Query Scheduling & Coordination

Iztok Savnik, FAMNIT

November, 2025.

SOURCES

- Course:
  - Carnegie Mellon University (CMU)
  - Advanced Database Systems (Spring 2024)
  - Prof. Andy Pavlo
  - Transparencies
- Papers:
  - Conferences VLDB, SIGMOD, CIDR, etc.
  - Journals ACM TODS, IS, VLDBJ, etc.

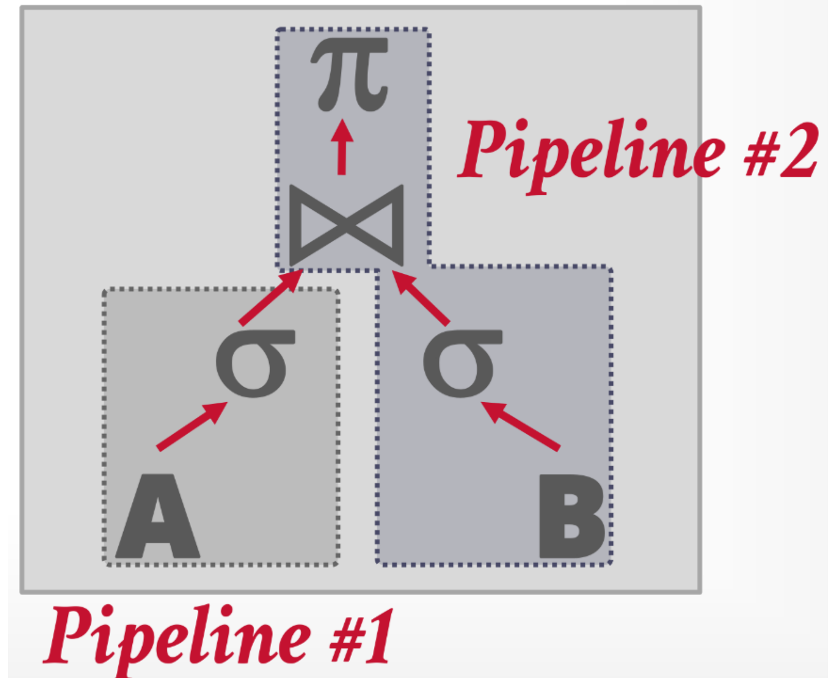
QUERY EXECUTION

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.



QUERY EXECUTION

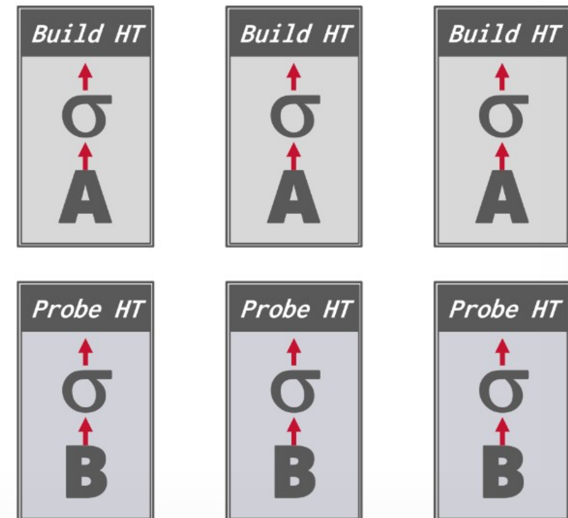
A query plan is a DAG of **operators**.

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.

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

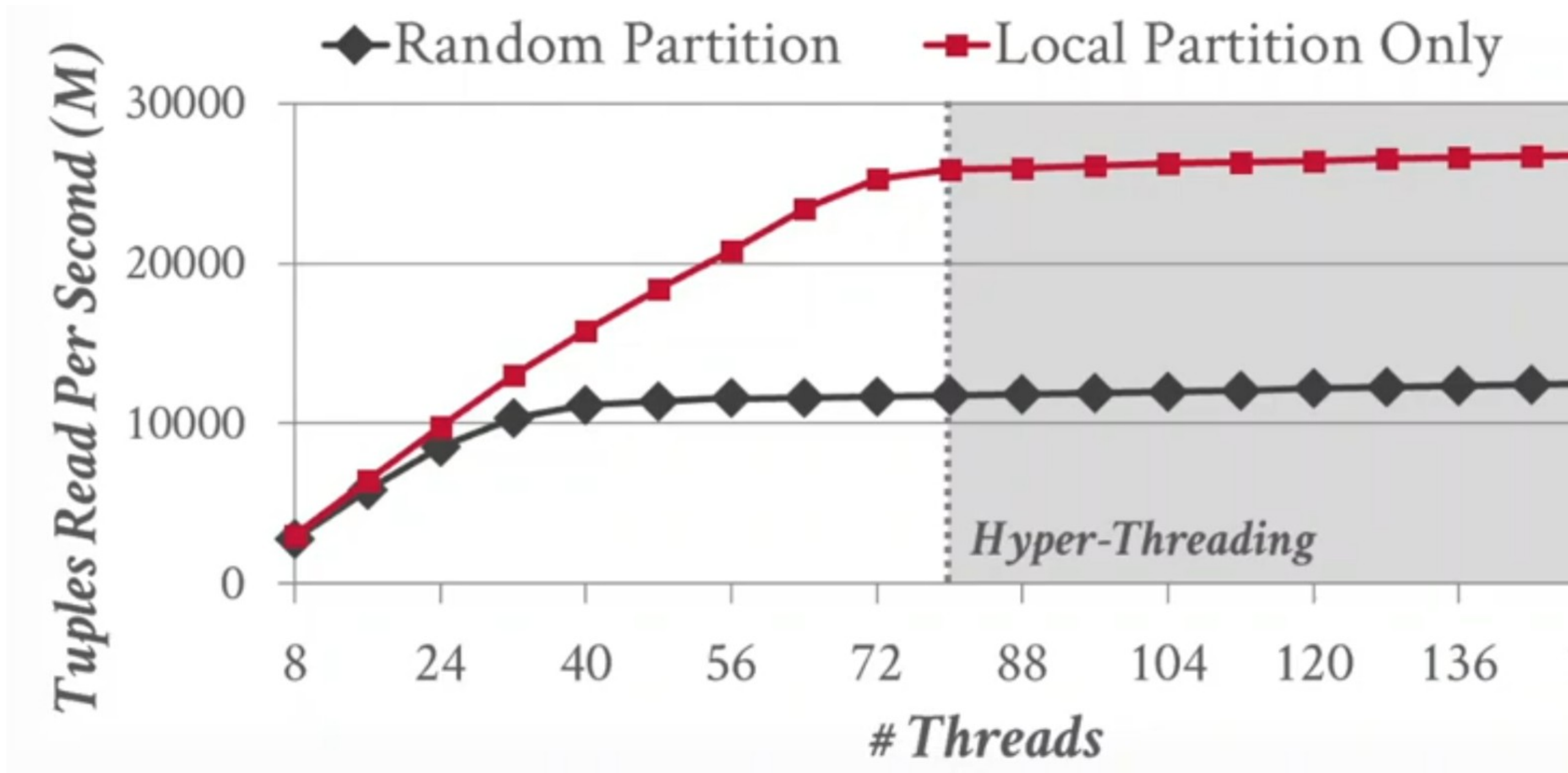
- Use a pool of workers per core (or per socket).
- Allows CPU cores to be fully utilized in case one worker at a core blocks.

DBMS primarily **processes data**!

- A large portion of the work involves memory access.
- … and we are using NUMA.

DATA PLACEMENT - OLAP

Sequential scan on 10M tuples, Processor: 8 sockets, 10 cores per node (2X HT)



OBSERVATION – LOCK-FREE DS

Why contention still happens?

CAS retry loops:

- Multiple threads pooling the same memory location using CAS. Only one succeeds — causing “CAS storms.”

