

Self-Tuning Query Scheduling for Analytical Workloads

Benjamin Wagner
Technische Universität München
benjamin.wagner@tum.de

André Kohn
Technische Universität München
andre.kohn@tum.de

Thomas Neumann
Technische Universität München
thomas.neumann@tum.de

ABSTRACT

Most database systems delegate scheduling decisions to the operating system. While such an approach simplifies the overall database design, it also entails problems. Adaptive resource allocation becomes hard in the face of concurrent queries. Furthermore, incorporating domain knowledge to improve query scheduling is difficult. To mitigate these problems, many modern systems employ forms of *task-based* parallelism. The execution of a single query is broken up into small, independent chunks of work (tasks). Now, fine-grained scheduling decisions based on these tasks are the responsibility of the database system. Despite being commonplace, little work has focused on the opportunities arising from this execution model.

In this paper, we show how task-based scheduling in database systems opens up new areas for optimization. We present a novel lock-free, self-tuning stride scheduler that optimizes query latencies for analytical workloads. By adaptively managing query priorities and task granularity, we provide high scheduling elasticity. By incorporating domain knowledge into the scheduling decisions, our system is able to cope with workloads that other systems struggle with. Even at high load, we retain near optimal latencies for short running queries. Compared to traditional database systems, our design often improves tail latencies by more than 10x.

CCS CONCEPTS

• **Information systems** → **Online analytical processing engines**; *Autonomous database administration*.

KEYWORDS

Database Systems; Query Scheduling; Parallelism; Self-Tuning

ACM Reference Format:

Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, June 20–25, 2021, Virtual Event, China. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3448016.3457260>

1 INTRODUCTION

Analytical database systems have to face complex workloads. These are particularly challenging when heterogeneous requests arrive in parallel and the system operates under high load. In these cases, many systems struggle to retain competitive query performance. For the user, this has severe repercussions. The system becomes less responsive, taking longer to provide the desired insights. Even worse, query performance becomes unpredictable. When running

SIGMOD '21, June 20–25, 2021, Virtual Event, China

© 2021 Association for Computing Machinery.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*, June 20–25, 2021, Virtual Event, China, <https://doi.org/10.1145/3448016.3457260>.

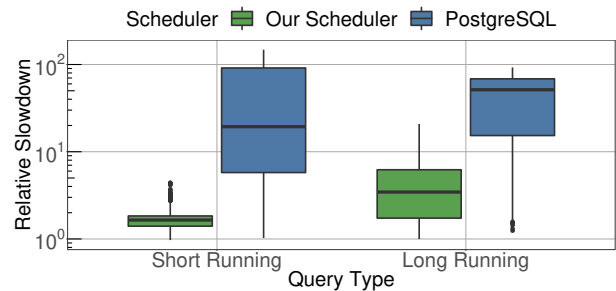


Figure 1: Query latencies at high load. The workload consists of 75% short and 25% long running queries. The systems are run at 95% of their maximum sustainable load for 20 minutes. The relative slowdown is measured with respect to the isolated query latency within each system.

the same request at different times, the user should observe similar latencies. This is not always possible. High system load will inevitably impact query durations. Nevertheless, performance should degrade as gracefully as possible. Our experiments show that this is not the case: in traditional systems, latencies of the same query often vary by more than 50x. In this paper, we show how fine-grained control over CPU resources solves these problems. This is exemplified in Figure 1. At high load, our novel scheduler provides far better overall performance and significantly shorter tail latencies than PostgreSQL. This is especially true for short running requests, which benefit most from our contributions.

Database systems like PostgreSQL transfer scheduling responsibilities to the operating system (OS) [20]. They create individual OS threads or processes for every new connection and execute the incoming requests in an isolated fashion [5]. Additional threads may be spawned to employ intra-query parallelism. The maximum number of concurrent OS threads is often bounded in order to not oversubscribe the system. Other examples for such systems include IBM DB2 [6] and Microsoft SQL Server [14].

Many modern database systems deviate from this classic approach. Their parallelization scheme closely resembles task-based parallelism. Each query is split into a set of independent tasks, which can be executed in parallel by different OS threads. This inverts the scheduling responsibilities. In these systems, the OS threads are not statically bound to a query anymore. Rather, they can dynamically execute tasks from different queries. This necessitates a scheduling policy within the database, dictating which task to pick next. In some systems, this user-space scheduling policy works in symbiosis with the OS scheduler. An example is SAP HANA, which employs an adaptive number of OS threads that pick tasks from different queries [22, 23].

Other systems like HyPer and Umbra relieve the OS from almost all scheduling decisions [8, 16]. On startup, they spawn as many OS threads as there are CPU cores. Since these threads do not block

when executing tasks, this maximizes performance. The system is not oversubscribed and context switches are kept to a minimum. The task-based parallelization scheme is realized through so-called morsels. A morsel represents a fixed set of tuples in the context of an executable pipeline and is the smallest unit of work during query execution [11]. For example, a morsel might read ten thousand tuples from a base relation, apply a filter predicate and insert the remaining tuples into a hash-table. Since morsels of the same pipeline can be executed in parallel, this enables both intra- and inter-query parallelism. At the same time, it enforces that all tasks picked by the OS threads are independent. It does not matter whether other threads execute tasks from the same pipeline concurrently.

General-purpose task schedulers like the one of Intel TBB are mainly focused on maximizing throughput [7]. Meanwhile, the database system should focus on objectives like fairness or query responsiveness. When implementing its own task scheduler, the database system can make smart scheduling decisions by exploiting domain knowledge. For example, the system might execute tasks from short running requests preferentially in order to retain low latencies at high load. Additionally, the granularity of spawned tasks can be changed adaptively to improve execution characteristics.

In this paper, we present a novel lock-free, self-tuning stride scheduler that we built into the task-based database system Umbra [16] in order to seize these opportunities. Our three primary contributions are the following: in Section 2 we present our scheduler design. Section 3 then focuses on morsel-driven database systems. We show how a morsel-based task structure can be used to make scheduling more robust and predictable. Finally, Section 4 adds self-tuning capabilities to our scheduler. By simulating its own execution based on the tracked workload, the scheduler transparently optimizes its hyperparameters to improve latency characteristics.

Our contributions enable predictable and high query performance for database systems with a task-based parallelization scheme, even as load increases. For short running queries, we improve the mean slowdown over traditional systems by more than 4.5x, with an even stronger effect on tail latencies. Resource intensive requests do not slow down lightweight queries anymore.

