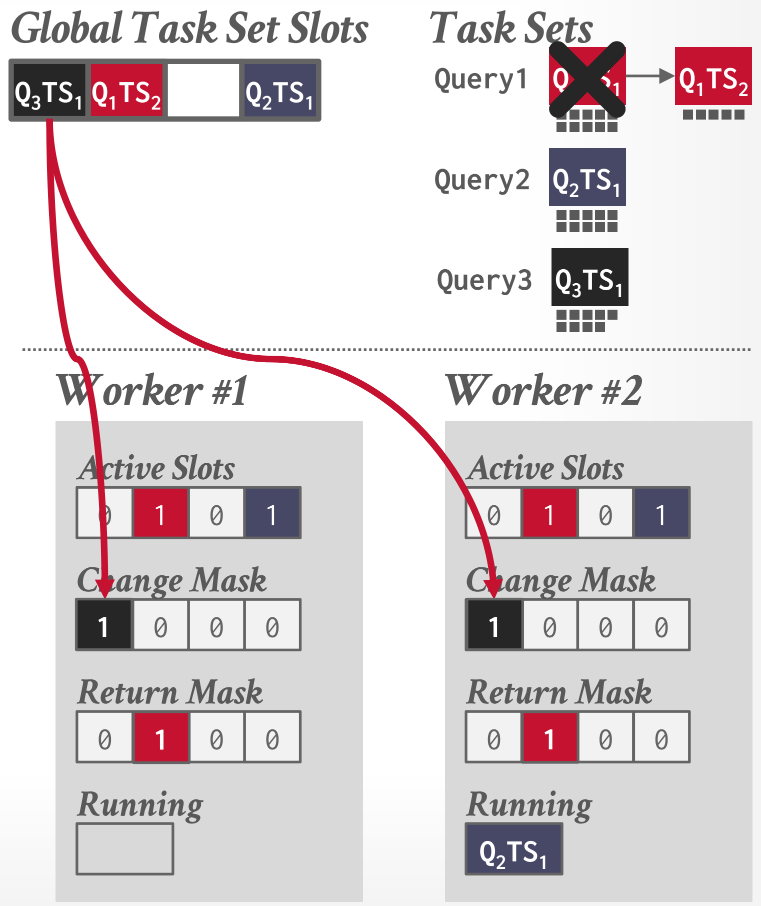
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



UMBRA: SCHEDULING STATE

When a worker completes the last morsel for a query's active task set, it inserts the next task set into the global slot array and updates the **Return Mask** for all workers.

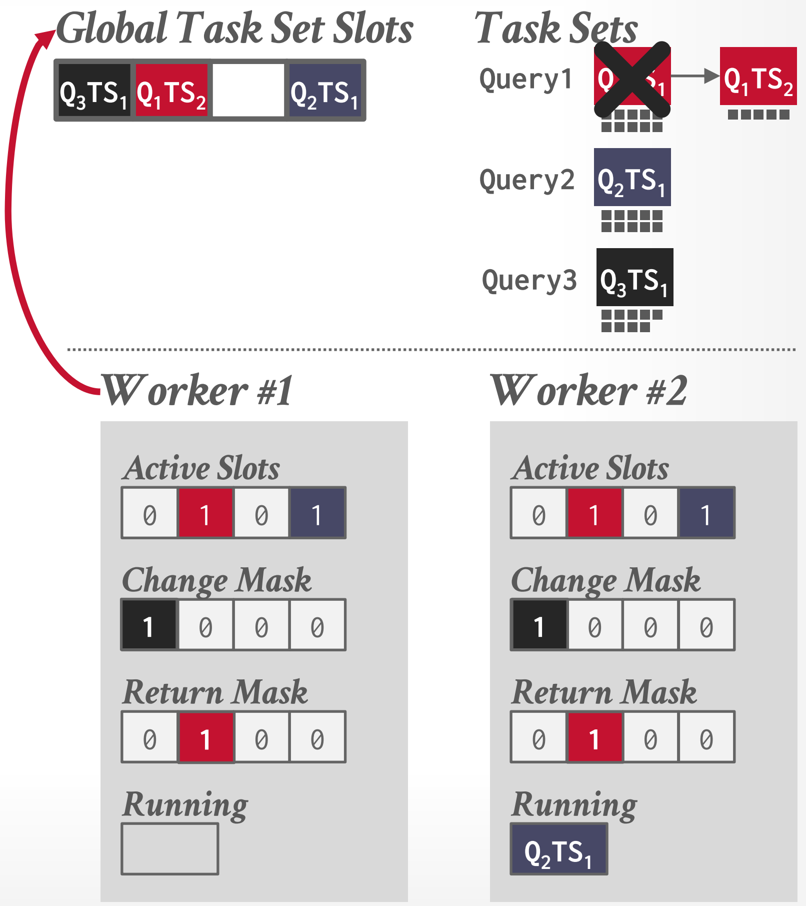
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



UMBRA: SCHEDULING STATE

When a worker completes the last morsel for a query's active task set, it inserts the next task set into the global slot array and updates the **Return Mask** for all workers.

