



# Scheduling

CREATING THE NEXT®

# Today's Agenda

---

Scheduling

Recap

Data Placement

Worker Allocation

Scheduling – Hyper

Scheduling – HANA

Scheduling – SQL Server

Flow Control

Conclusion

Retrospective

# Recap

# 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 running on the server that is responsible for executing tasks on behalf of the client and returning the results.
- Approaches
  - ▶ Approach 1: Process per DBMS Worker
  - ▶ Approach 2: Process Pool
  - ▶ Approach 3: Thread per DBMS Worker

# Execution Parallelism

---

- **Inter-Query**: Different queries are executed concurrently.
  - ▶ Increases throughput & reduces latency.
- **Intra-Query**: Execute the operations of a single query in parallel.
  - ▶ Decreases latency for long-running queries.

# I/O Parallelism

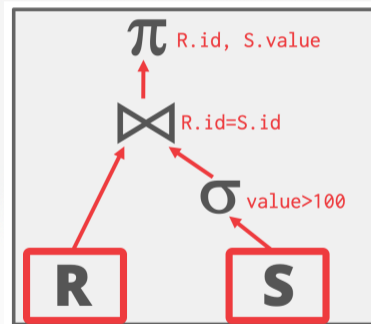
---

- Split the DBMS installation across multiple storage devices.
  - ▶ Multiple Disks per Database
  - ▶ One Database per Disk
  - ▶ One Relation per Disk
  - ▶ Split Relation across Multiple Disks