Cache-line ping-pong:

- Tuching (CAS) the same memory region. Cache invalidations across cores. Cache line bounces among cores — contention on the memory bus (inter-socket coherence traffic)

NUMA remote memory:

- Access time is 2-3X slower then the NUMA local memory access.

Memory bandwidth saturation (inter-socket):

- Saturation of the memory interconnect between sockets, which limits the rate at which data can be transferred across NUMA nodes.

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.
- Distributed algorithms!

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 Range
- 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 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.

Slower tasks (**stragglers**) 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.

HYPER: ARCHITECTURE

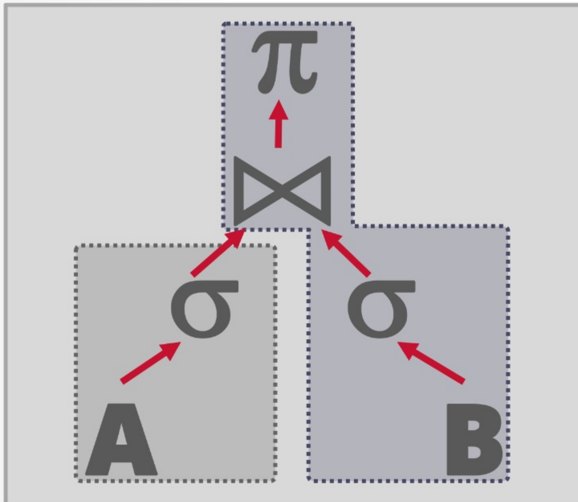
No separate dispatcher thread.