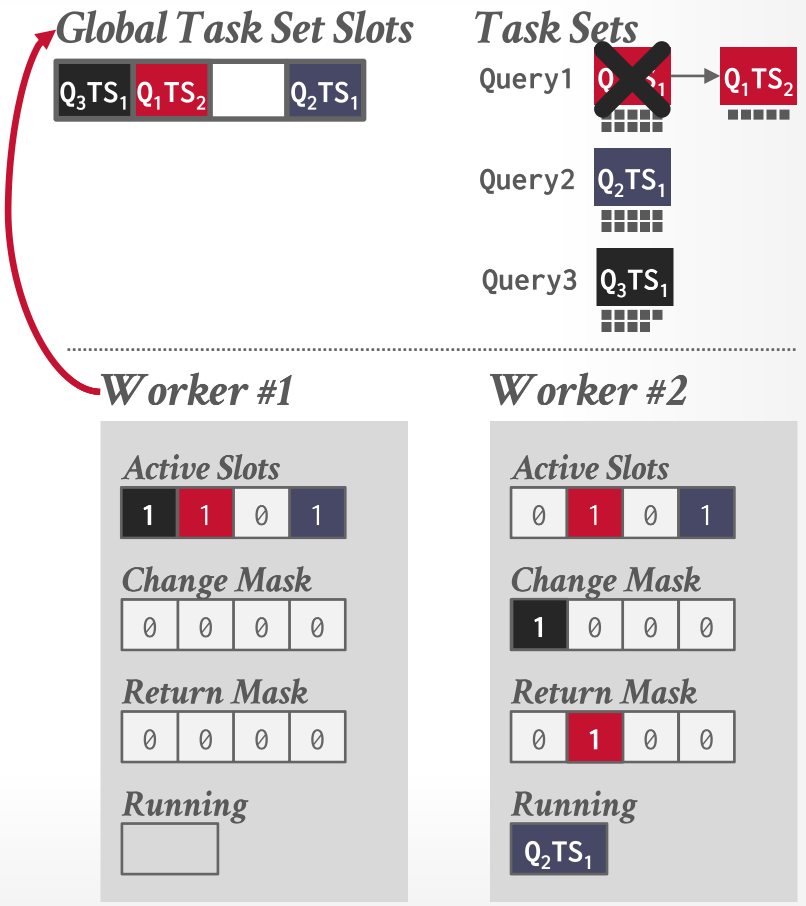
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



UMBRA: SCHEDULING STATE

When a worker completes the last morsel for a query's active task set, it inserts the next task set into the global slot array and updates the **Return Mask** for all workers.

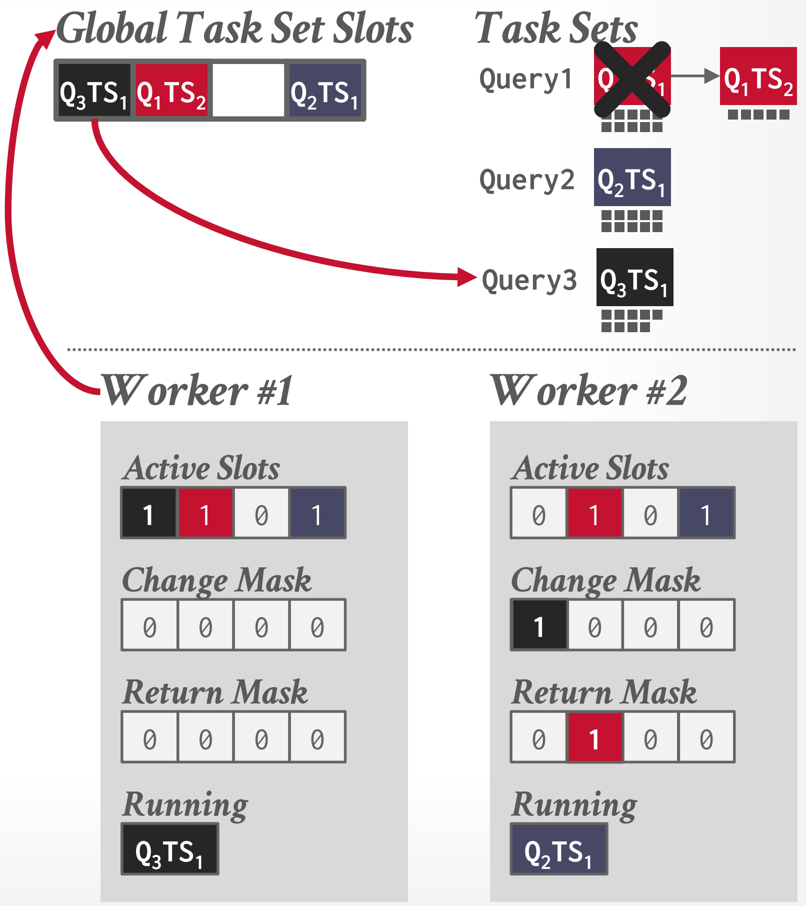
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



UMBRA: SCHEDULING STATE

When a worker completes the last morsel for a query's active task set, it inserts the next task set into the global slot array and updates the **Return Mask** for all workers.

When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



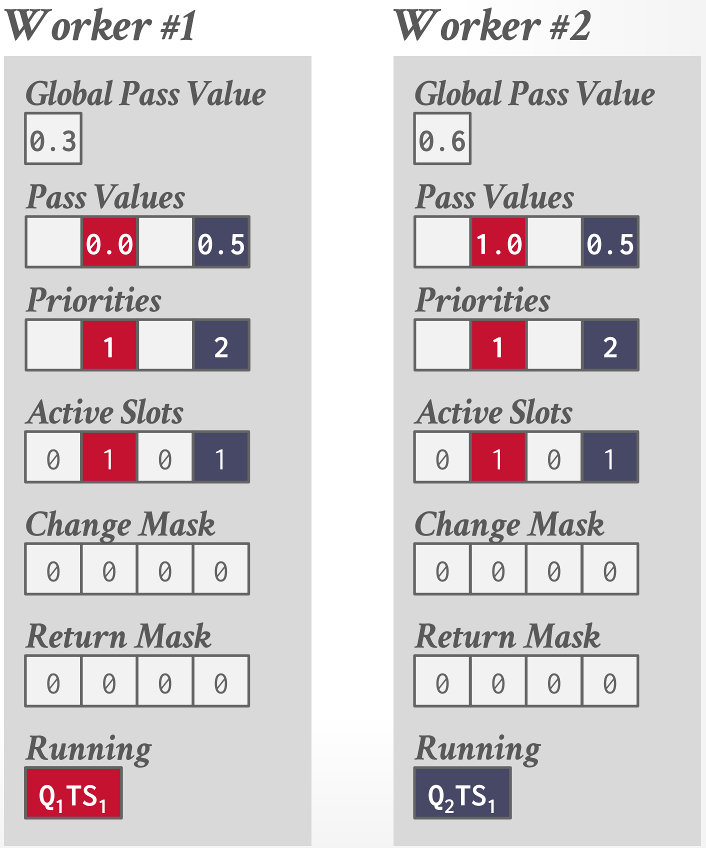
UMBRA: PRIORITY DECAY

Each worker maintains additional thread-local meta-data to compute the priorities of queries in real-time:

→ **Global Pass:** How many quantum rounds the DBMS has completed.