2 SCALABLE TASK SCHEDULING

In database systems with task-based parallelism, the task picked by an OS thread is dictated by a user-space scheduling policy. This policy has to be highly scalable in the face of parallel hardware and provide fine-grained control over CPU resources. In this section, we present a novel, lock-free implementation of stride scheduling. It provides the strong theoretical guarantees of classic stride scheduling, while minimizing the communication cost between different threads, since almost all scheduling decisions can be made on a thread-local basis. This section focuses on the implementation details of our scheduler. Section 3 then shows how the scheduler can be made robust in the context of morsel-driven database systems.

2.1 Background

Stride scheduling [30] is a classic priority scheduling algorithm. Assume we are given tasks t_1, t_2, \dots, t_n with corresponding integer priorities p_1, p_2, \dots, p_n . Each task gets assigned a stride $S_i = (p_i)^{-1}$. If all tasks arrive at the same time, stride scheduling becomes simple. Every task is mapped to a pass P_i , which is initially set to zero.

The scheduling decisions now proceed in the same way: the task with the minimal pass gets picked and executes one time slice of work. The pass then gets updated to $P_i + S_i$. For example, if t_1 has a priority of 5 and t_2 has a priority of 10, the stride of t_1 is twice as large as the stride of t_2 . This implies that t_2 gets scheduled twice as often as t_1 . Thus, stride scheduling provides proportional-share resource allocation. Task t_i obtains $p_i / \sum_{k=1}^n p_k$ of the computational resources. In this context, the pass of a task serves as an abstracted notion of execution time that takes the priority of the task into account. Stride scheduling is fair if all tasks have the same priority.

Stride scheduling requires some minor modifications to work with a set of dynamically changing tasks. If a task is added to the scheduler at an arbitrary point in time it requires an initial pass value. For this, the scheduler maintains a global stride $S_G = (\sum_{k=1}^n p_k)^{-1}$ as well as a global pass P_G . After every scheduled time slice, the global pass gets incremented by the global stride. The global pass can now be used to compute the initial pass value for a new task. Intuitively, the global pass represents the timestamp of the scheduler. If a task has a pass lower than the global one, it has not yet received the resources it is entitled to. Tasks that have a larger pass than the global one have received too many resources.

Stride scheduling can easily be extended to work in a non-preemptive setting. If a task t_i consumes a fraction f of its allocated time slice, the pass gets updated to $P_i + fS_i$. In the same way the global pass gets set to $P_G + fS_G$. Here, f may be larger than one.

2.2 Scheduling in Umbra

We now provide an overview of the scheduling concepts in Umbra, the OLAP system into which we built our scheduler. Umbra is an ACID-compliant database built to provide high performance beyond main-memory [16]. Umbra uses code generation to efficiently evaluate queries [15]. Figure 2 visualizes Umbra’s task structure.

Every executable pipeline is turned into a so-called task set. A task set contains an arbitrary number of independent tasks which can be executed in parallel by different OS threads. Task sets of the same query may be subject to ordering constraints. Within the figure, the blue pipeline of the left algebra tree has to be finished before the orange one can start. This is because the build side of the join needs to be materialized before probing can commence. This is enforced by putting all task sets of a query into a so-called resource group, which stores the task sets in an ordered fashion. A task set within a resource group may only be started once all previous task sets have been finished. As an added benefit, resource groups allow us to track resource consumption at query granularity. This will later be a crucial detail when prioritizing short running queries.

Resource groups and task sets allow us to reason about query execution at a high level of abstraction. We now want to drill down into the structure of the tasks being executed on OS threads. Umbra uses morsel-driven parallelism to handle intra- and inter-query parallelism [11]. Morsels are the smallest units of work during query execution. Each morsel processes a set of tuples within an executable pipeline. Different morsels of the same pipeline can be executed in parallel by different OS threads. Morsel-driven database systems like HyPer implement a 1:1 mapping between scheduler tasks and executable morsels [11]. Our design breaks up this rigid dependency. Every task may consist of an arbitrary number of morsels. Within Umbra, tasks are not created statically during

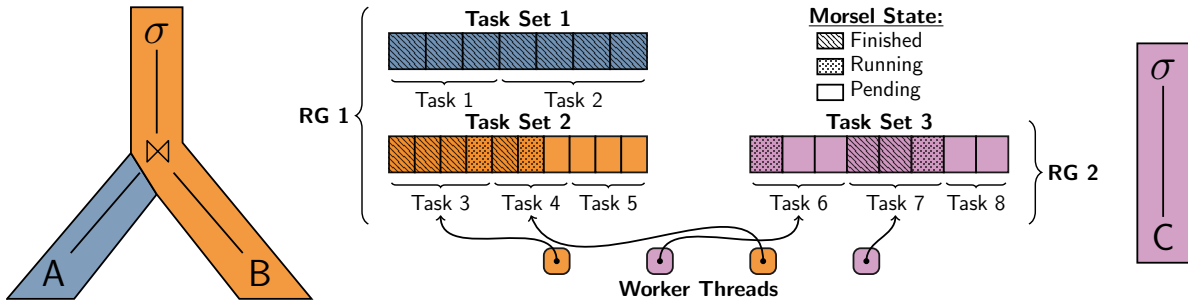


Figure 2: In Umbra, each pipeline is mapped to a task set. Every task set contains several tasks which are made up of morsels. Each worker thread is pinned to one task. Task sets of the same query are wrapped into a resource group (RG).

query compilation. Rather, tasks and morsels are carved out at runtime. This allows us to dynamically change the task structure based on runtime observations. To keep presentation simple, this was omitted in Figure 2. A more detailed discussion will follow in Section 3, which uses the flexibility of tasks to improve scheduling.

Tasks are executed by OS threads. On startup, Umbra creates as many OS threads as there are CPU cores. We also call these threads worker threads. The worker threads are only responsible for executing scheduler tasks. This design minimizes context switches and prevents oversubscription. Since we study analytical workloads that fit in main-memory, worker threads do not block. The scheduling logic of a single worker is simple. It picks one of the active task sets, carves out a task and then executes it. Since tasks from the same task set can be executed in parallel, the workers do not have to be aware of concurrent scheduling decisions.