# 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 (*a.k.a.*, a **pipeline**).

```
SELECT R.id, S.cdate
FROM R, S
WHERE R.id = S.id AND S.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.



# Today's Agenda

---

- Data Placement
- Worker Allocation
- Scheduling
  - ▶ Hyper
  - ▶ HANA
  - ▶ SQL Server
- Flow Control

# Data Placement

# Observation

---

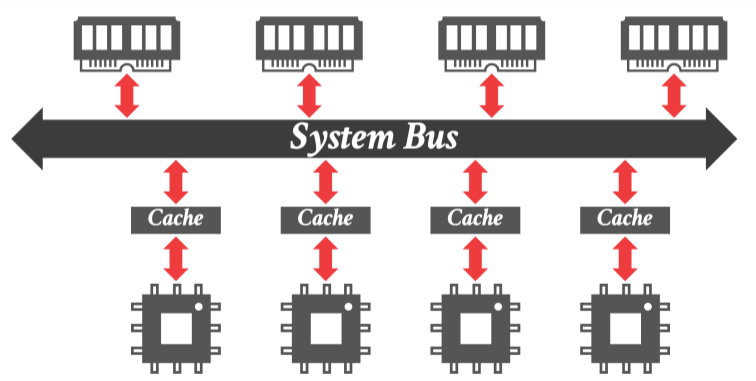
- Regardless of what worker allocation or task assignment policy the DBMS uses, it's important that workers operate on **local data**.
- The DBMS's scheduler must be aware of its hardware memory layout.
  - ▶ Uniform vs. Non-Uniform Memory Access
- Reference

# Uniform Memory Access

---

- Cost of accessing data from a CPU core to any memory bank is roughly the same.
- Need to access data through the system bus.
- *a.k.a.*, Symmetric multi-processors (SMP).
- If two CPUs have a memory location in their caches and one of them does a write, then that CPU must send a cache invalidation message over the bus to the other CPU.

# Uniform Memory Access

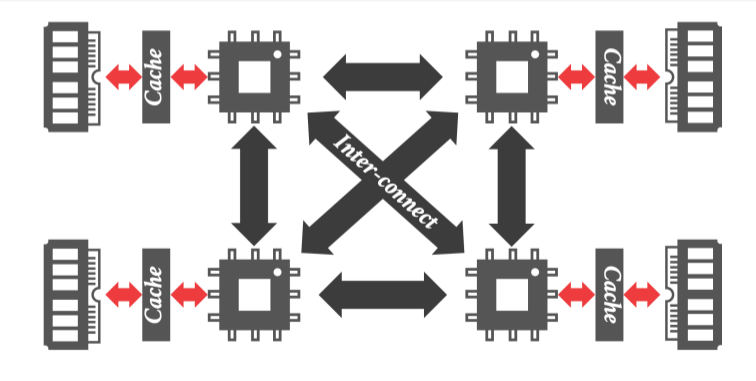


# Non-Uniform Memory Access

---

- Every core has its own L1/L2 cache.
- All cores on the same **socket** share an L3 cache.
- Cost of accessing data from a CPU core to any memory bank is **not uniform**.
  - ▶ Intel (2008): QuickPath Interconnect
  - ▶ Intel (2017): UltraPath Interconnect
  - ▶ AMD (2017): Infinity Fabric

# Non-Uniform Memory Access



# Data Placement

---

- The DBMS can partition memory for a database and assign each partition to a CPU.
- Same problem arises in **distributed DBMSs**.
- By controlling and tracking the location of partitions, it can schedule operators to execute on workers at the closest CPU core.
- Linux Support
  - ▶ move\_pages: moves the specified pages to the given memory nodes
  - ▶ numactl: runs processes with a specific NUMA scheduling or memory placement policy.
  - ▶ cpunodebind: Only execute command on the CPUs of given nodes.
  - ▶ membind: Only allocate memory from nodes.



# Memory Allocation

---

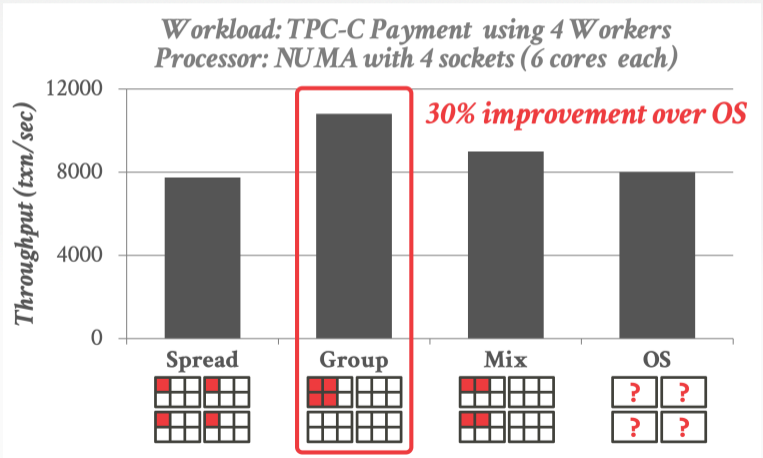
- What happens when the DBMS calls malloc?
  - ▶ Assume that the allocator doesn't already have a chunk of memory that it can give out.
- Almost nothing:
  - ▶ The allocator will extend the process' data segment.
  - ▶ But this new **virtual memory** is not immediately backed by physical memory.
  - ▶ 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.
  - ▶ Default policy that works well for most applications.
- **Approach 2: First-Touch**