→ **Pass Values:** How much time a query has consumed.

→ **Priorities:** Decremented as the query runs longer.



Source: Benjamin Wagner

SAP 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

SAP HANA: THREAD GROUPS

DBMS maintains **soft** and **hard** priority task queues for each thread group.

→ Threads can 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**: Waiting for a task (like a free thread) but blocked in the kernel until the watchdog thread wakes it up.

SAP HANA: NUMA-AWARE SCHEDULER

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

→ Allow more cross-region stealing if DBMS is CPU-bound.

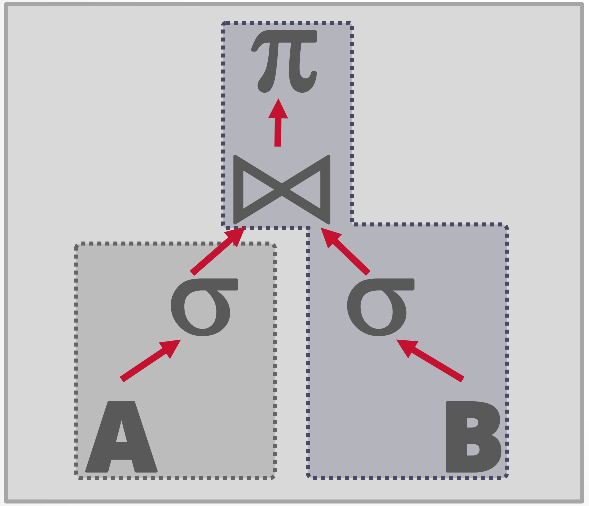
SAP found that work stealing was not as beneficial for systems with a larger number of sockets.

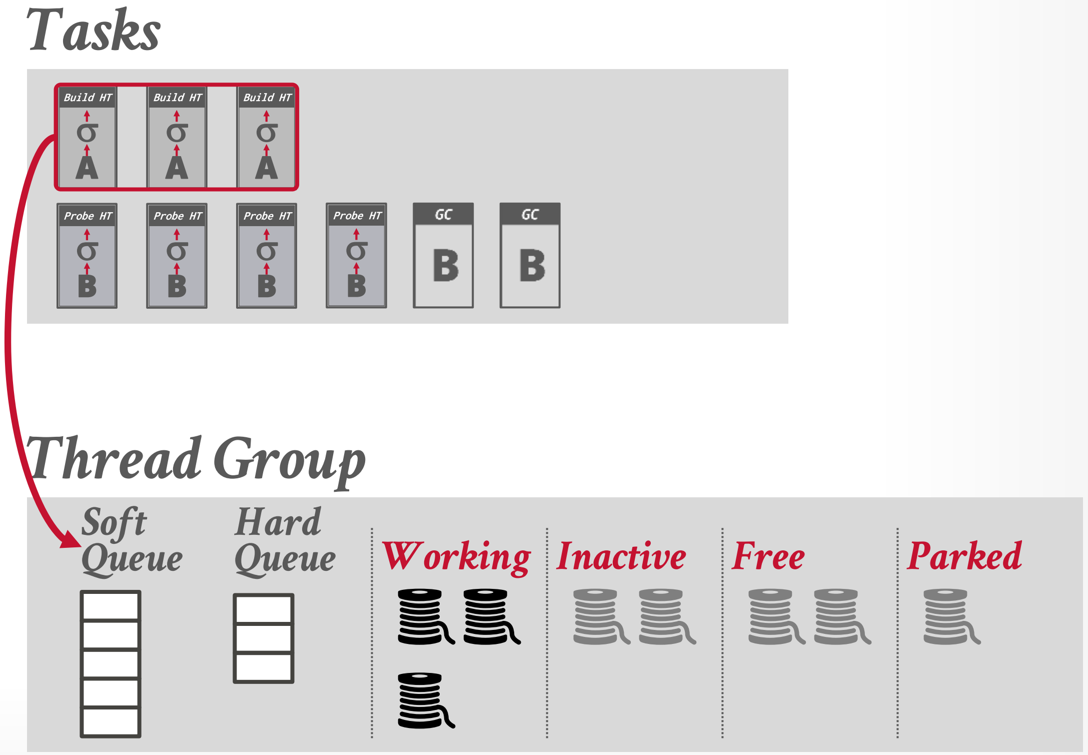
→ HyPer (2-4 sockets) vs. HANA (64 sockets)

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

SAP HANA: NUMA-AWARE SCHEDULER

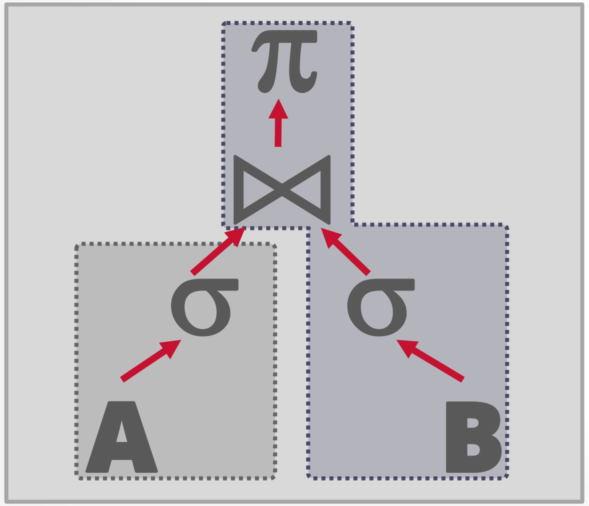
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