2.3 Thread-Local Scheduling

Stride scheduling provides strong deterministic scheduling guarantees. However, this alone is insufficient on modern hardware, since excessive synchronization between threads becomes detrimental with a rising number of CPU cores. The database should spend most of its time on query processing, which implies that the scheduling overhead must be negligible. This section therefore presents a novel, task-based implementation of stride scheduling, which scales well with an increasing number of worker threads. Our design can perform all scheduling decisions on a thread-local basis. Workers are only notified of changes to the active task sets. This minimizes the synchronization overhead between threads and allows for high scheduling performance.

Compared to classical stride scheduling, our implementation maintains an upper bound on the number of active resource groups. In Umbra, this is set to 128, but our design allows for an arbitrary upper limit. This limit is only reached when the system is severely oversubscribed. In these cases, we tolerate graceful latency degradation in order to bound the memory consumption of the system. The resource groups of newly arriving queries are put into a preceding wait queue until a free slot becomes available within the scheduler.

We now discuss the core building blocks of our scheduler design. While we focus on stride scheduling, our approach can be easily modified to work with other scheduling algorithms. This only requires altering the thread-local scheduling logic. The remaining infrastructure can stay in place. For example, we implemented non-deterministic lottery scheduling [29] besides stride scheduling in less than 100 lines of C++ code.

Thread-Local Decisions. Our scheduler maintains a global array of slots which are bound to active resource groups. Each slot stores a pointer to the currently active task set of the resource group. When a task set of a resource group finishes and a new one becomes active, it is put into the same slot again. This simplifies scheduling considerably, since priorities are tied to resource groups and not task sets. If task sets of the same resource group could be put into different slots, this would require more complicated bookkeeping.

Other than that, all scheduling metadata is stored in a thread-local fashion. This includes a bitmask tracking the currently active slots within the global resource group array, as well as a mapping from slots to priorities and pass values. Additionally, each worker thread stores its own global pass. This is shown in Figure 3.

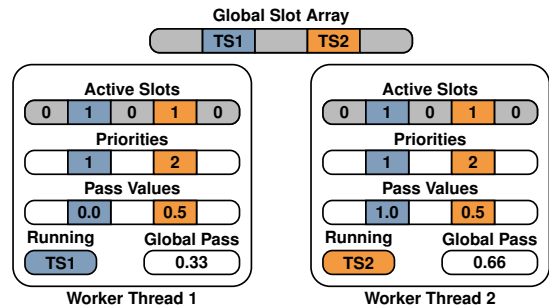


Figure 3: Scheduling with two resource groups (blue and orange). Only the active task sets (TS) are stored globally.

If the global slots and the local activity mask of a worker are in sync, scheduling is simple. The worker picks the active slot with minimal pass value. Afterwards, it performs an atomic read on the global slot array to obtain a pointer to the current task set. It can then pick a task, track the execution time and update its thread-local pass values accordingly. This protocol is very lightweight. All of the core scheduling decisions can be performed independent of the other threads. Most notably, a worker picking a task does not have to know if other threads are working on the same task set. Furthermore, the global slot array is only being written to when a new task set becomes active. These writes are relatively infrequent, which does not lead to excessive cache invalidations.

Changing the Active Task Sets. The cost of picking a task for execution is minimized by keeping almost all information local to each worker thread. Nevertheless, some communication is required to

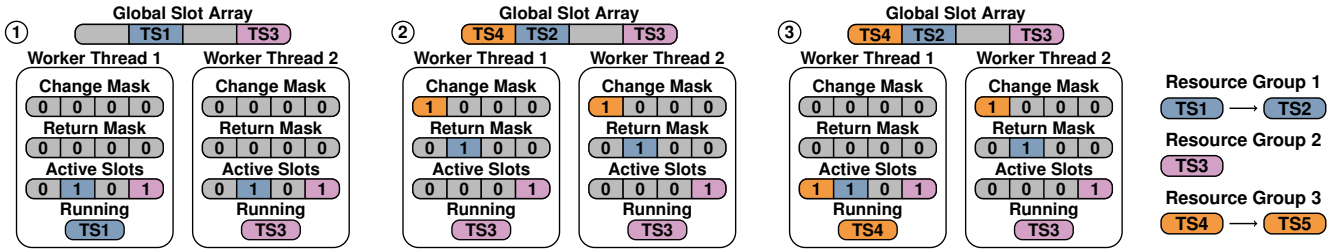


Figure 4: The change and return bitmasks within each worker are used to incorporate changes to the active task sets (TS).

keep the different threads in sync. A worker has to be able to detect the following three events:

- (1) A task set in one of the global slots was finished. The worker has to disable the corresponding local slot. It cannot be guaranteed that the slot will contain a new task set soon.
- (2) The initial task set of a new resource group was assigned to one of the global slots. The worker needs to pick an initial pass value and priority for the resource group. It also has to set the slot to active in its local activity bitmask.
- (3) A new task set of an active resource group was inserted into the corresponding global slot. The worker has to reactivate its local slot and set an initial pass value. Since priorities are bound to resource groups, the worker can retain the slot priority.

We handle event (1) optimistically. When a task set runs out of work, its global slot gets marked as inactive by tagging the contained pointer. Thus, we do not have to notify the worker threads when a task set was finished. Instead, the slot remains active in each local activity mask. When a slot gets chosen for execution, the worker has to read the pointer in the global array anyways. It only disables the slot locally if the value indicates that it is not valid anymore.

For events (2) and (3), such an approach is undesirable since it requires periodic checks of all inactive slots. This makes it hard to guarantee high responsiveness. Instead, each worker maintains two atomic bitmasks for updates to the active task sets. When a worker inserts a new task set into one of the global slots, these bitmasks get updated accordingly for all workers. On event (2) we update the first bitmask. On event (3) we update the second one. From now on, we will call the first bitmask the *change mask* and the second one the *return mask*. When talking about both masks, we use the term *update masks*.

Pushing updates into these bitmasks is simple. Assume we want to indicate that the k 'th global slot has received the initial task set of a new resource group. For this we set the k 'th bit in the change mask of each worker thread to one. We can set the target bit without changing the state of any other slot by performing an atomic fetch or of the change bitmasks and the value 2^{k-1} . This way, multiple threads can push state updates into the local worker state without coarse grained synchronization. Similarly, each worker can easily pull updates into its local scheduling state. It first performs an atomic exchange of its update masks with zero. This resets the change and return mask to reflect that the worker was notified of all outstanding global changes. The original mask values can now be used to incorporate the state updates. The worker has to extract the indices of set bits in the old mask values. For the slots at these indices, the local scheduling state gets adjusted accordingly. This can be done efficiently by repeatedly counting the leading zeros and

then shifting the value by this number. This is supported through `clz` and `shl` instructions on all modern hardware platforms.