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

```sql
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

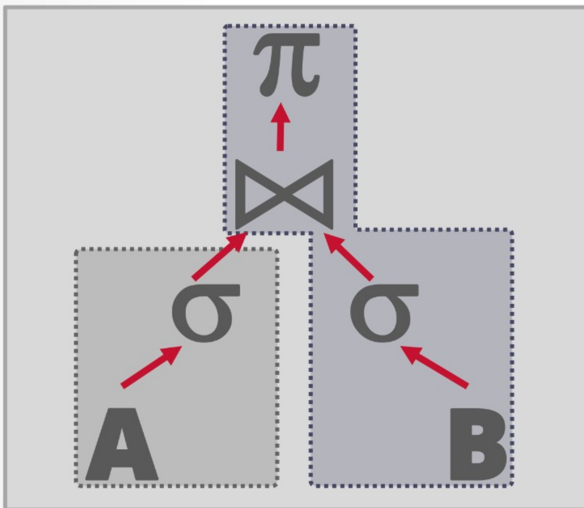
id	a1	a2	a3

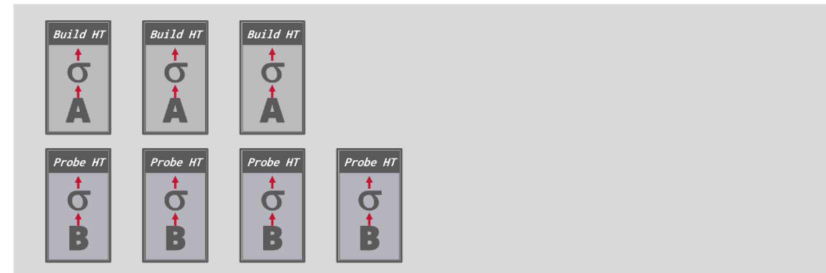
A_1 → 1

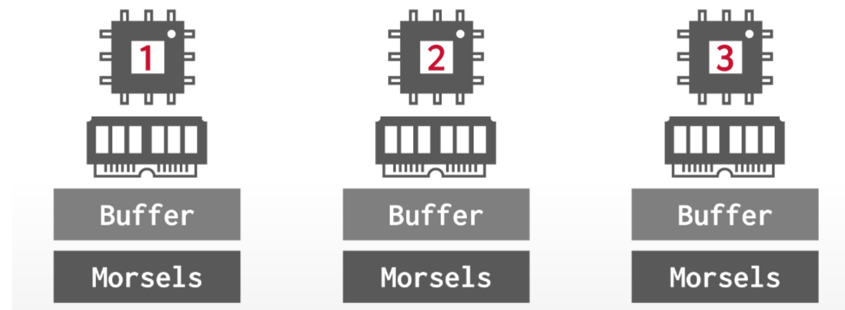
A_2 → 2

A_3 → 3

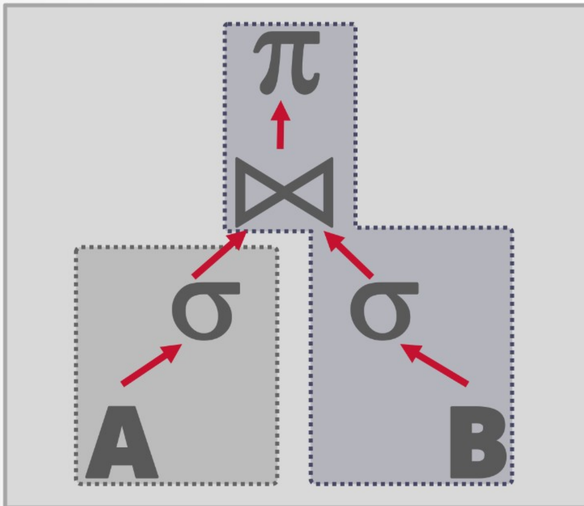
HYPER: EXECUTION EXAMPLE



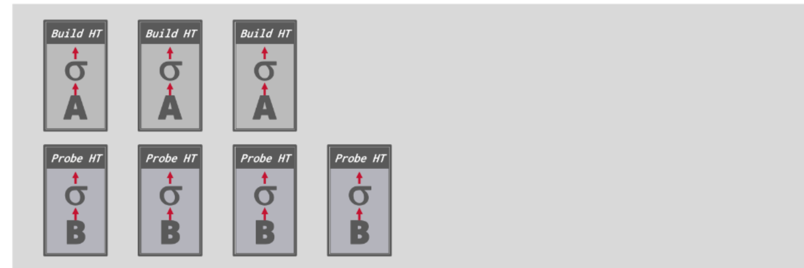


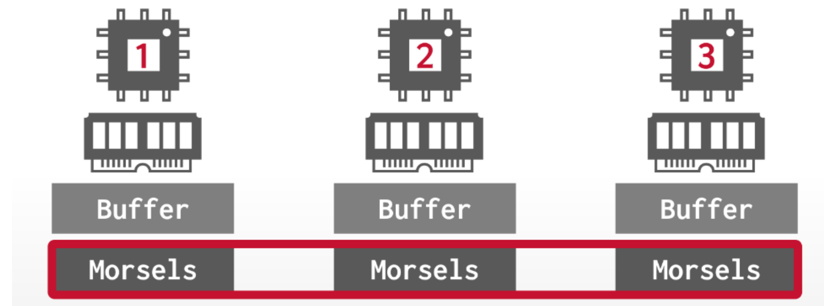


HYPER: EXECUTION EXAMPLE

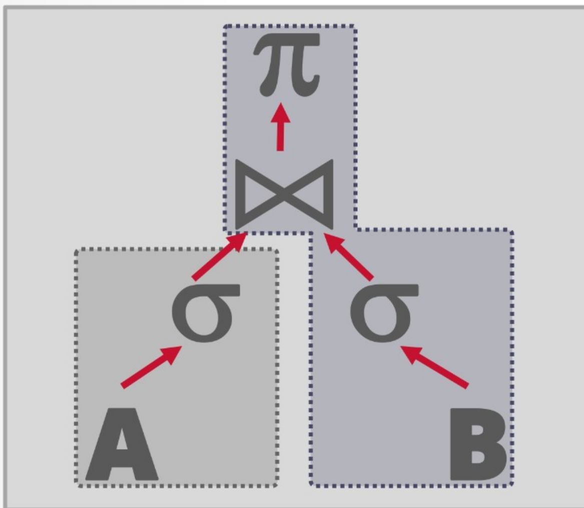


Global Task Queue

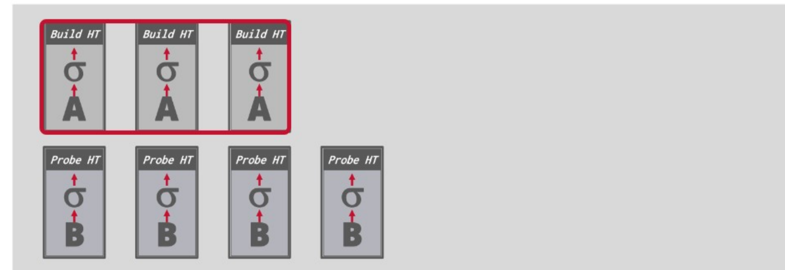