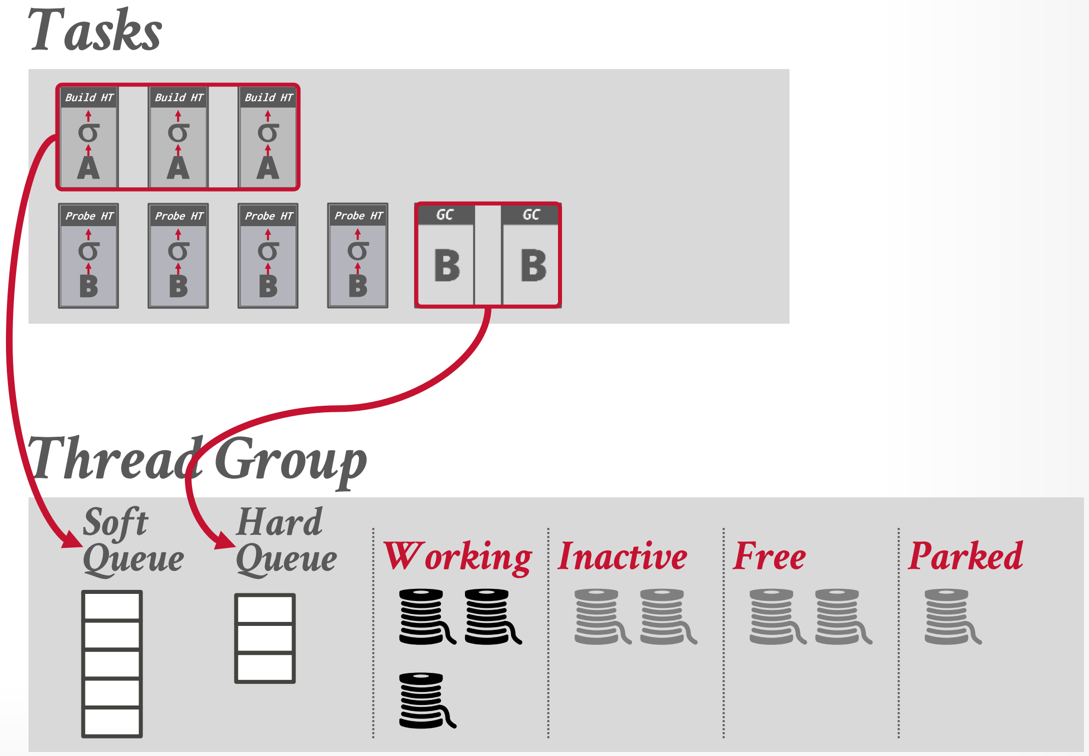




SAP HANA: NUMA-AWARE SCHEDULER

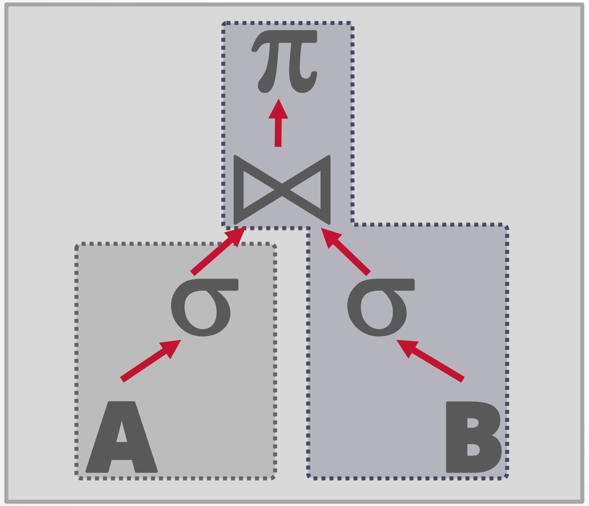
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