Before picking a task for execution, each worker synchronizes with its local update masks to encompass new task sets in its scheduling decision. If there are no outstanding changes, this update is cheap. Cache invalidations are avoided since there were no atomic writes to the local update masks since the last read. Furthermore, no changes to the local scheduling state are required. An example is shown in Figure 4. In the second image, two new tasks sets have been inserted into global slots. These updates have already been pushed into the update masks of both workers. Note that we use the return mask to reflect that TS2 comes from an existing resource group, while we update the change mask to notify the workers of TS4. Since the workers are currently executing a task of TS3, they were not yet able to pull the updates into their local scheduling state. In the third image, the first worker has now synchronized itself with its update masks. Worker two is still pinned to its task in TS3 and has not yet incorporated the updates. Thus, two workers do not have to be in sync with respect to their active task sets.

Despite using atomic bitmasks, our scheduler can cope with an arbitrary number of slots that is not bound by the width of atomic instructions on the target architecture. In Umbra we maintain a slot limit of 128. Each update mask is made up of two atomic eight byte integers. This works since we do not require a complete operation on the bitmask to be atomic. It is sufficient if individual steps in an operation satisfy atomicity constraints. This way it is still guaranteed that no update to the bitmask is lost.

Task Set Finalization. Once a task set is done we have to activate the next task set in the resource group (if one exists). This may only happen once all tasks of the original task set have been fully processed. We call this step task set finalization. To increase flexibility, we also allow task sets to run additional finalization steps once they are done. Examples are the shuffling of partitions during sorting, or the merging of partial aggregates during grouping.

A worker thread is notified if it tries to pick a task from a task set with no remaining work. At first glance, it might appear as if we could start finalization as soon as this happens. However, this approach is insufficient since we have to guarantee that all other threads finished their outstanding work. Other workers might still be pinned to the task set, executing the last remaining tasks. In this case, we would start finalization too soon. Our scheduler avoids this scenario by introducing a lightweight finalization phase for each task set. When a worker picks a slot for execution, it publishes this decision in a global state array. This happens *before* the atomic read of the global slot. The finalization phase of a task set is started once

a worker notices that the task set is exhausted. The first worker to notice this coordinates the finalization phase.

The coordinating worker has to ensure that the last thread *finishing* work on the task set invokes the finalization logic. First, the coordinator marks the global slot as invalid by tagging the pointer to the task set. This way, any worker that chooses the slot from now on will disable it instead of trying to pick a task. Afterwards, the coordinating worker iterates through the global state array to find the workers that are still pinned to the task set. For each of them, it exchanges the slot information in the state array with a dedicated finalization marker. All worker threads that are marked this way have to explicitly deregister at the task set once they finish their current task. This is done through an atomic finalization counter in each task set. The coordinating worker increments the finalization counter by the number of workers for which it successfully set the finalization marker. When a worker finishes executing a task, the worker checks if the task's global state field contains the finalization marker. If this is the case, it decrements the finalization counter of the owning task set by one. Since this might happen before the coordinating worker finished iterating through the state array, the finalization counter can temporarily become negative. The worker that sets the counter to zero invokes the finalization logic. This enforces that it was the last thread working on the task set. If the worker thread that performs finalization does not find a successive task set within the resource group, it attempts to obtain a new resource group from the global wait queue.

This finalization phase is very lightweight. The overhead during each scheduling decision is limited to updating the global state array. As long as a worker does not pick a task set undergoing finalization, its writes to the global state array are uncontended. Furthermore, we ensure that finalization affects as few workers as possible. Only those threads that are pinned to a task set when it runs out of work take part in the finalization phase. If there are many active task sets, this will only be a fraction of the total workers.

Coping With High Load. As system load increases, letting all workers pick arbitrary tasks becomes suboptimal. Since pipelines do not scale perfectly with respect to the number of concurrent workers, it is inefficient to have multiple workers pick tasks from the same task set. We mitigate this by restricting which threads may work on certain task sets. As soon as half of the slots are occupied, we linearly decrease the number of workers into which we push task set updates. Once all slots are full, each task set is only pushed into a single worker, eliminating all contention within pipelines. In addition, this also reduces the overhead during task set updates.

3 ROBUST MORSEL SCHEDULING

Section 2 presented a lock-free stride scheduler that we built into Umbra. The scheduler is designed for database systems with a task-based parallelization scheme. This section presents further optimizations for systems utilizing a morsel-based task structure.

In a first step, we show how morsel-driven parallelism can be made robust. Classic morsel-driven parallelism entails an extreme variance in task granularity, which leads to undesirable scheduling artifacts. The scheduling overhead becomes unpredictable and workers can be blocked for a long time. Section 3.1 presents a solution to this problem. By normalizing the execution time of tasks, we enable predictable scheduling overhead and high responsiveness.

In a second step, we exploit database domain knowledge to optimize latency characteristics for mixed analytical workloads. We use this term to refer to workloads consisting of analytical queries with a high variance in query duration. For example, our evaluation in Section 5 samples from TPC-H queries at scale factor 3 and 30. Through query priorities, our stride scheduler provides fine-grained control over relative resource consumption. Rather than having the user assign priorities to each request, the system should be able to choose smart query priorities without any user input. Section 3.2 shows how adaptively decaying priorities enable the transparent prioritization of short running requests. While our design profits from the novel task structure introduced in Section 3.1, it can also be applied to other systems with task-based parallelism.

3.1 Adaptive Morsel Execution

In HyPer, there is a one to one correspondence between morsels and scheduler tasks. Every time a worker picks a task, it executes one morsel of work. Leis et al. recommend fixed morsel sizes, providing a trade-off between scheduling overhead and responsiveness [11].

However, this approach suffers from high variance with respect to the morsel durations. The code generated for different pipelines can vary considerably in complexity. A pipeline consisting of a simple selection and hash table insert will spend less time on each tuple than a pipeline performing complex string matching and several hash table probes. For the scheduler, this implies that the granularity of different tasks varies considerably. While stride scheduling can in principle cope with this scenario, it is still undesirable in practice. A small fixed morsel size leads to extremely short task durations, resulting in high scheduling overhead. If the fixed morsel size is too large on the other hand, workers are blocked for a long time, making the system less responsive. This is presented in Figure 5a. Here, we compare the execution traces of TPC-H queries 13 and 21 at scale factor one. All morsels have a fixed size of 60 thousand tuples. However, morsel durations differ by more than 30x.