  - ▶ At the CPU of the thread that accessed the memory location that caused the page fault.
  - ▶ Better policy for DBMSs.
- The OS can try to move memory to another NUMA region from observed access patterns.

# Data Placement - OLTP

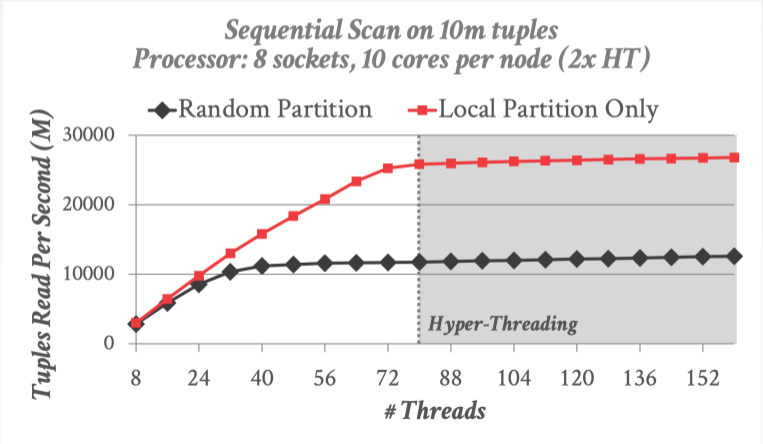


## Data Placement - OLTP

---

- Spread: assigns each thread to a core in a different socket.
- Group: assigns all threads to the same socket.
- Mix: assigns two cores per socket.
- OS: let the operating system do the scheduling.

# Data Placement - OLAP



# Data Placement - OLAP

---

- We are always processing the same number of tuples.
- Performance gap is smaller with fewer threads since more tuples are local to the core.
- With hyper-threading, no significant performance improvement since we are bottlenecked by memory bandwidth.
- So adding more logical cores doesn't help (already waiting for cacheline fills).

# Partitioning vs. Placement Schemes

---

- 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.
  - ▶ Place the partition on a single socket
  - ▶ Distribute the partition across all sockets

# Worker Allocation



# Observation

---

- Determining the right number of workers to use for a query plan depends on:
  - ▶ the number of CPU cores.
  - ▶ the size of the data.
  - ▶ the 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.
  - ▶ `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.

# Scheduling – Hyper

# Observation

---

- We have the following so far:
  - ▶ Process Model
  - ▶ 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 when it generates the plan.
- It does not change while the query executes.
  - ▶ The easiest approach is to just use the same number of tasks as the number of cores.
  - ▶ Can still assign tasks to threads based on data location to maximize local data processing.
- Limitation: our assumption about the distribution of data can be wrong.
- This leads to stragglers.

# Dynamic 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.
- Duplicate or steal tasks to avoid stragglers.
- **Reference**

# Architecture

---

- No centralized dispatcher thread (*i.e.*, pull model)
- 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.

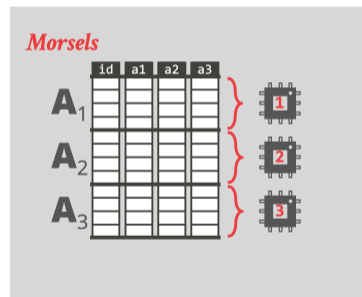


# Data Partitioning

---

- Morsel is a Hyper term.
- Number of tuples to provide the right amount of parallelism (e.g., 100 K tuples)
- Slightly bigger than a block, smaller than a partition.

*Data Table*

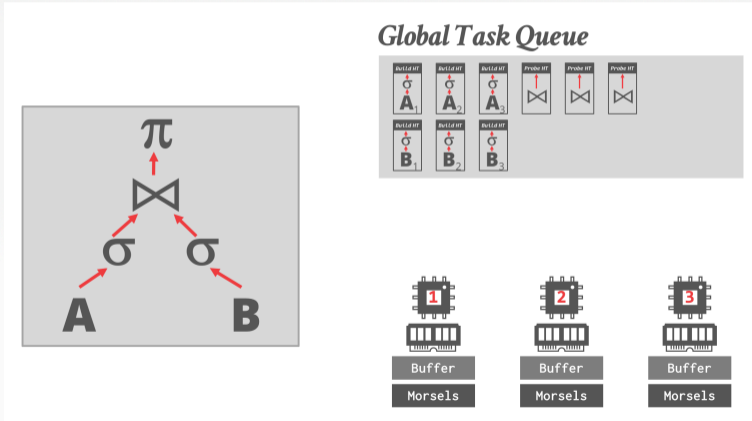


# Morsel-Driven Dynamic Scheduling

---