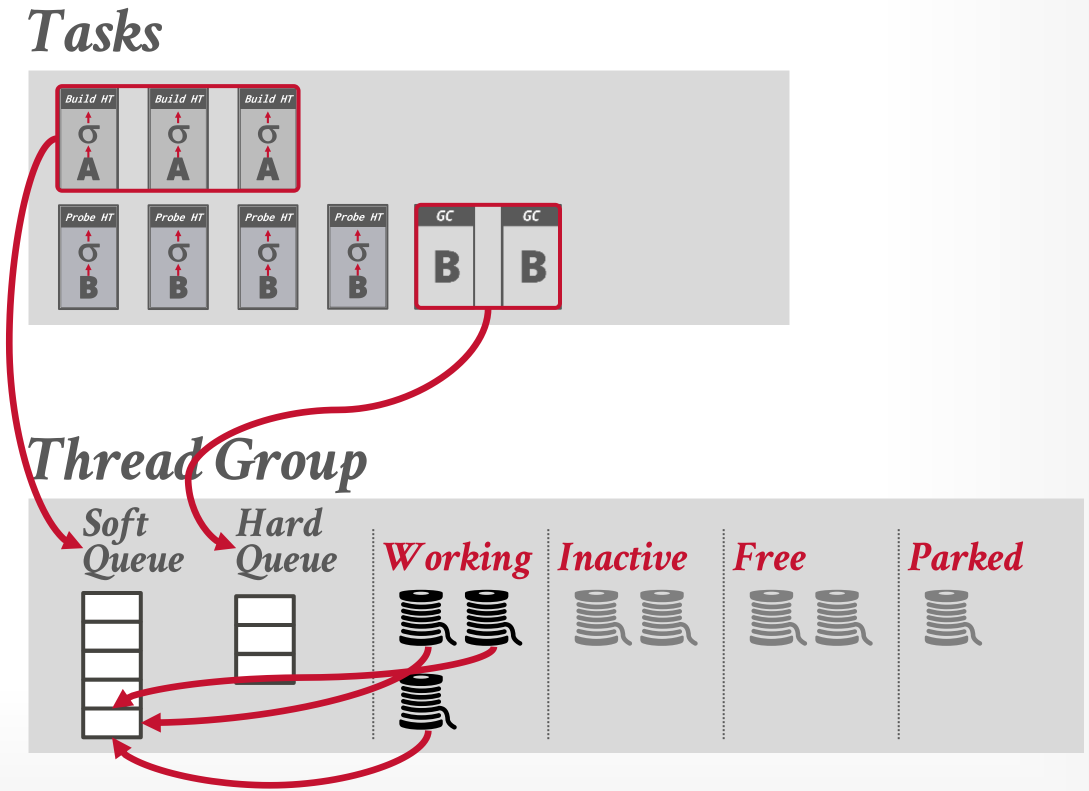




SAP HANA: NUMA-AWARE SCHEDULER

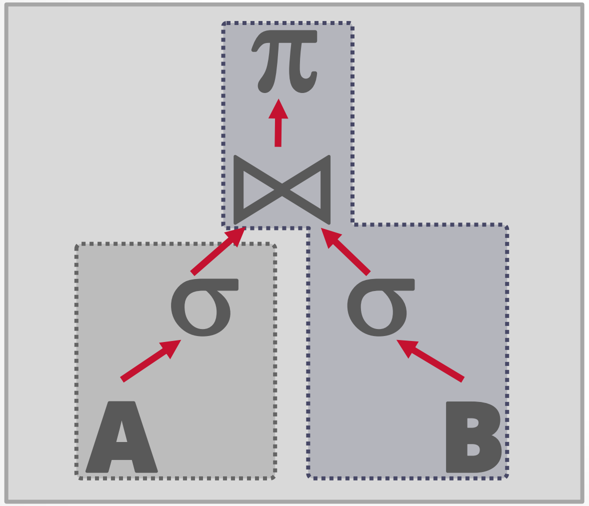
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

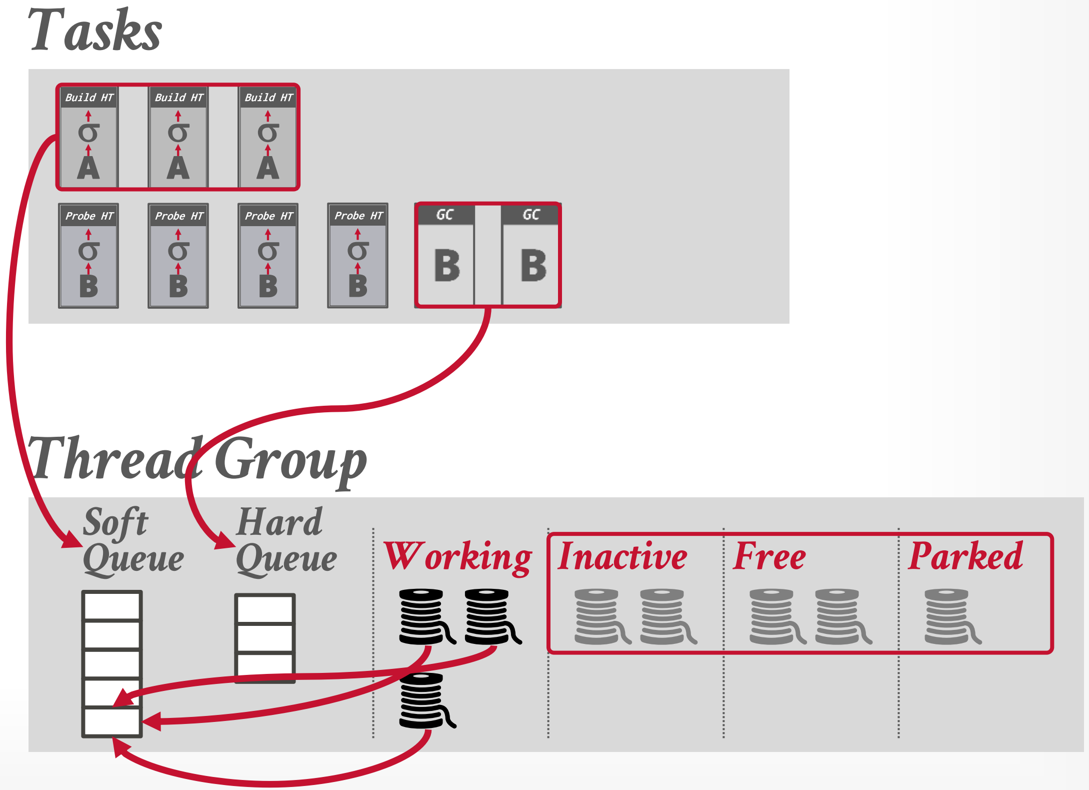




SAP HANA: NUMA-AWARE SCHEDULER

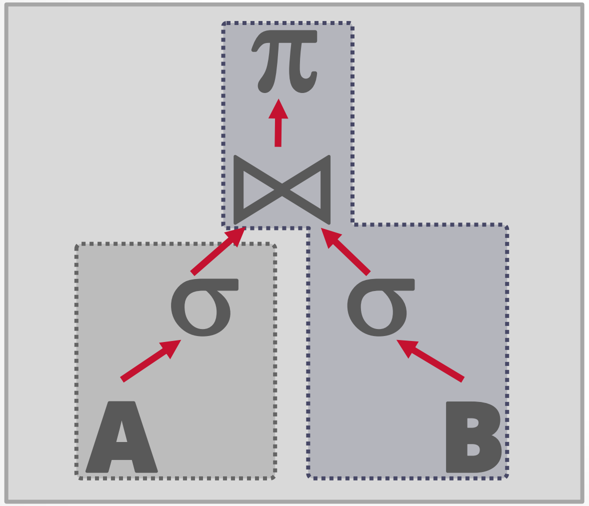
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