As we have seen, the classic approach to morsel scheduling is static in two dimensions: (1) it relies on fixed morsel sizes and (2) it utilizes a static mapping between morsels and scheduler tasks. To overcome the limitations of this design, we introduce a novel, adaptive framework for executing tasks. The scheduler defines a target duration t_{max} . When a worker picks a task, the task tries to schedule morsels that exhaust this target duration as precisely as possible. This approach replaces the static nature of the previous design with a dynamic runtime policy. A task may (1) use adaptive morsel sizes depending on the current pipeline throughput and (2) execute multiple morsels. This way, it becomes possible to execute several morsels without increasing scheduling overhead. After all, the scheduler is not aware of the underlying task structure. Through this, the scheduling pressure becomes predictable. In Umbra, we empirically set t_{max} to 2ms to balance scheduling overhead and responsiveness. Since we measured that each scheduling decision takes less than one microsecond, this caps the overhead at 0.05%.

Depending on the execution progress of the pipeline, different strategies are employed in order to exhaust the target duration. This is achieved by transforming each pipeline into a state machine. The current pipeline state dictates how morsels are chosen. Our design relies on the dynamic nature of morsels and tasks outlined in Section 2.2. Since morsels are carved out from the set of tuples

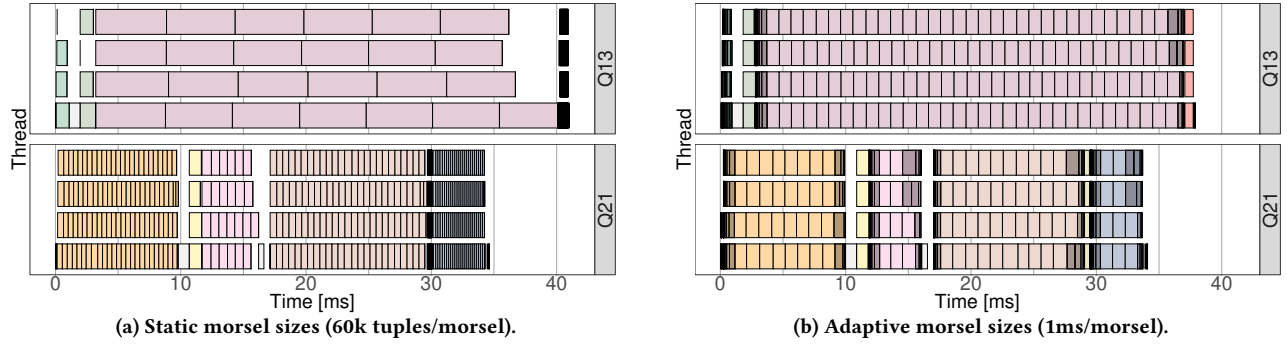


Figure 5: Adaptive morsel sizes lead to predictable execution profiles. The morsels are colored at pipeline granularity.

at runtime, it is possible to dynamically fill a scheduler task with multiple morsels of different sizes.

Default State. When a pipeline is in its *default state*, it tries to pick a single morsel that completely exhausts the target duration t_{max} . Such an approach is feasible as soon as we have a reliable throughput estimate T , measured in tuples per second. In this case, the pipeline picks a morsel of size $T \cdot t_{max}$ tuples. Since the execution time is roughly proportional to the number of tuples being processed, this size ensures that the morsel takes approximately t_{max} to execute. Once the morsel is done, we know its real execution time t . The measured throughput $\hat{T} = (T \cdot t_{max})/t$ is then incorporated into the estimate in order to maintain fresh throughput statistics. Given the old throughput estimate T as well as $\alpha \in [0, 1]$, we set the new throughput estimate $T' = \alpha \hat{T} + (1 - \alpha) \cdot T$. Thus, the update of the throughput estimate is based solely on the measured throughput of the morsel. In Umbra, we choose $\alpha = 0.8$, since we want to give a lot of weight to the most recent measurements.

Startup State. This approach works well once we have an initial throughput estimate. However, obtaining such an estimate is non-trivial. Statically picking an initial size is undesirable. Similarly to picking all morsel sizes in a static fashion, this can result in extremely long running morsels, blocking the progress of other queries. As a solution, we introduce an initial *startup state* for each pipeline. This state is responsible for providing a first throughput estimate. When a worker thread picks a task of a pipeline in this state, it executes exponentially growing morsels until the target duration is exhausted. We first run a morsel containing C_0 tuples, for which we measure the execution time t_0 . In Umbra, we empirically set $C_0 = 16$. C_0 has to be chosen sufficiently small to ensure that $t_0 \leq t_{max}$. All successive morsels are scheduled in the same way. Assume we already executed i morsels with sizes C_0, \dots, C_{i-1} . We now want to schedule a morsel of size $C_i = 2 \cdot C_{i-1}$. We can expect this morsel to take roughly $2 \cdot t_{i-1}$ to execute. Since we should not exceed the target duration t_{max} , the morsel is only scheduled if $2t_{i-1} \leq t_{max} - \sum_{k=0}^{i-1} t_k$. Once we cannot execute additional morsels, the pipeline switches from its startup state to the default state. The measured throughput of the final startup morsel is used as initial throughput estimate.

Optimizations. We implement two additional enhancements to improve execution characteristics. In addition to the two pipeline states presented above, we introduce a *shutdown state*. The remaining execution time of a pipeline can be estimated through the number of remaining tuples and the current throughput estimate. If

our scheduler has W worker threads, the shutdown state is entered once the predicted remaining time of the pipeline drops below $W \cdot t_{max}$. In this case, we aim for a “photo finish” of the worker threads to avoid stragglers. Given the estimated remaining time t and a minimum morsel duration t_{min} , we schedule morsels of duration $\max(\frac{t}{W}, t_{min})$ until we exhaust the target duration t_{max} , or the pipeline runs out of work. The second optimization concerns tasks that do not support adaptive morsel sizes. These still have to be treated efficiently, since some tasks may just not be suited for high adaptivity. If we detect at runtime that a task executes morsels that only consume a fraction of the target duration, we allow it to execute further morsels until t_{max} is exhausted. This also enables the piecewise and efficient adoption of adaptive morsel sizes.