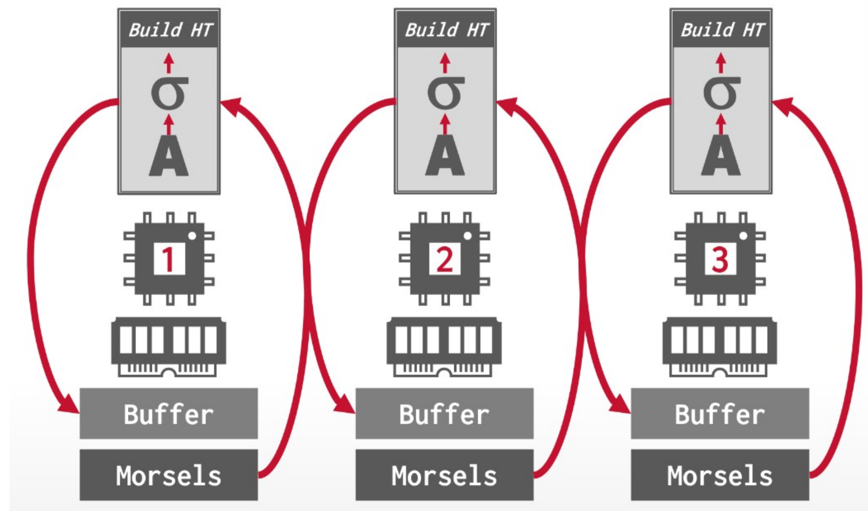


HYPER: EXECUTION EXAMPLE

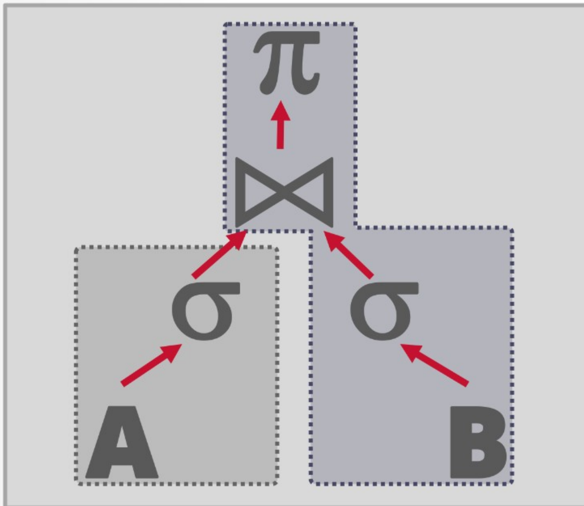


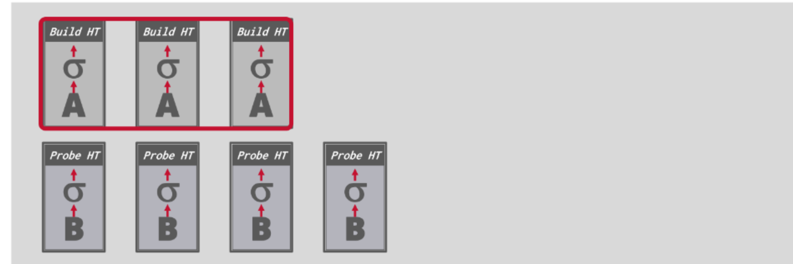
Global Task Queue

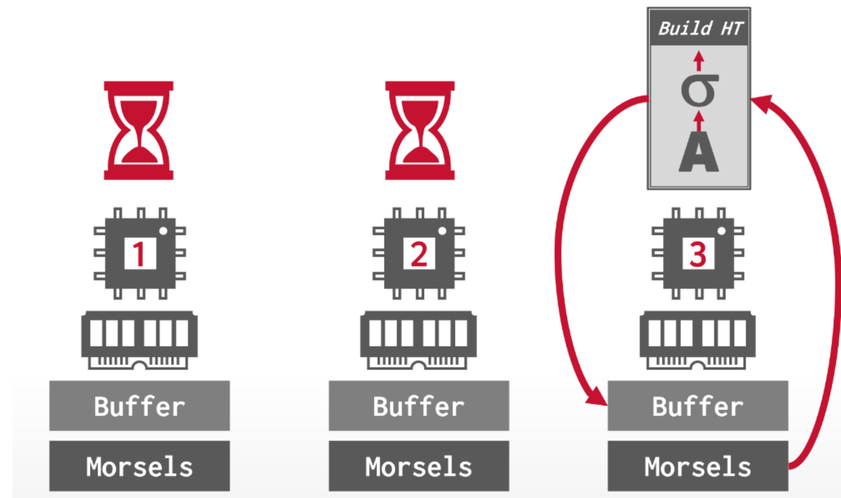




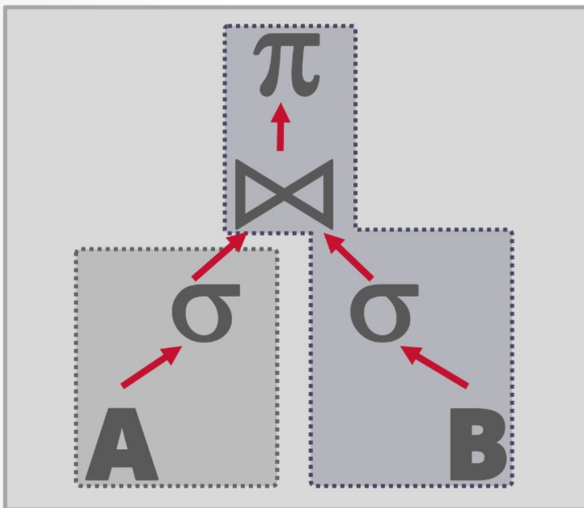
HYPER: EXECUTION EXAMPLE

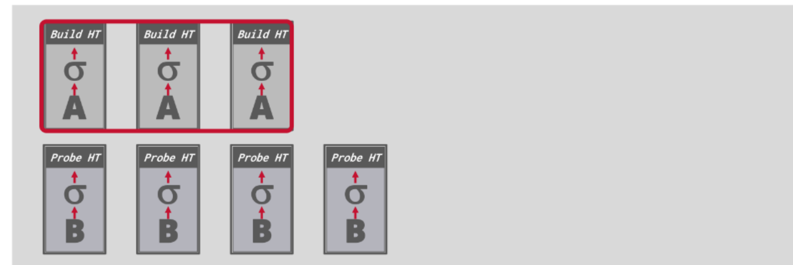


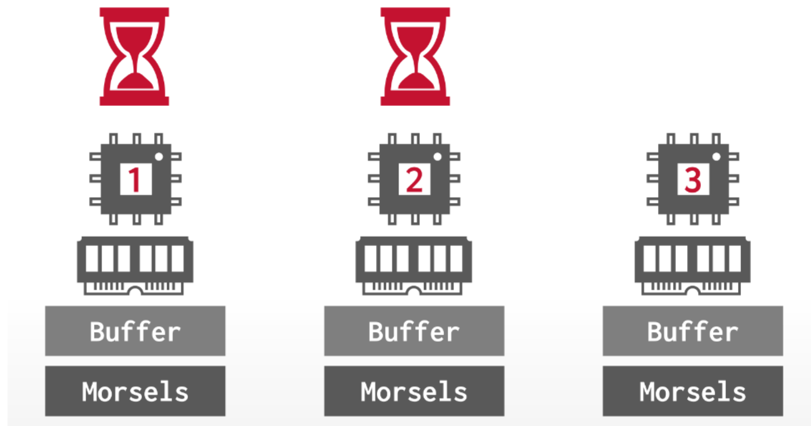




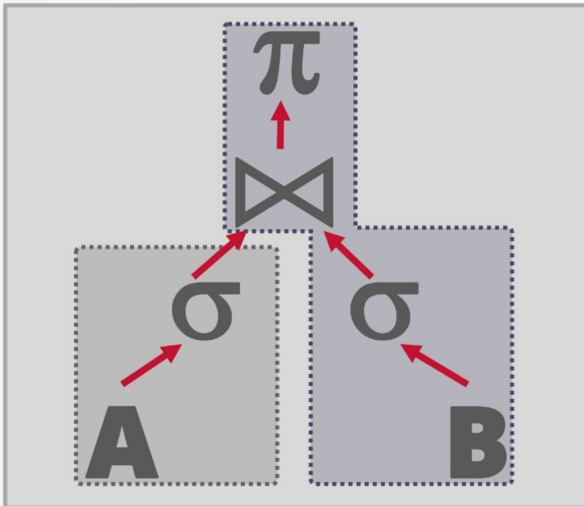
HYPER: EXECUTION EXAMPLE

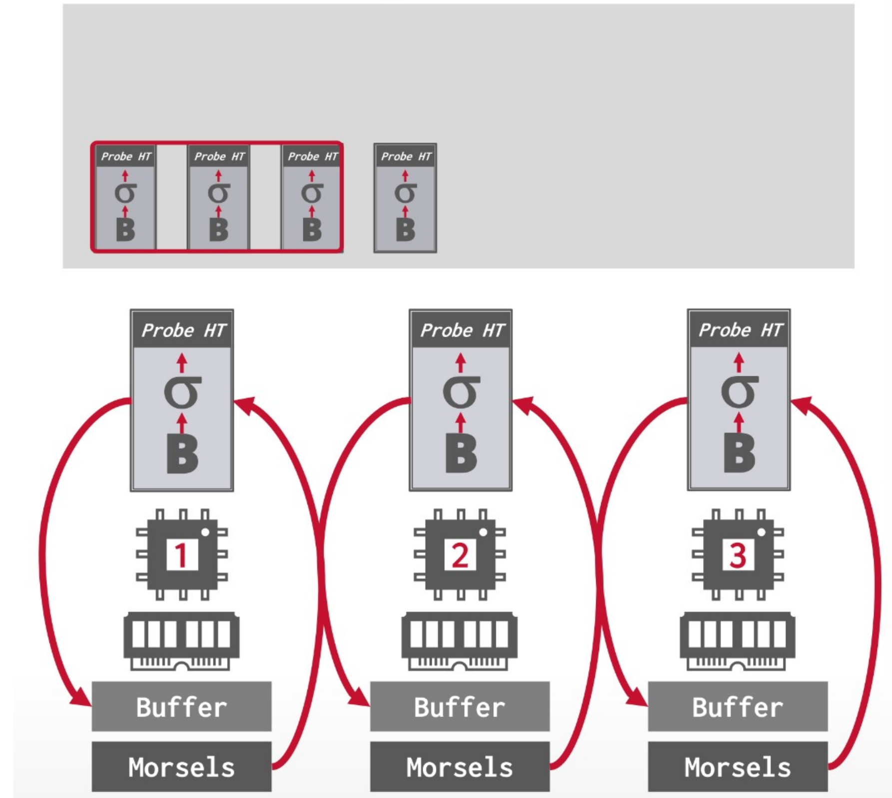




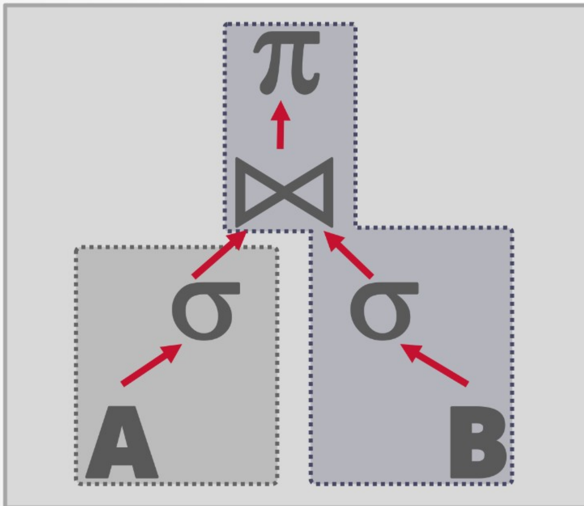


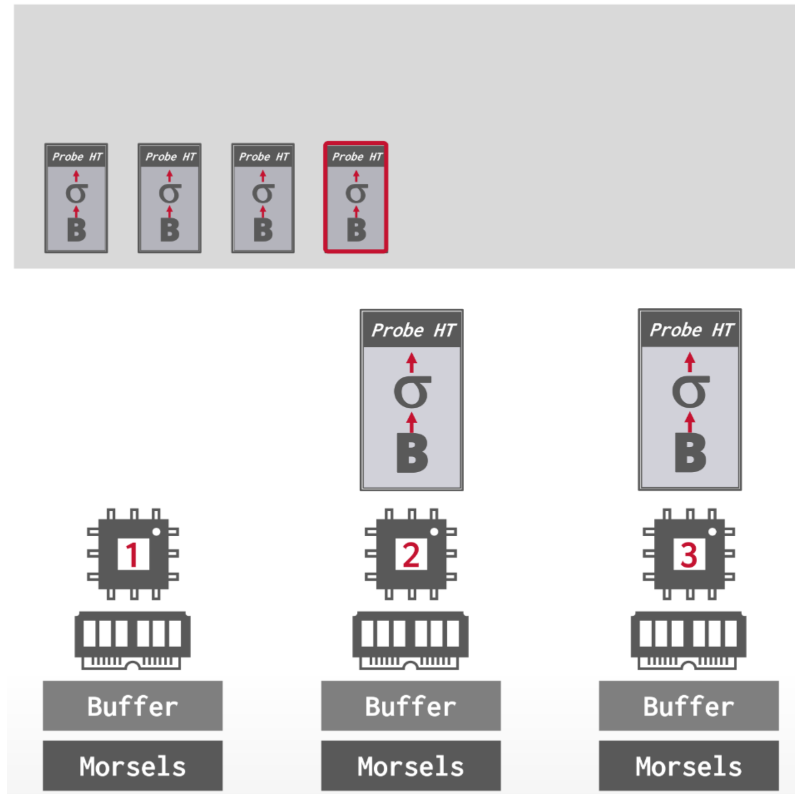
HYPER: EXECUTION EXAMPLE