- Because there is only one worker per core, 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.

# Example

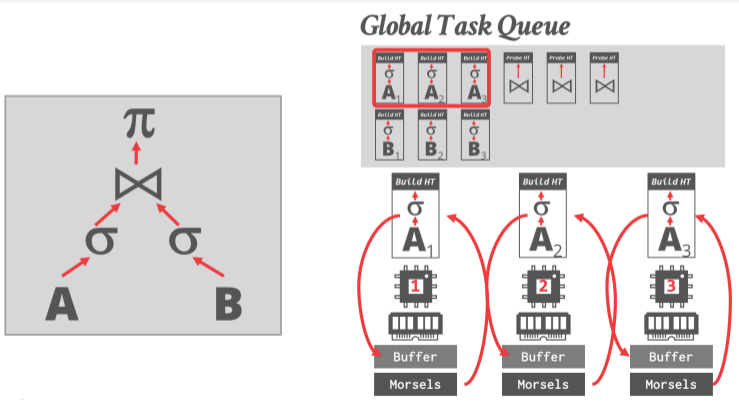


## Morsel-Driven Dynamic Scheduling

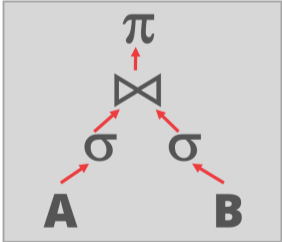
---

- Each worker will have the morsels stored locally.
- As the workers execute tasks, they will store the output in their local buffers (rather than a shared global buffer).
- When they select the next task, they try to pick ones that will maximize the reuse of morsels in their local buffers.
- This scheduling policy minimizes cross-communication between workers.

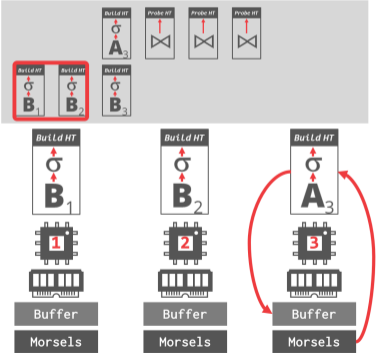
# Example



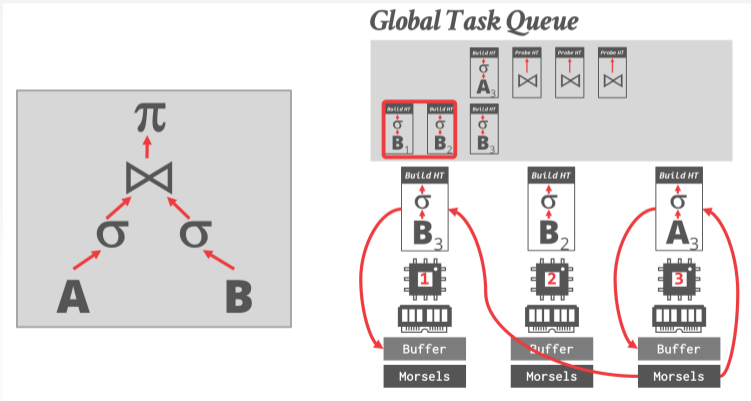
# Example



## Global Task Queue



# Example



# Scheduling – HANA



# NUMA-Aware Scheduler

---

- Pull-based scheduling with multiple worker threads that are organized into **groups**.
  - ▶ Each CPU can have multiple groups.
  - ▶ The scheduler can scale up/down the number of threads in a group
- Uses a separate **watchdog thread** to check whether groups are saturated and can reassign tasks dynamically.
- [Reference](#)

# Thread Groups

---

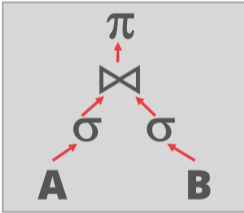
- Each thread group has a soft and hard priority queue.
  - ▶ **Soft queue:** Threads **can** steal tasks from other groups' soft queues.
  - ▶ **Hard queue:** Threads **cannot** steal tasks from other groups' hard queues (*e.g.*, garbage collection, networking).
- Four different pools of threads per 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.

## NUMA-Aware Scheduler

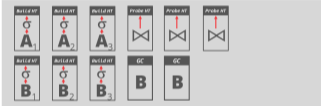
---

- 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 (e.g., 64 sockets).
- If you have too many sockets, then put all tasks in the hard queue to prevent stealing.
- Using thread groups allows cores to execute other tasks instead of just only queries.

# Example



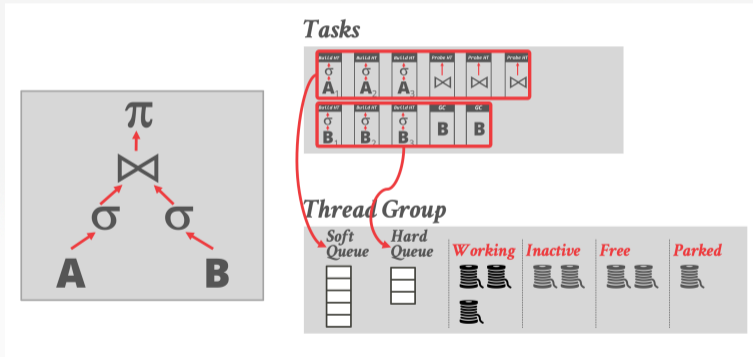
### Tasks



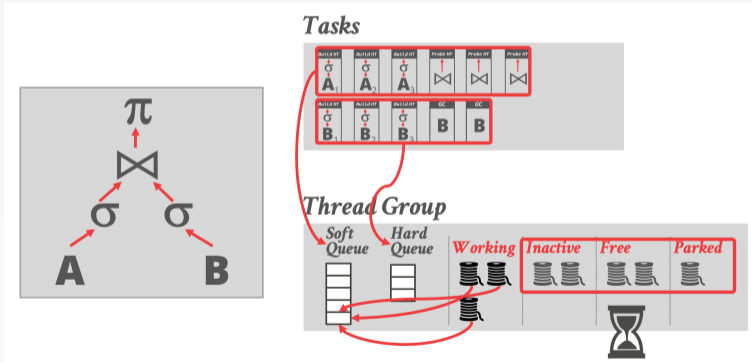
### Thread Group



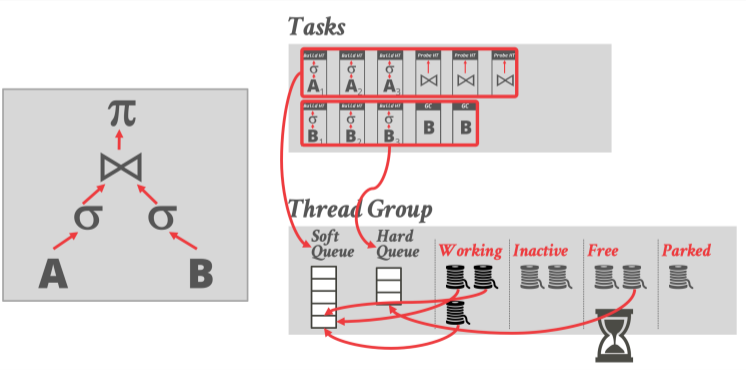
# Example



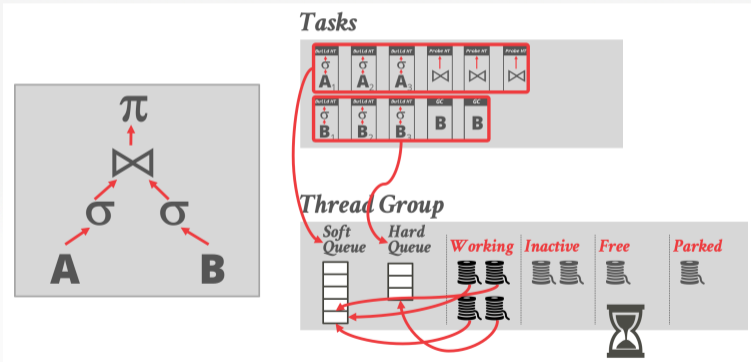
# Example



# Example



# Example





# Scheduling – 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.
- [Reference](#)

# SQLOS

---

- **Quantum** is the amount of time that the scheduler allows a thread to run before making a new decision.
- SQLOS quantum is 4 ms but the scheduler **cannot** enforce that.
- Linux: Quantum length is not fixed. 100 ms for special-purpose real-time processes.
- DBMS developers must add explicit **yield** calls in various locations in the source code.

# SQLOS

---