Evaluation. Execution traces utilizing our adaptive morsel framework are shown in Figure 5b. We use the same queries as in Figure 5a. For each pipeline, the startup and shutdown phase are darkened out. The image also shows the nested morsels of tasks in the startup and shutdown phase. Since they are transparent to the scheduler, they can be short running without causing additional overhead. We can nicely see the exponentially growing nested morsels at the beginning of each pipeline. We can also observe that the shutdown phase leads to a precise photo finish, reducing the latency of query 13 compared to static morsel sizes.

Overall, our design ensures that query processing produces predictable execution traces. For this, we utilize domain knowledge of the database to alter the task structure within the scheduler. This is fundamentally different from general purpose schedulers that have to cope with arbitrary workloads. By setting the target duration t_{max} , the database can balance responsiveness and scheduling overhead. Through the flexible nature of morsel-driven parallelism, scheduling in database systems becomes robust.

3.2 Adaptive Query Priorities

Stride scheduling paves the way for adaptive workload management in modern database systems. By changing the priority of different queries, we obtain fine-grained control over their relative resource consumption. For example, a query of a data analyst running business critical investigations could be granted a higher priority than requests used for non-critical accounting.

However, using these capabilities to their full extent is not easy. In general, a user should not have to deal with the intricacies of workload management. Instead, the system should allocate resources in a way that yields desirable latency characteristics without

user input. In this section, we show how this can be achieved for mixed analytical workloads. We utilize adaptive query priorities to transparently treat short running requests in a preferential fashion. Our approach requires no user input and assigns priorities solely based on runtime characteristics of the given queries.

We first want to discuss how we can quantify “desirable” latency characteristics. Since the user provides no priority input, we assume that all queries are of equal importance. Under this assumption, we propose that query scheduling should be guided by two fundamental principles.

(1) Query latencies should remain predictable under load.

If the system receives two queries at the same time, the shorter query should finish first. We now formalize this requirement. Given a query q , we define the base latency $L_B(q)$ as the latency of q when executed in isolation. A workload W consists of tuples (q_i, t_i) , where q_i is a query with arrival time t_i . We can interpret a scheduling policy P as a mapping from a workload W to a latency function P_W . Here, P_W encodes the latency of each query in W when utilizing P as a scheduling policy. Thus, given a workload W and $(q_1, t), (q_2, t) \in W$, it must hold that $L_B(q_1) < L_B(q_2) \Rightarrow P_W(q_1) < P_W(q_2)$.

(2) Query latencies should be kept as low as possible. Let \mathcal{P} denote the set of scheduling policies that satisfy principle (1). Given a workload W , we define a cost function

$$f_W : \mathcal{P} \rightarrow \mathbb{R}^+ : P \mapsto \sum_{(q,t) \in W} \frac{P_W(q)}{L_B(q)}. \quad (1)$$

The database scheduler should utilize a scheduling policy \hat{P} for which $f_W(\hat{P})$ is close to $\min_{P \in \mathcal{P}} f_W(P)$. We thus want to minimize the mean relative slowdown of queries. However, other cost functions could be considered as well.

If the database respects both of the above principles, it can achieve predictable and high query performance. While fair scheduling does respect invariant (1), it is not optimal with respect to principle (2). Often, short running queries can be prioritized without noticeably altering the latency of other, long running requests. Assume we are executing two types of queries. The short running ones make up 90% of the workload and take 10ms to execute. Meanwhile, the long running requests take 1s. Even if we treat all short requests preferentially, this amounts to less than 10% of the overall workload being prioritized. As a result, the long running requests are not slowed down significantly. At the same time, we improve the latency characteristics for nine out of ten queries. Compared to fair scheduling, this leads to lower cost in Equation 1.

We propose adaptive query priorities in order to transparently prioritize short running requests. Similar to scheduling with multi-level feedback queues [9], the priority of a query depends on the amount of CPU resources it received so far. Remember that we wrap each query into a resource group which is then passed to the worker threads. When a resource group is registered at one of the workers, it is assigned an initial priority p_0 . The more time the worker spends on the resource group, the lower its priority becomes. Specifically, we update a resource group’s priority after it has received a fixed quantum t of CPU time from the worker thread. Given its old priority p_i , the new priority p_{i+1} is calculated as

$$p_{i+1} = \begin{cases} p_i & , i < d_{start} \\ \max(p_{min}, \lambda p_i) & , i \geq d_{start}. \end{cases} \quad (2)$$

The priority update is governed by three hyperparameters. The parameter d_{start} defines how soon a resource group’s priority starts to decay. The speed of this decay is regulated by $\lambda \in [0, 1]$. Finally, priorities must never drop below $p_{min} > 0$. This ensures that queries never starve. These parameters should be chosen such that the cost function (1) is minimized. Good parameter values are workload dependent, we present a self-tuning optimizer in Section 4. This design benefits from our previous contributions in two main ways. (1) Through our lock-free scheduler design, all priority updates happen in a thread-local fashion. (2) By setting the update quantum t to the target task duration t_{max} introduced in Section 3.1, priority decay usually happens after every scheduled task. This way, priority updates are highly regular across resource groups.

Compared to fair scheduling, our approach leads to more skewed relative performance differences. This however is necessary in order to achieve significantly better performance for short running requests. Nevertheless, we still guarantee invariant (1). The priority decay of two queries arriving at the same time is identical. Thus, they receive comparable resources and the shorter one finishes first.

Custom Priorities. Adaptive query priorities provide desirable workload characteristics when there is no additional information on the importance of queries. However, additional information may be available in some scenarios. There are two simple ways to extend adaptive query priorities to incorporate this knowledge.

(1) It is possible to use our adaptive priority design by default, while still allowing the user to specify a static priority for a subset of queries. For example, especially important queries could have the static non-decayed priority p_0 . In this case, they are always treated in the same way as a new query which has just arrived.

(2) Priorities can also be attached to users. The user priority then influences the decay parameters for all queries of this user. Both the initial priority p_0 and the minimum priority p_{min} are scaled by the user priority. This way, different users are prioritized differently. However, each user still benefits from adaptive query priorities.