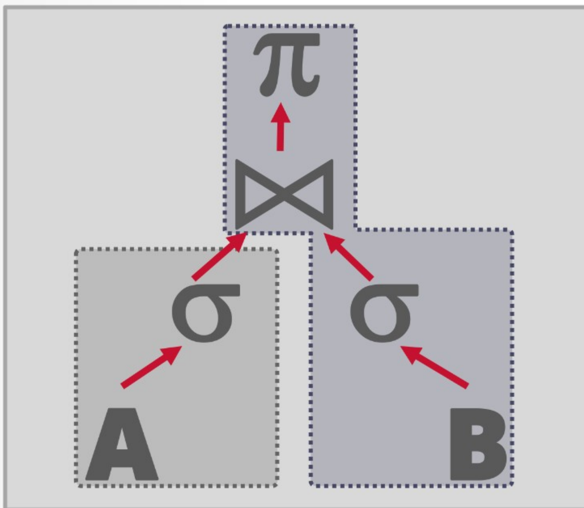


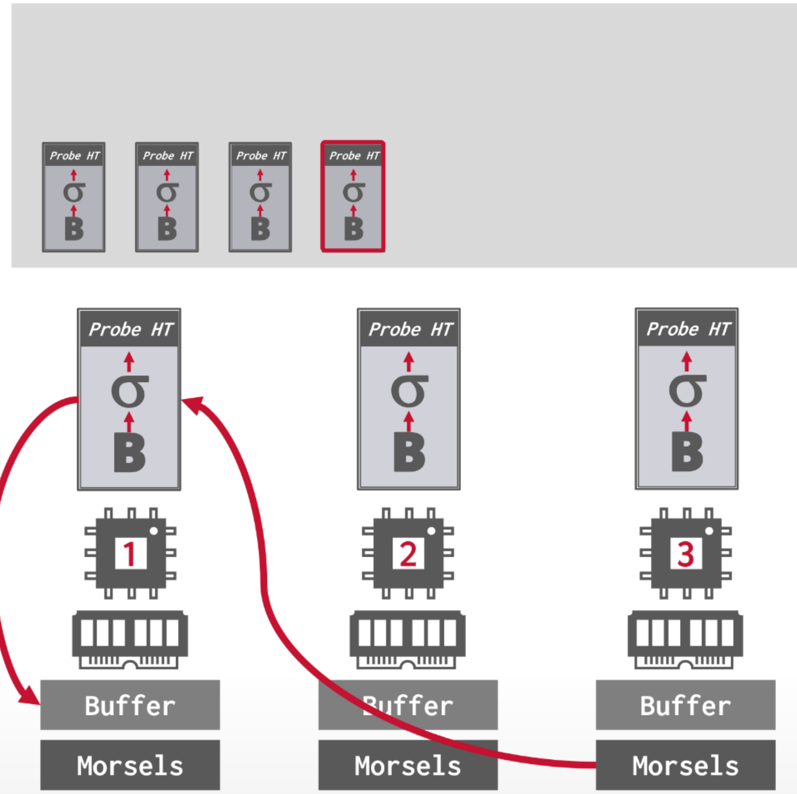
HYPER: EXECUTION EXAMPLE





HYPER: EXECUTION EXAMPLE





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 on data chunks (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**.

STRIDE SCHEDULING

Each job in the system has a **stride**, which is inverse in proportion to the number of **tickets** it has.

Every time a process runs, we will increment a counter for it which is called its **pass value**.

Scheduler then uses the stride and pass to determine which process should run next.