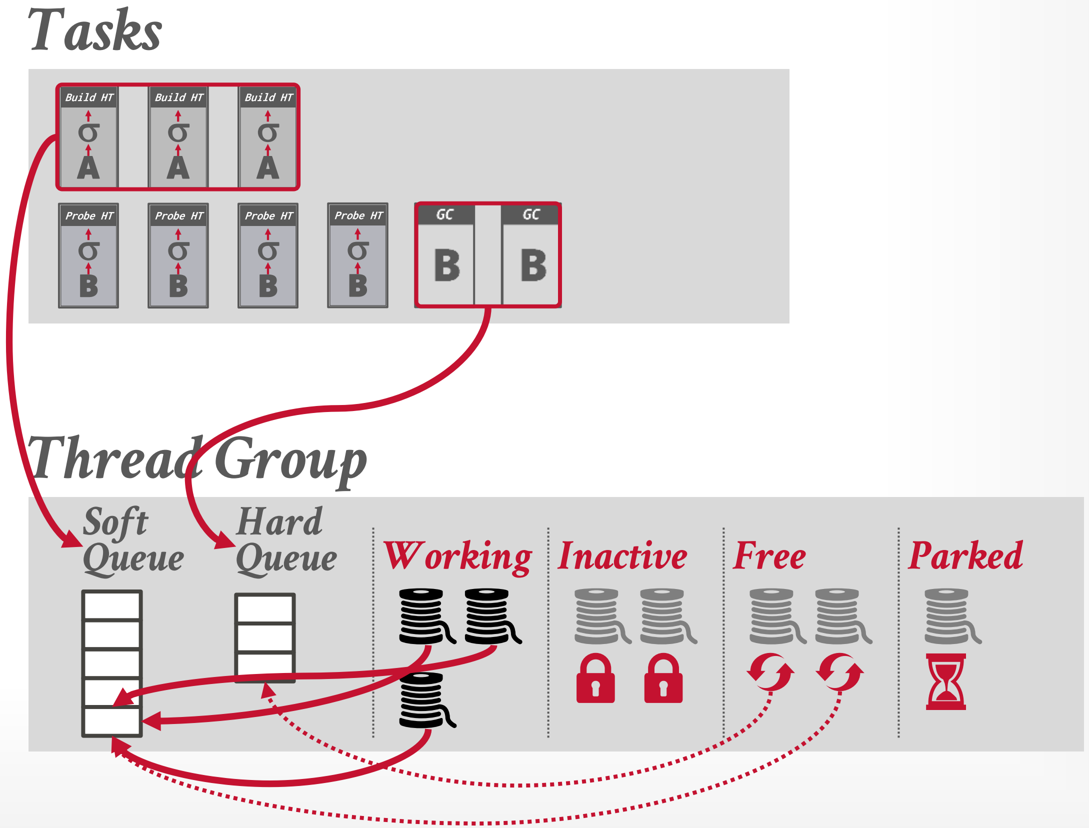




SAP HANA: NUMA-AWARE SCHEDULER

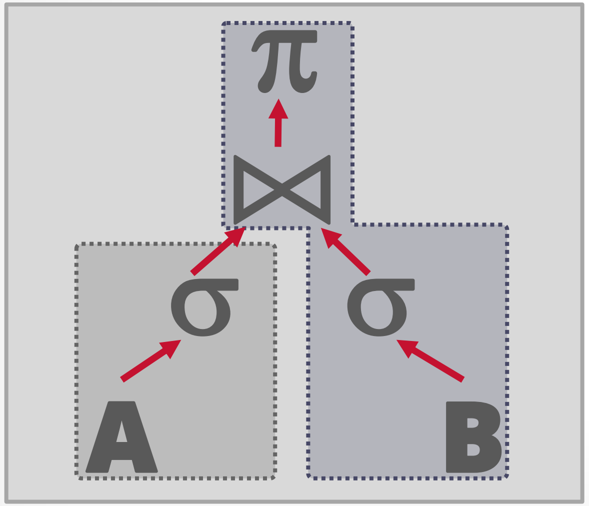
```
SELECT A.id, B.value
  FROM A JOIN B
    ON A.id = B.id
 WHERE A.value < 99
   AND B.value > 100
```

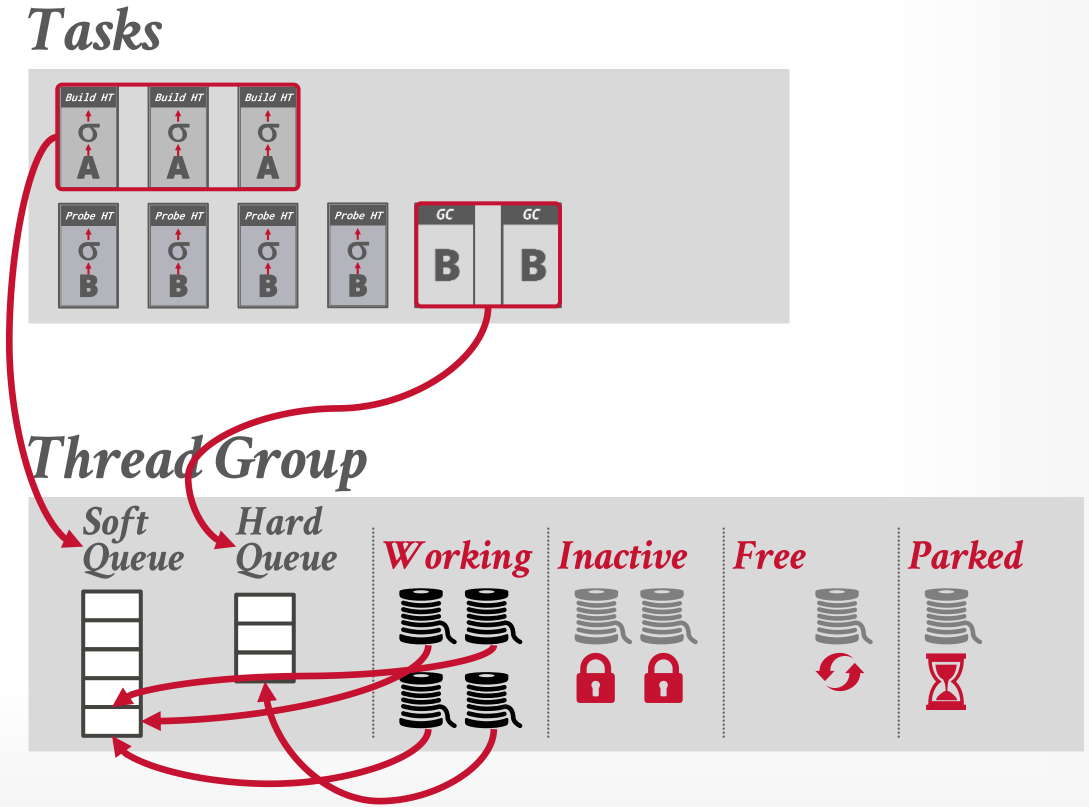




SAP HANA: NUMA-AWARE SCHEDULER

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





SQL SERVER: SQLOS

SQLOS is a user-mode NUMA-aware OS layer that runs inside of the DBMS and manages provisioned hardware resources.

→ Determines which tasks are scheduled onto which threads.
→ Also manages I/O scheduling and higher-level concepts like logical database locks.