```
SELECT *  
  ~IFROM A  
  ~IWHERE A.val = ?  
  
last = now()  
for t in range(table.num_tuples):  
    tuple = get_tuple(table, t)  
    if eval(predicate, tuple, params):  
        emit(tuple)  
    if now() - last > 4ms:  
        yield  
        last = now()
```

# Flow Control

# Observation

---

- If requests arrive at the DBMS faster than it can execute them, then the system becomes overloaded.
- The OS cannot help us here because it does not know what threads are doing:
  - ▶ CPU Bound: Do nothing
  - ▶ Memory Bound: Out-of-memory error
- Easiest DBMS Solution: Crash

# Flow Control

---

- **Approach 1: Admission Control**
  - ▶ Abort new requests when the system believes that it will not have enough resources to execute that request.
- **Approach 2: Throttling**
  - ▶ Delay the responses to clients to increase the amount of time between requests.
  - ▶ This assumes a synchronous submission scheme.

# Conclusion



# Conclusion

---

- A DBMS is a beautiful, strong-willed independent piece of 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
- Don't let the OS ruin your life.

# Retrospective

# What did we learn

---

- You are tired of systems programming
- You are exhausted
- Let's take a step back and think about what happened

## Lessons learned

---

- Systems programming is hard
- Become a better programmer through the study of database systems internals
- Going forth, you should have a good understanding how systems work

# Big Ideas

---

- Database systems are awesome – but are not magic.
- Elegant abstractions are magic.
- Declarativity enables usability and performance.
- Building systems software is more than hacking
- There are recurring motifs in systems programming.
- CS has an intellectual history and you can contribute.

## What Next?

---

- We have barely scratched the surface. Follow-on course: CS 8803 (DBMS Implementation - Part II)
  - ▶ Query Optimization
  - ▶ Concurrency Control
  - ▶ Logging and Recovery Methods
  - ▶ Query Compilation + Vectorization
- Stay in touch
  - ▶ Tell me when this course helps you out with future courses (or jobs!)
  - ▶ Ask me cool DBMS questions

# Parting Thoughts

---

- You have surmounted several challenges in this course.
- You make it all worthwhile.
- Please share your feedback via CIOS.