4 SELF-TUNING SCHEDULER

Section 3.2 showed how decaying query priorities can be used to improve the latencies of short running requests. In this section, we extend our scheduler by making it self-tuning. It dynamically changes the decay parameters in order to maximize query performance. This enables the database to make smart scheduling decisions for arbitrary analytical workloads, without relying on any user input. This is achieved as follows: our scheduler tracks the workload over a fixed interval. By simulating its own execution, it then finds parameter values that optimize the latencies of the tracked queries. This procedure is repeated periodically in order to adapt to workload changes. We now study this process in detail.

Motivation. For the previous parameters introduced in the paper, finding sensible choices that work well across workloads is simple. An example is the task target duration t_{max} . Here, we chose 2ms to balance responsiveness and scheduling overhead. This choice is independent of the actual workload. Sadly, this is not possible for the

decay parameters. Given a mixed workload consisting of queries taking 10ms and 100ms, it is possible to decay aggressively after a few milliseconds in order to prioritize the short requests. If we choose the same parameters for queries taking 1s and 10s however, all requests will quickly reach the minimum priority p_{min} . The short running queries are not being treated preferentially anymore. In this case, we want to significantly increase the decay onset d_{start} . The burden of picking scheduling parameters should not be offloaded to the user. Finding smart parameters for a static workload is hard already, and workloads may fluctuate heavily over time.

Optimization Problem. Using the notation of Section 3.2, finding good decay parameters can be formulated as a constrained optimization problem. In order to maintain progress guarantees, we fix the initial priority $p_0 = 10^4$ and enforce a lower priority bound $p_{min} = 100$. We define the parameter space

$$S = \{(\lambda, d_{start}) \mid \lambda \in [0, 1], d_{start} \geq 0\}.$$

For some $s \in S$, let $P(s)$ denote the stride scheduling policy with the priority decay parameters in s . Given a workload W , we try to find parameters that minimize the mean relative slowdown of queries by solving

$$\operatorname{argmin}_{s \in S} \sum_{(q,t) \in W} \frac{(P(s))_W(q)}{L_B(q)}. \quad (3)$$

Tuning Process. For a database system, knowing the workload W in advance is usually not feasible. But if the workload is well-behaved over a short period of time, this can be mitigated effectively. The scheduler can track the queries being executed for a fixed duration t_t . It can then use the tracked workload to solve the optimization problem in Equation 3. If the workload does not change drastically after tracking has finished, this will still yield reliable decay parameters. We have to ensure that the system remains responsive with respect to workload fluctuations. For this, we define a refresh duration $t_r \gg t_t$. For all $k \in \mathbb{N}$, a tracking run is started after $k \cdot t_r$. Each tracking run is used to update the decay parameters.

We require tracking and optimization to be very lightweight. The system should only spend a fraction of its overall resources on finding optimal scheduling parameters in order to not impact query latencies. This becomes possible through our thread-local scheduler design. Since the workload is symmetric across worker threads, it is sufficient to only track execution on a single worker. Optimization can then be performed by considering the scheduling problem on a single thread with the reduced workload. This minimizes both tracking and optimization cost, especially on modern, highly parallel hardware. Measurements of the tuning overhead under increasing core counts are shown in Section 5.3.

A complete iteration of the tuning process is depicted in Figure 6. For some $k \in \mathbb{N}$, the iteration is started at time $k \cdot t_r$. We first attach a lightweight tracker to one of the worker threads. This does not impact the regular scheduling flow outlined in Section 2. The tracker only logs the execution time spent on each of the active resource groups. Once the interval t_t is exhausted, the tracking worker will not execute any more tasks. Instead, it uses the tracked workload to perform parameter optimization. This does not impact the other workers, which can still execute active tasks. Once optimization is done, the new parameters are pushed into all workers.

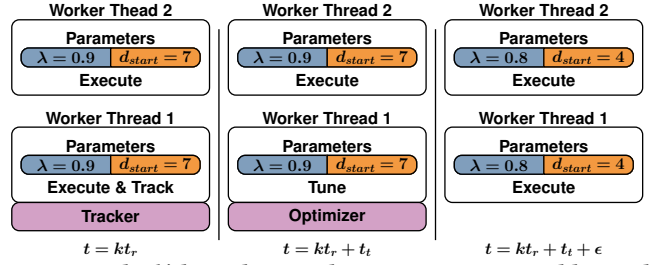


Figure 6: The k 'th tracking and optimization run. Additional work is restricted to a single thread.

The scheduler can now operate without any tuning overhead until a new tracking run is started at time $(k + 1) \cdot t_r$.

Self-Simulation. For parameter optimization, we have to be able to evaluate the cost function in Equation 3. Given scheduling parameters $s \in S$ and the tracked workload W , the system simulates the execution of W with the scheduling policy $P(s)$. This yields simulated query latencies $(P(s))_W$, which can be used to efficiently evaluate the cost function. This simulation can be kept very lightweight. Adaptive morsel execution as outlined in Section 3.1 leads to highly regular execution traces. The simulator can thus keep a discretized notion of time, performing a simple loop over equally spaced scheduling decisions in the tracking interval. During each iteration, it picks an active query based on the simulated policy $P(s)$. This also leads to predictable simulation performance across different parameter sets and workloads.

Optimizer. Due to the discrete nature of scheduling, the optimization problem presented in Equation 3 is non-continuous. Thus, classic approaches to numerical optimization cannot be applied easily. Instead, we perform a directional search commonly used in derivative free optimization [1]. Given a workload W after the k 'th tracking run, we pick a set of starting values $s \subseteq S$. If $k = 0$, we set $\lambda_0 = 0.9$. For any $k \geq 1$, we use the optimal decay parameter of the previous tracking run as the starting value. We then choose multiple values d_{start} as the minimal values that ensure that 5%, 10%, ..., 35% of the tracked morsels are executed without decay. This can be computed by sorting the tracked queries by their duration. Heuristically, these parameter choices provide decent latency characteristics. For each chosen value d_{start} , we now refine the decay λ_0 through a local search procedure. We define an initial step width $\alpha_0 = 1$ and search directions $D = \{0.05, -0.05\}$.

During the k 'th iteration of the local search procedure we evaluate the cost function at all points in $(d_{start}, (\lambda_{k-1} + \alpha_{k-1}D)) \cap S$. If all points yield larger cost than $(d_{start}, \lambda_{k-1})$, we set $\lambda_k = \lambda_{k-1}$ and $\alpha_k = 0.5 \cdot \alpha_{k-1}$. Otherwise, we set λ_k to the value with the smallest cost and update $\alpha_k = 1.5 \cdot \alpha_{k-1}$. Finally, we choose the best of all refined starting values.