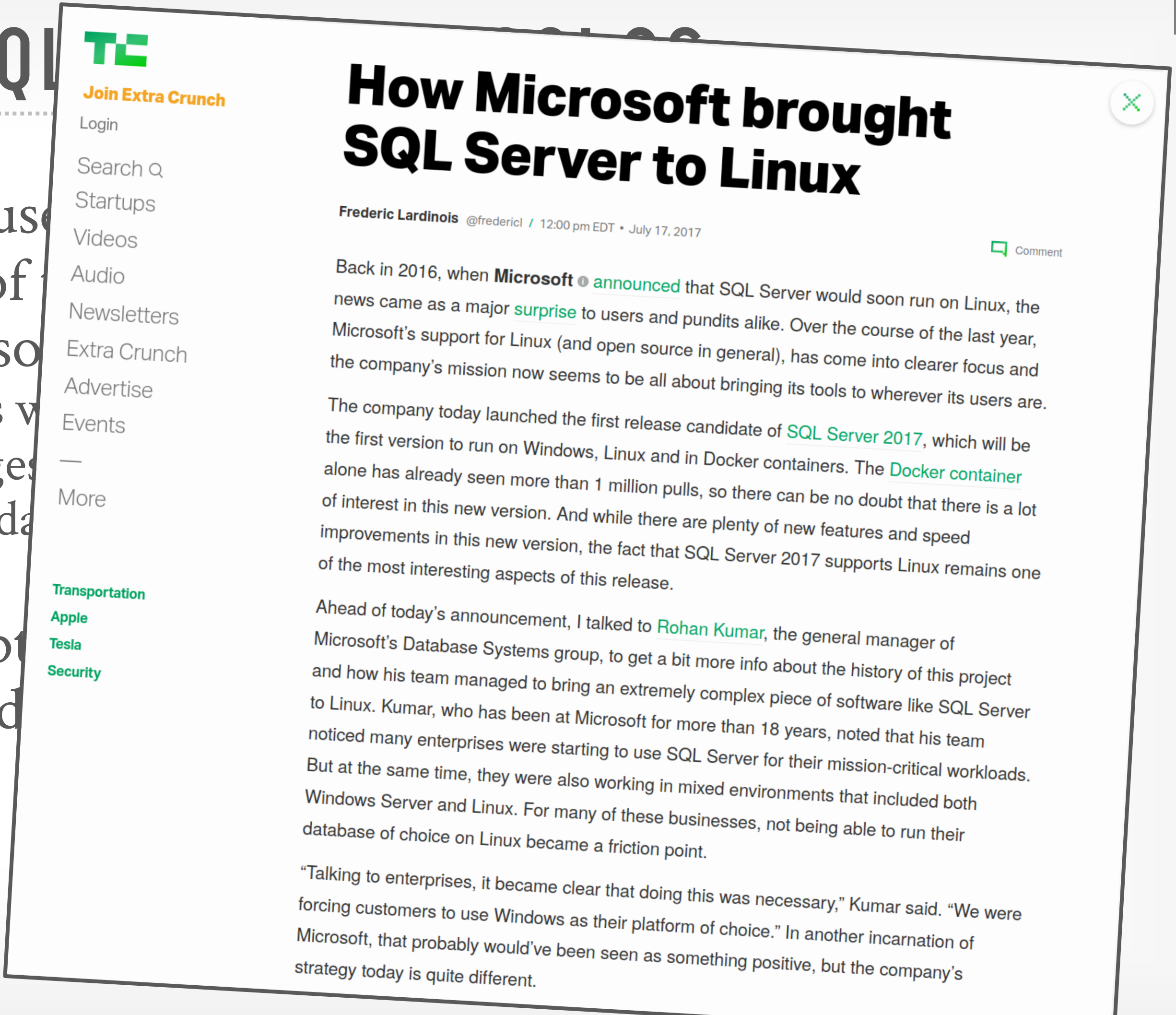
Non-preemptive thread scheduling through instrumented DBMS code.

MICROSOFT SQL SERVER 2012 INTERNALS
PEARSON 2013

SQL

SQLOS is a use
runs inside of
hardware reso
→ Determines w
→ Also manages
like logical da

Non-preempt
instrumented



How Microsoft brought SQL Server to Linux

Frederic Lardinois @fredericl / 12:00 pm EDT • July 17, 2017

Back in 2016, when Microsoft announced that SQL Server would soon run on Linux, the news came as a major surprise to users and pundits alike. Over the course of the last year, Microsoft's support for Linux (and open source in general), has come into clearer focus and the company's mission now seems to be all about bringing its tools to wherever its users are.

The company today launched the first release candidate of SQL Server 2017, which will be the first version to run on Windows, Linux and in Docker containers. The Docker container alone has already seen more than 1 million pulls, so there can be no doubt that there is a lot of interest in this new version. And while there are plenty of new features and speed improvements in this new version, the fact that SQL Server 2017 supports Linux remains one of the most interesting aspects of this release.

Ahead of today's announcement, I talked to Rohan Kumar, the general manager of Microsoft's Database Systems group, to get a bit more info about the history of this project and how his team managed to bring an extremely complex piece of software like SQL Server to Linux. Kumar, who has been at Microsoft for more than 18 years, noted that his team noticed many enterprises were starting to use SQL Server for their mission-critical workloads. But at the same time, they were also working in mixed environments that included both Windows Server and Linux. For many of these businesses, not being able to run their database of choice on Linux became a friction point.

"Talking to enterprises, it became clear that doing this was necessary," Kumar said. "We were forcing customers to use Windows as their platform of choice." In another incarnation of Microsoft, that probably would've been seen as something positive, but the company's strategy today is quite different.

MICROSOFT SQL SERVER 2012 INTERNALS
PEARSON 2013

SQL SERVER: SQLOS

SQLOS quantum is 4 ms but the scheduler cannot enforce that.

DBMS developers must add explicit yield calls in various locations in the source code.

Other Examples:
→ ScyllaDB (via Seastar)
→ FaunaDB
→ CoroBase

```
SELECT * FROM R WHERE R.val = ?
```

Approximate Plan

```
for t in R:
  if eval(predicate, tuple, params):
    emit(tuple)
```

SQL SERVER: SQLOS

SQLOS quantum is 4 ms but the scheduler cannot enforce that.

