Lecture #08: Scheduling & Coordination

15-721 Advanced Database Systems (Spring 2024)
https://15721.courses.cs.cmu.edu/spring2024/
Carnegie Mellon University
Prof. Andy Pavlo

1 Query Execution

A query plan generated by the optimizer is comprised of a DAG 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 task set is a collection of executable tasks for a logical pipeline.

Scheduling

For each query plan, the DBMS has to decide where, when and how to use it in a multi-core, multi-threaded environment. The DBMS needs to know the number of tasks it should use to divide the query. We also need to figure out how many CPU cores the DBMS should use and which CPU cores to execute the tasks on. After a task is completed, the DBMS needs to know where to store the output. Instead of leaving the details of scheduling to the OS, the DBMS should handle scheduling on its own as much as possible. The DBMS has more understanding on the functionality of queries and the shape of the data, so it is in a better position of deciding specific scheduling policies. We discuss scheduling for a single system, but these ideas extend to a distributed environment.

Goals

We want the scheduler to achieve high throughput with low scheduling overhead, maintain fairness (reduce starvation), and minimize tail latencies. Short queries, in particular, should be prioritized over long-running queries to increase responsiveness as it's much more obvious when a short query is taking longer than it should.

2 Process Models

We first focus on how to design a multi-threaded, multi-worker processing system for a DBMS. 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. There are three different types of process models: *Process per DBMS Worker*, *Process Pool* and *Thread per DBMS Worker*.

Process Per Worker

Each worker is a separate OS process. It has its own pid and address space that is managed by the OS. Therefore, the only way for the workers to communicate with each other is through shared-memory. A process crash in this model also does not take down the entire system.

When a client sends out a request, it is received by the dispatcher that is responsible for handing out the request to some worker. The worker communicates with the client to take care of all the client's future requests. It then runs all the requests on the DBMS and returns the results to the client.

Back in 1980–1990s, `pthread` does not exist yet and there is no standard threading package. Since most OSs support multi-processes in shared-memory, so this approach is used in older DBMSs.

Examples: **IBM DB2**, **Postgres**, **Oracle**

Process Pool

Process Pool is a variant model of Process Per Worker and still relies on the OS scheduler and shared-memory.

Instead of binding a particular worker to a client, the dispatcher can hand off the request to any worker available in the process pool. To process queries in parallel, workers in the process pool can communicate with each other to divide tasks using shared-memory. Since all the workers still have their own separated address space, this approach can cause slowdowns due to bad CPU cache locality.

Examples: **IBM DB2**, **Postgres** (since 2015)

Thread Per Worker

Use a single process with multiple worker threads. This model no longer relies on the OS scheduling; the DBMS has to manage its own scheduling. The DBMS uses a pool of worker threads to handle requests (it may or may not use a dispatcher thread). The threads can write to shared address space that is protected by latches. However, a thread crash (may) kill the entire system.

Using a multi-threaded architecture has several advantages. It is easier to engineer, since we do not have to worry about shared memory and `pthread` is readily available. All the threads run in the same address space, so there is less overhead per context switch.

The thread per worker model does not imply it yields intra-query parallelism. Even if the DBMS is multi-threaded, it may still be executing one thread at a time (e.g., **MySQL**).

Examples: **IBM DB2**, **MSSQL**, **MySQL**, **Oracle** (since 2014)

3 Woker Allocation

There are two ways of assigning workers to CPU cores. The first way pins one worker to a CPU core using the OS syscall - see `sched_setaffinity`. The second way assigns multiple workers to a CPU core. The first approach prevents contention between multiple threads, such as thrashing the L3 cache, but will result in a stalled CPU during IO. However, the first approach is preferred for maximum performance because we assume database systems will be compute bound, so a stalled CPU should be infrequent with good caching and prefetching practices. Note that the first approach would turn off hyperthreading, so each core has only one hardware thread.

4 Data Placement

Regardless of what worker allocation or task assignment policy the DBMS uses, it is important that workers operate on local data. The DBMS should be aware of it's underlying hardware's memory layout: whether it is uniform or non-uniform memory access.

5 Task Assignment

There are two ways of assigning tasks - push vs pull. The push approach consists of a centralized dispatcher that pushs tasks to the workers and tracks their progress. When a worker finishes a task, it notifies the dispatcher and is given a new task. The pull approach maintains a queue of tasks which the workers pull from, process, and return to retrieve more tasks. The main difference between these two approaches is that the pull based approach "decentralizes" the dispatching - all workers cooperate and make decisions on their own as to which task to process next from the queue.

Regardless of which approach the DBMS uses, workers should strive to process local data. Either the scheduler or the workers will need to know where the data lies relative to the workers, depending on the approach.

Uniform Memory Access

Each CPU has its own local cache. In order to read data from the memory DIMMs, the requests would go through the bus. The bus retrieves the data and handles cache invalidations. The cost of getting data from any one DIMM is roughly the same. The hardware covers lower-level details and provides cache-coherent guarantees.

Non-Uniform Memory Access (NUMA)

A newer form of memory access is NUMA. Each socket has its own cache and set of DIMMs. The DIMMs are physically close to the CPU, so the CPU can read/write to local memory quickly. The cost of getting data from a DIMM is higher if it is in a farther physical location. When a CPU tries to access a memory location in a different CPU, it has to go through the inter-connect to that socket and cache the results. Therefore, for in-memory DBMS, it is important for the DBMS know where to run its tasks in order to operate on data in close physical proximity.

The different CPU vendors refer to this inter-connect channel by different marketing terms: *Intel UltraPath Interconnect* (2017) and *AMD Infinity Fabric* (2017).

Data Placement

The database 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.