```
curr = remove_min(queue);      // pick client with min pass
schedule(curr);                // run for quantum
curr->pass += curr->stride;    // update pass using stride
insert(queue, curr);           // return curr to queue
```

STRIDE SCHEDULING - EXAMPLE

We start with three processes (A, B, and C), with stride values of 100, 200, and 40, and all with pass values initially at 0.

Pass(A) (stride=100)	Pass(B) (stride=200)	Pass(C) (stride=40)	Who Runs?
0	0	0	A
100	0	0	B
100	200	0	C
100	200	40	C
100	200	80	C
100	200	120	A
200	200	120	C
200	200	160	C
200	200	200	...

KEY COMPONENTS

Slot -- Worker thread (usually pinned to a CPU core). Continuously fetches morsels to execute.

Morsel (Task) -- A small, independent unit of work: *data chunk* (10^3-10^5 tuples) + *pipeline code*.

Task Set -- A group of morsels for one task (pipeline stage). Task sets are ordered in a DAG by the relation depend-on..

Global Queue -- Holds all ready-to-run morsels from all queries. Scheduling happens here.

Local Query Queue -- A queue of task sets, used to generate new morsels as dependencies finish.

Scheduler -- Picks morsels from the global queue according to query priorities (“pass values”).

UMBRA: SCHEDULING STATE

There is a single **global queue** of tasks (morsels) and a **local queue** of task sets for each query.

Scheduling operates on morsels from the global queue but on the level of queries.

- Queries, not task sets or morsels (tasks), have priorities!
- The scheduler selects the next morsel from the query with lowest pass value.

A task set is **added** to the global queue either when query arrives, or after a slot completes work on a task set.