DBMS developers must add explicit yield calls in various locations in the source code.

Other Examples:
→ ScyllaDB (via Seastar)
→ FaunaDB
→ CoroBase

```
SELECT * FROM R WHERE R.val = ?
```

```
last = now()
for tuple in R:
  if now() - last > 4ms:
     yield
     last = now()
  if eval(predicate, tuple, params):
    emit(tuple)
```

SQL SERVER: SQLOS

SQLOS quantum is 4 ms but the scheduler cannot enforce that.

DBMS developers must add explicit yield calls in various locations in the source code.

Other Examples:
→ ScyllaDB (via Seastar)
→ FaunaDB
→ CoroBase

```
SELECT * FROM R WHERE R.val = ?
```

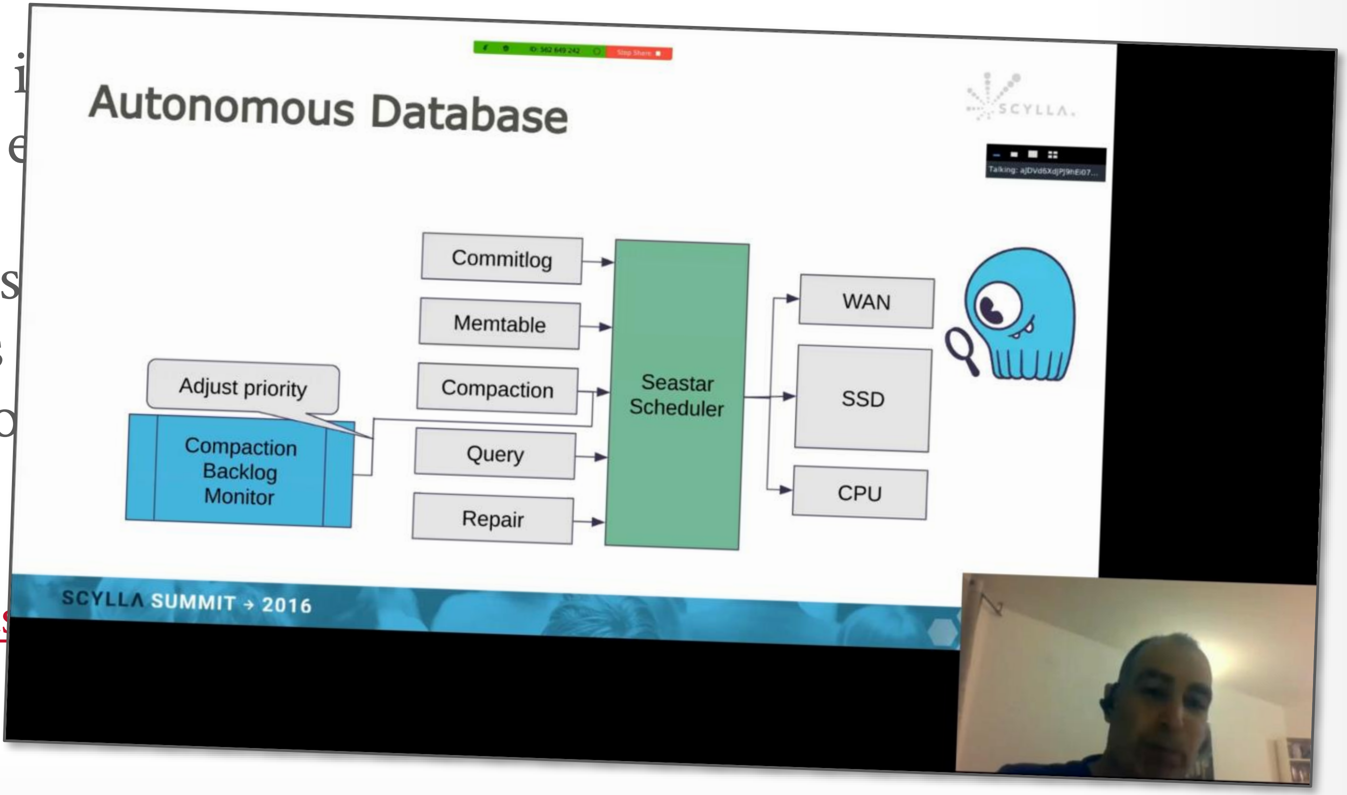
```
last = now()
for tuple in R:
  if now() - last > 4ms:
  ➡ yield
     last = now()
  if eval(predicate, tuple, params):
    emit(tuple)
```

SQL SERVER: SQLOS

SQLOS quantum i
scheduler cannot e

DBMS developers
explicit yield calls
locations in the so

Other Examples:
→ ScyllaDB (via Seas
→ FaunaDB
→ CoroBase


Autonomous Database

OBSERVATION

All the methods discussed today assume they are running on a single-node DBMSs with low-latency access to worker state.

→ Worker state resides on compute resources.

They also assume that the only resource consumed by queries is compute.

→ Did not consider memory, network, temp disk space.

→ Some resources are fungible.

DISTRIBUTED QUERY SCHEDULING

Separate service that is responsible for scheduling query tasks across nodes.

→ All the major cloud-based DBMSs will favor completing short running queries over longer running ones.

Scheduler could be **centralized** or **hierarchical**.

→ Scheduling granularity could either be a bundle of tasks for a node or a single task per worker.

→ Still need to track dependencies between query pipelines / tasks as we did on a single node.

DYNAMIC SCALING VS. WORK STEALING

There are two design decisions on how to handle queries that take longer to complete than expected.

Approach #1: Dynamic Scaling
→ Provision additional workers before a query starts.
→ Example: Snowflake Flexible Compute

Approach #2: Work Stealing
→ A worker takes tasks from a peer.
→ Example: Snowflake Scheduler

PARTING THOUGHTS

A DBMS is a beautiful, strong-willed independent software. But it must use 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

Do <u>not</u> let the OS ruin your life.

We ignored maintenance tasks, but they are just like queries with lower priorities.

NEXT CLASS

Hash Joins!

Hash Joins!

Hash Joins!