Memory Allocation

When the DBMS calls `malloc`, if the allocator does not already have a chunk of memory that it can give out, it will request memory from the OS. The allocator will extend the process's data segment. However, this new virtual memory is not immediately backed by physical memory; the OS only allocates physical memory when there is a page fault. There are two possible ways for the OS to allocate physical memory in a NUMA system:

- **Interleaving:** Distribute allocated memory uniformly across CPUs.
- **First-Touch:** Distributed allocated memory at the CPU of the thread that accessed the memory location that caused the page fault. This approach is more favored as physical proximity increases CPU access in NUMA systems.

Partition vs. Placement

A DBMS's *partitioning scheme* splits the database into disjoint subsets based on some policy. Some common partitioning schemes are: Round-robin, Attribute Ranges, Hashing and Partial/Full Replication. The partition policy will depend on a target objective, for example, minimizing the execution time of joins.

For OLAP systems, partitioning is usually file-based Round-robin due to its simplicity, as we cannot easily write/manipulate object stores in a shared-disk Lakehouse environment.

A DBMS's *placement scheme* tells the DBMS where to put the partitions. Common placement schemes include Round-robin and Interleave Across Cores.

6 Scheduling

The DBMS uses scheduling to decide how to create a set of tasks from a logical query plan.

Static Scheduling

The DBMS decides how many threads to use to execute the query when it generates the plan. Nothing changes while the query executes. The easiest approach is to have the DBMS use the same number of tasks as the number of cores.

This approach assumes the data is uniformly-distributed; each socket is expected to have the same amount of work. However, in practice, there are often discrepancies between the amount of work executed on each socket.

HyPer: Morsel-Driven Scheduling

The DBMS dynamically schedules tasks that operate over horizontal partitions called "morsels" that are distributed across cores [1]. Each "morsel" consists of about 100k tuples determined empirically for the best performance. This approach uses one worker per CPU core, pull-based task assignment and round-robin data placement across the sockets. It also supports parallel, NUMA-aware operator implementations.

In **HyPer**, the DBMS uses pull-based task assignments instead of separated dispatcher threads. The threads 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. This will help mitigate the issue of stragglers.

Note that the global task queue suffers from synchronization costs when the number of workers is high. In **Umbra**, the global task queue is replaced with a more scalable distributed approach.

HyPer Execution Example

Each Hyper worker has its own memory buffer, on which it stores the intermediate results of the completed tasks. Should there be idle workers and tasks remaining in the global queue, the idle workers will retrieve the tasks even if the data is not colocated in the same NUMA region.

Umbra: Morsel Scheduling 2.0

Umbra introduces two improvements to the original Morsel scheduling. First, it assigns morsel sizes dynamically. In the original paper, morsels are static fixed size chunks. However, depending on selectivity of filters, different chunks may experience very different processing times. In Umbra, tasks are monitored and sized according to a target of 1ms per task (exponential changes to the morsel size). This introduces more uniform morsel processing times. Second, it introduced automatic priority decays to long running queries by using a modern implementation of stride scheduling. This ensures shorter queries are not starved for resources.

Umbra Execution Example

Umbra workers are responsible for updating the global task queue when their tasks are finished and notifying all other workers of the update through a bit-mask. When a new query is added to the task queue, the scheduler is responsible for notifying all workers of the new update though a (differet) bit-mask.

SAP HANA: NUMA-Aware Scheduler

Because there is only one worker per core, **HyPer** has to use work stealing because otherwise threads could sit idle waiting for stragglers. A solution to this is to use a lock-free hash table to maintain the global work queues.

SAP HANA uses pull-based scheduling with multiple worker threads that are organized into *thread groups* (i.e., pools) [2]. This NUMA-Aware Scheduler can dynamically adjust thread pinning based on whether a task is CPU memory bound. Each CPU can have multiple groups, and each group has a soft and hard priority queue. The DBMS uses a separate "watchdog" thread to check whether groups are saturated and can reassign tasks dynamically.

Each thread group has a soft and hard priority task queues. Threads are only allowed to steal tasks from other groups' soft queues. There are 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 does not wake up on its own.

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

Researchers also found that work stealing was not as beneficial for systems with a larger number of sockets. The overhead of moving data around on a machine becomes problematic on a larger scale.

7 SQLOS

SQLOS is a user-mode NUMA-aware OS layer that runs inside the DBMS and provides hardware resource abstraction. It provides scheduling support through coroutines (non-preemptive thread scheduling). A quantum in SQLOS is 4ms, but it cannot enforce this and relies on developer-placed yields in the source code.

8 Flow Control

If requests arrive at the DBMS faster than it can execute them, then the system becomes overloaded. The OS cannot help the DBMS. If the DBMS is memory bound, an out-of-memory (OOM) error is returned. If the DBMS is CPU bound, the requests queue will keep stalling and stacking future requests until the DBMS runs out of memory. The easiest DBMS solution to this is to simply crash. We introduce two additional control methods that are often used together in a DBMS:

Admission Control: Abort new requests when the system believes that it will not have enough resources to execute that request. This approach appears more often in newer DBMS systems.

Throttling: Delay the responses to clients to increase the amount of time between requests. This approach assumes a synchronous submission scheme (i.e., the client waits for response before sending the next request).

References

[1] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 743–754, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323765. doi: 10.1145/2588555.2610507. URL https://doi.org/10.1145/2588555.2610507.

[2] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki. Scaling up concurrent main-memory column-store scans: Towards adaptive numa-aware data and task placement. *Proc. VLDB Endow.*, 8 (12):1442–1453, Aug. 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824043. URL https://doi.org/10.14778/2824032.2824043.