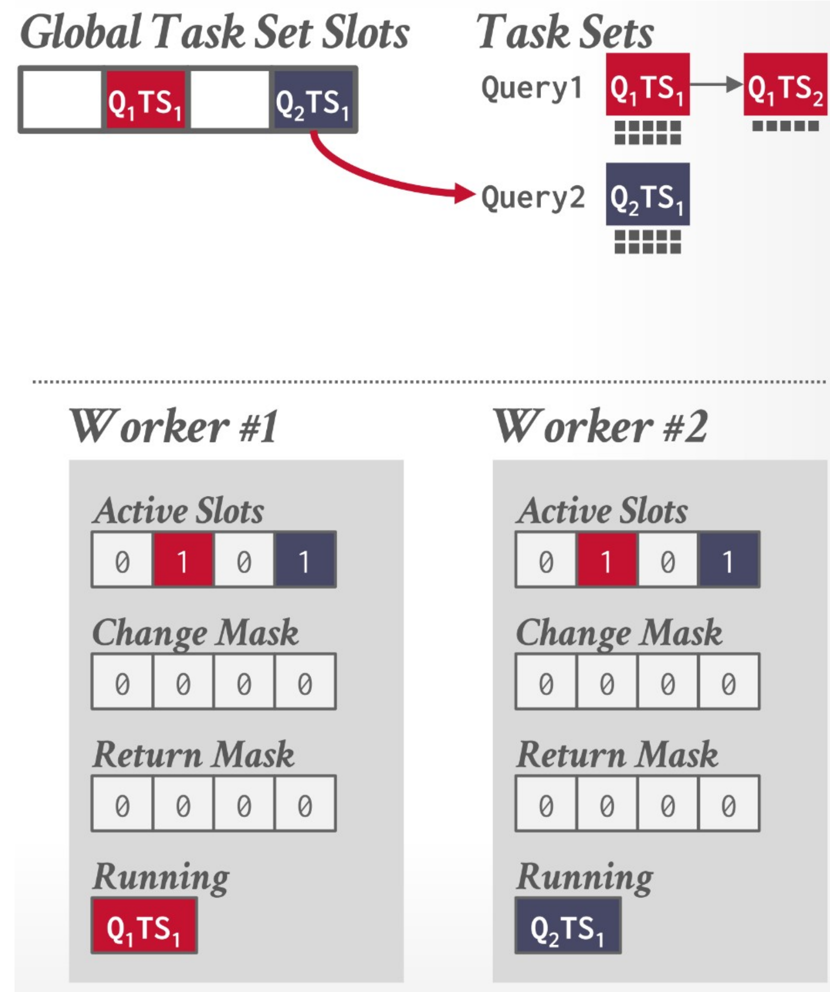
- In the former case adding is done by scheduler; in the latter, it is triggered by the slot that finishes a task set.

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.

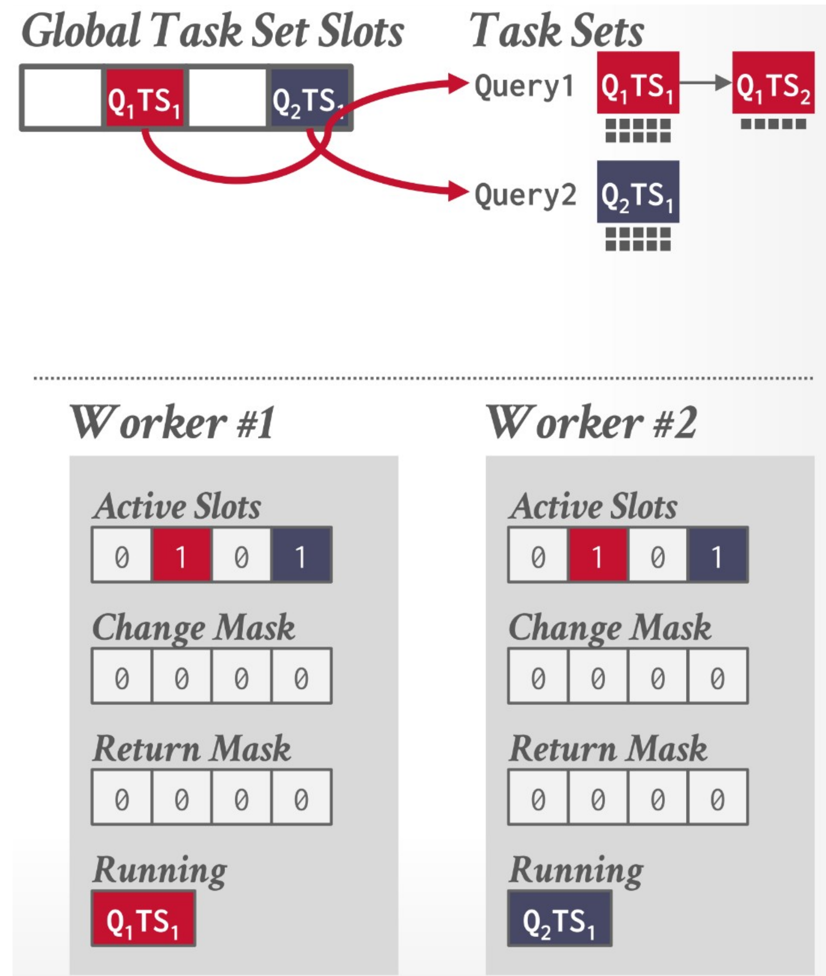


UMBRA: SCHEDULING STATE

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.



EXECUTION FLOW

1) **Query submission**
   - When a query arrives, its first ready task set(s) are created.
   - The scheduler enqueues their morsels into the global queue.
2) **Morsel scheduling**
   - Each slot repeatedly fetches a morsel from the global queue.
   - Selection is query-aware: the scheduler chooses morsels from the query with the lowest pass value.
3) **Task-set progression**
   - As slots complete morsels, they mark their completion bits in the task set's return mask.
   - When all bits are set then the task set is complete.
   - Dependent task sets (next stage in the pipeline DAG) are now activated, and their morsels are enqueued into the global queue.

EXECUTION FLOW

4) Parallelism and fairness

- Multiple slots can execute morsels from the same task set concurrently.
- Idle slots can pick morsels from any query, respecting query priority.

5) Completion

- When a query’s final task set finishes, the query is complete and its resources are freed.

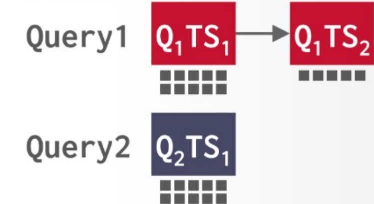
UMBRA: SCHEDULING STATE

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.

Global Task Set Slots

Task Sets



Worker #1

Active Slots

Change Mask

Return Mask

Running

Worker #2

Active Slots

Change Mask

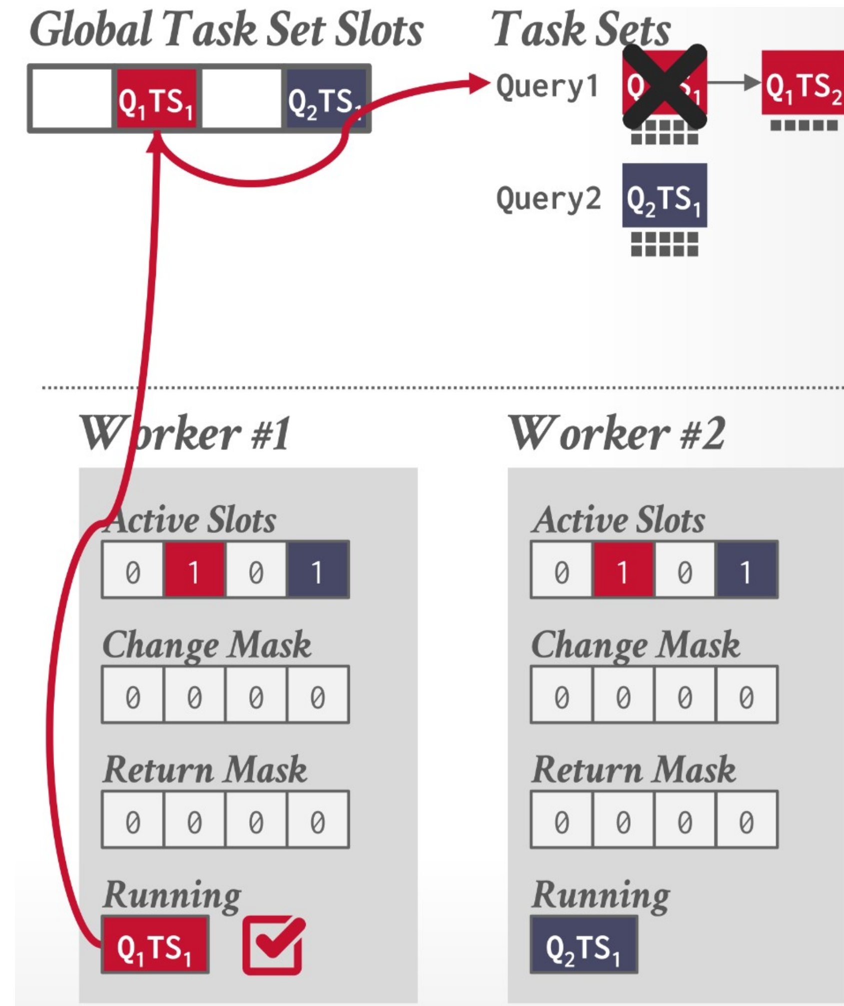
Return Mask

Running

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.

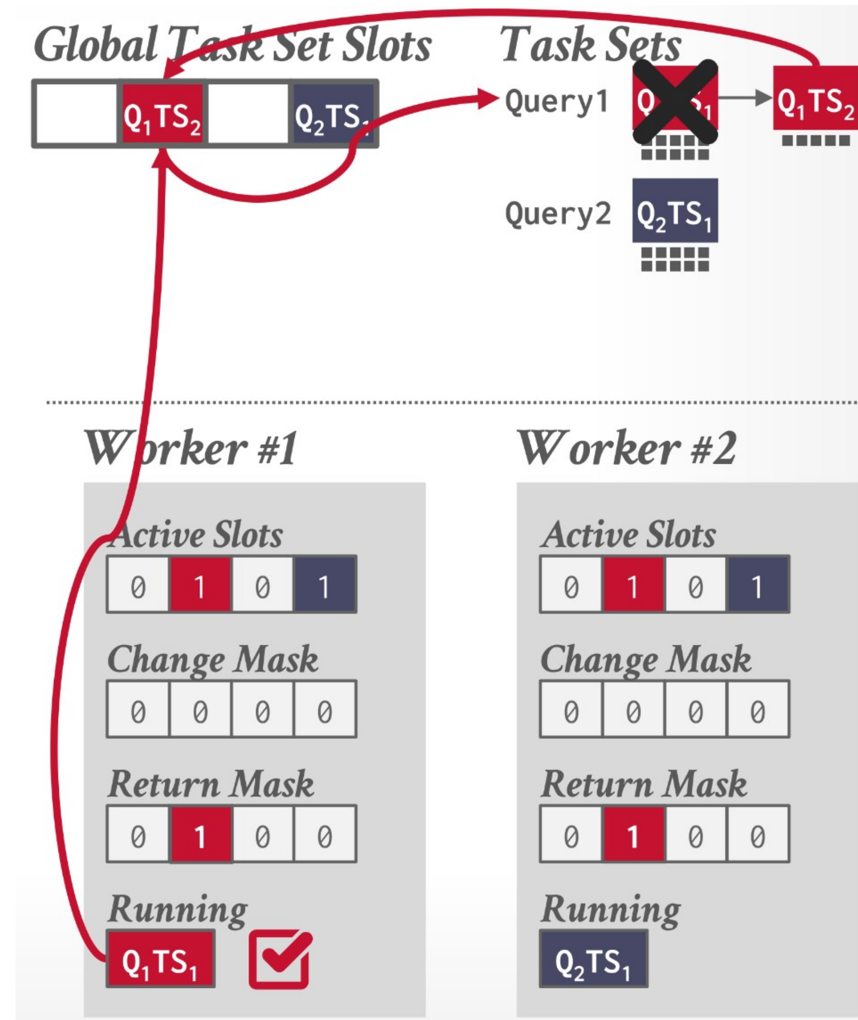
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



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.

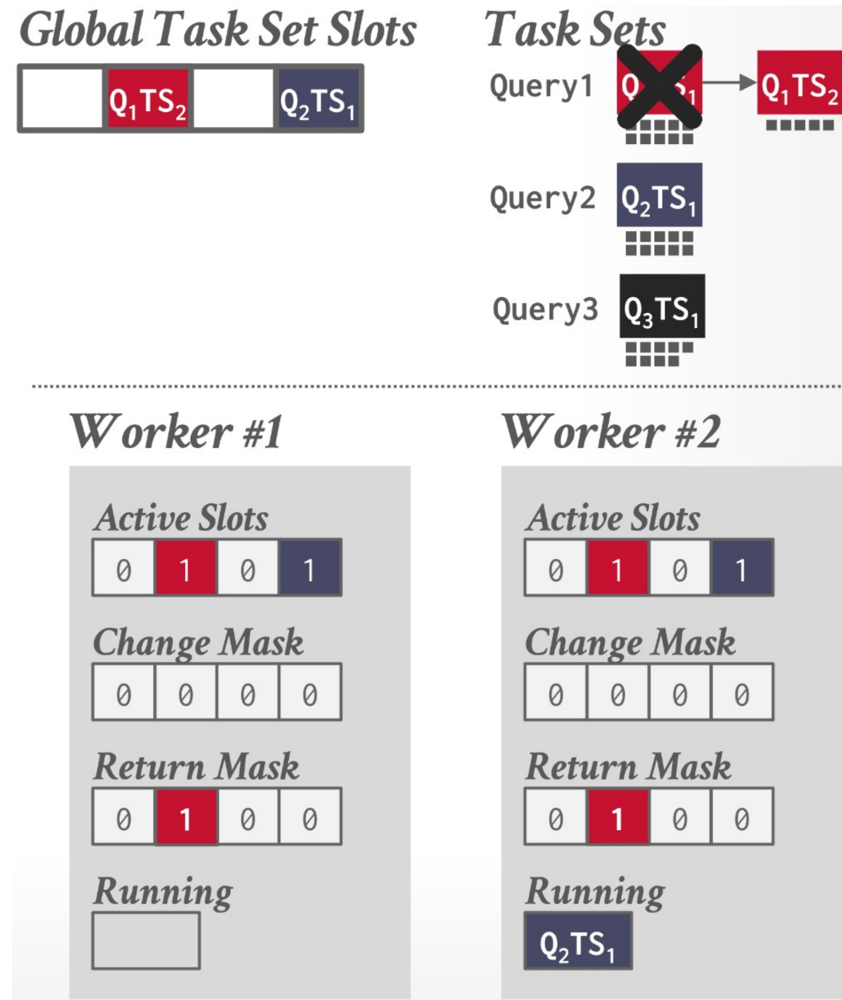
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



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.

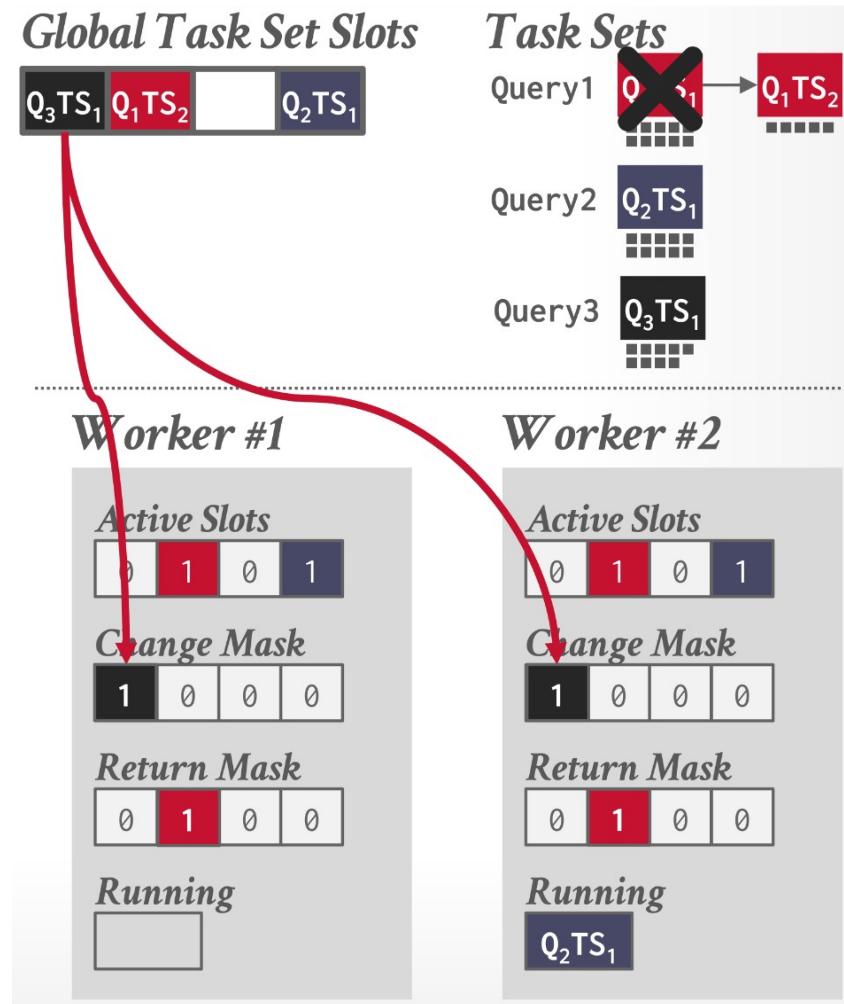
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



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.

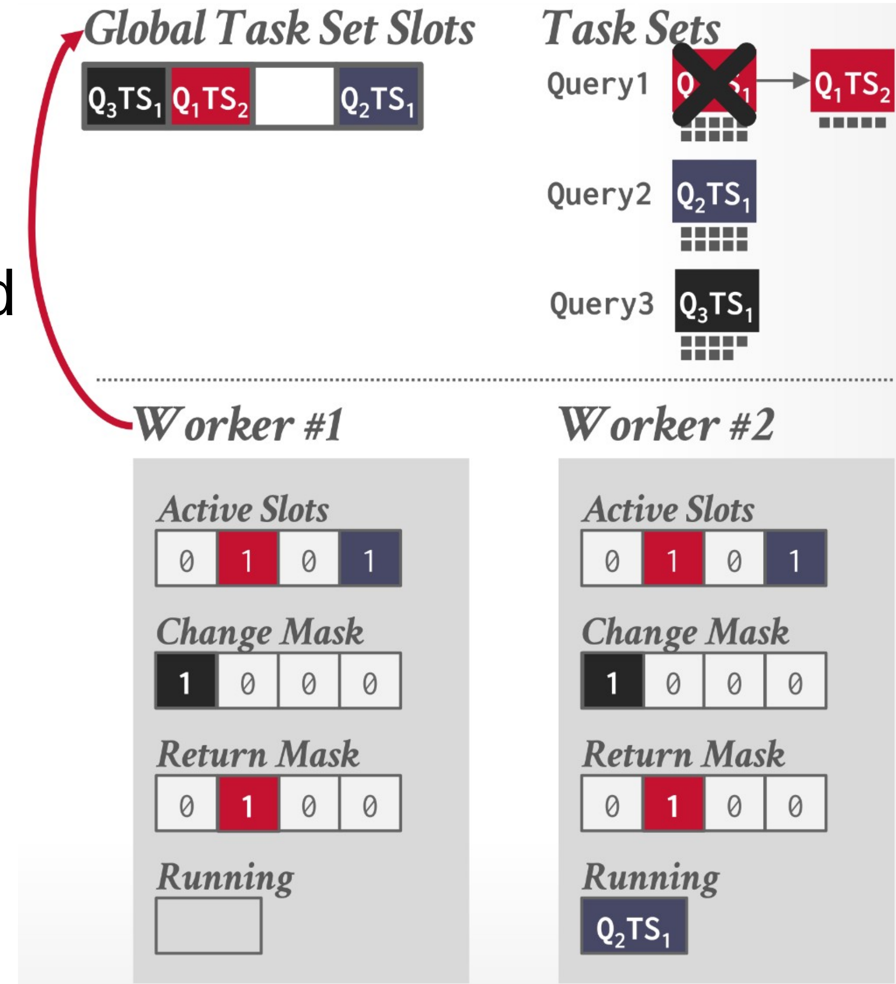
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



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.

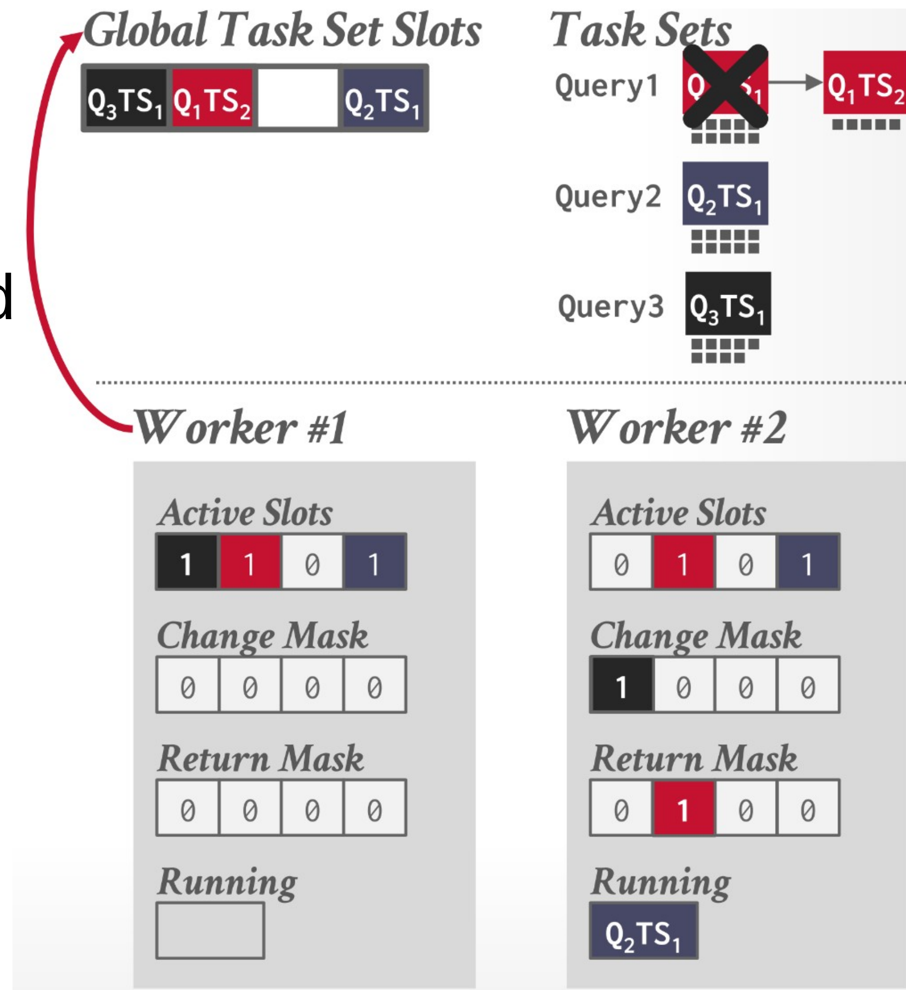
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



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.

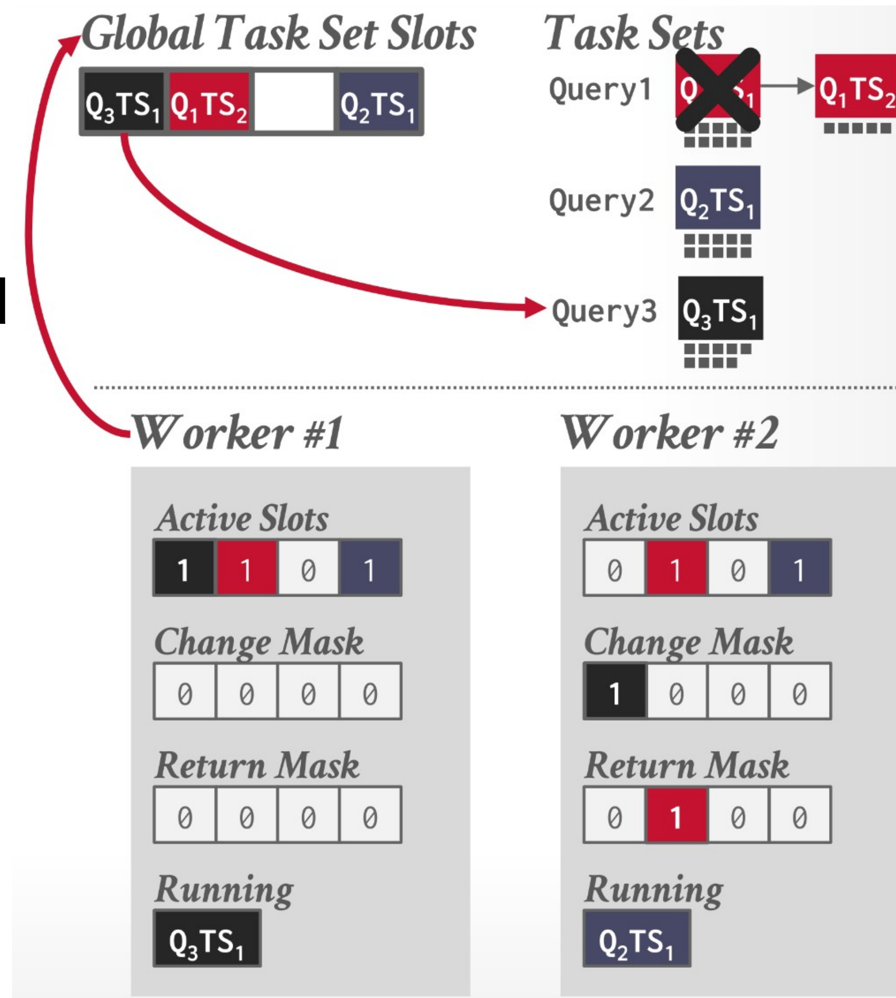
When a new a query arrives, the scheduler updates the workers' **Change Mask** to inform them of the new query tasks in a slot.



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

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.

Worker #1

Global Pass Value
0.3

Pass Values
	0.0		0.5

Priorities
	1		2

Active Slots
0	1	0	1

Change Mask
0	0	0	0

Return Mask
0	0	0	0

Running
Q_1TS_1

Worker #2

Global Pass Value
0.6

Pass Values
	1.0		0.5

Priorities
	1		2

Active Slots
0	1	0	1

Change Mask
0	0	0	0

Return Mask
0	0	0	0

Running
Q_2TS_1

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 not let the OS ruin your life.

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