While this optimization method may appear somewhat crude, we found that it works well in practice. We also tried a multivariate directional search procedure, but found that choosing d_{start} heuristically provides more stable parameter choices. We leave the evaluation of more sophisticated optimizers to future work. In order to obtain deterministic optimization costs, we always perform a total of 7 search steps for each starting value. By choosing $t_t = 20s$

and $t_r = 60s$ on a system with 20 threads, finding optimal parameters takes between 20ms and 100ms, consuming less than 0.01% of the total processing time. Overall, this provides a lightweight self-tuning scheduler design which can adapt to workload changes.

5 EVALUATION

In this section, we experimentally evaluate our design of a self-tuning stride scheduler. Section 5.1 presents the experimental setup used throughout the following benchmarks. The evaluation itself is split into three parts. In Section 5.2, we compare our implementation to other scheduling algorithms that we built into Umbra. Since these experiments take place in the same query engine, they allow for an isolated comparison of the different algorithms. As such, these experiments are the main focus of our evaluation. Section 5.3 investigates the overhead when running our scheduler on systems with high core counts. Finally, Section 5.4 sets our contributions into the context of other database systems. In contrast to the previous experiments, dissecting the benefits that arise from query scheduling becomes significantly harder. After all, database performance is influenced by many complex factors.

5.1 Experimental Setup

Our stride scheduler is designed to work well for mixed analytical workloads, consisting of queries with a high variance in execution time. In order to create a challenging experimental setup, we send a variety of queries into the system at different time points. As the workload should exhibit fluctuations between periods of low and high load, we calculate the spacing of queries by sampling from an exponential distribution with expected value $1/\lambda$. This way, we can control the expected arrival rate of λ queries per second. This is similar to [2] – by not spacing the queries uniformly, we obtain dynamic workload fluctuations with bursts of queries arriving in short succession. To obtain a mixture of long and short running analytical requests, we sample from TPC-H queries at SF3 and SF30. Throughout the evaluation, picking queries at SF3 is three times more likely than picking queries at SF30. This is done in order to obtain a more interesting workload. By choosing short running queries preferentially, we reduce the expected query duration. This enables us to increase λ without oversubscribing the system. Note that while 3/4 of the queries are short running, they only require about 1/4 of the total time being spent on query execution.

We can alter the database load by choosing different values for the expected arrival rate λ . By calculating the mean query duration d , we can obtain a load factor of α by choosing $\lambda = \alpha \cdot d^{-1}$. If we, for example, target a load of 0.95 and obtain a mean query duration of 100ms, we would choose $\lambda = 9.5$.

When scheduling queries for an extended period of time, we will mainly consider load factors in the range $\alpha \in [0.8, 1]$. If the load factor is very small, there will only be a few active queries at any point in time. This reduces the impact of the scheduling algorithm. If we choose a load factor $\alpha > 1$ however, schedulers will quickly suffer from arbitrary performance degradation. After all, the system is permanently oversubscribed. Note that while we do not evaluate the systems under *sustained* oversubscription, load factors which are close to one will still lead to *temporary* oversubscription. We do evaluate the scheduling algorithms in these contexts.

Our self-tuning parameter optimizer uses tracking duration $t_t = 20s$ and refresh duration $t_r = 60s$. The main experiments in section 5.2 and 5.4 are run on a Linux 5.3 system with an Intel Core i9-7900X (10 cores, 20 hardware threads) and 128GB of main memory.

5.2 Comparison Within Umbra

We now compare different scheduling algorithms in Umbra. Since we want to focus on the runtime characteristics of the different algorithms, all queries are pre-compiled. This will later be relaxed in Section 5.4, where we run end-to-end experiments with different systems. As a baseline, we implemented a FIFO and a fair scheduler. The fair scheduler is based on our lock-free stride scheduler, the only difference being that it uses fixed priorities. As such, the fair scheduler also benefits from the lock-free design presented in Section 2. For reference, we also include Umbra’s original scheduler. It tries to minimize workers switching between task sets while remaining as fair as possible. It maintains a queue of the active task sets and balances worker threads uniformly across them. If there are n active task sets and w workers, every task set will obtain either $\lfloor n/w \rfloor$ or $\lceil n/w \rceil$ workers.

We evaluate the system under sustained load α . As outlined in Section 5.1, this is governed by the expected arrival rate of queries per second. We schedule queries for a total of five minutes. Even at the lowest load $\alpha = 0.8$ this results in roughly 3000 queries.

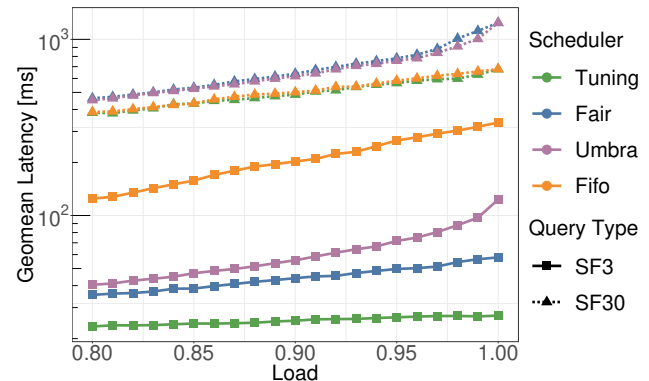


Figure 7: Query latencies under increasing load. The workload consists of queries at SF3 and SF30 running simultaneously. We break down latencies at scale-factor granularity.

Figure 7 shows the latency development of queries when increasing the database load. For each scheduler, we plot the geometric mean of the query latencies at SF3 and SF30 at load $\alpha \in [0.8, 1]$. Note that the two lines for each scheduler do not come from different experiments. Rather, they represent a different set of queries within the same experiment.

Our self-tuning stride scheduler outperforms all other schedulers for both short and long running queries. For short running requests, it is able to retain near optimal latencies, even under high load. The geometric mean of the query latencies at SF3 only deteriorates by roughly 17% when moving from load 0.8 to 1.0. The latencies for fair scheduling meanwhile get worse by 63%. This results in a 2x latency improvement at full load. Keep in mind that the fair scheduler we use for comparison already provides competitive performance, as it builds upon our lock-free scheduler design from Section 2. These

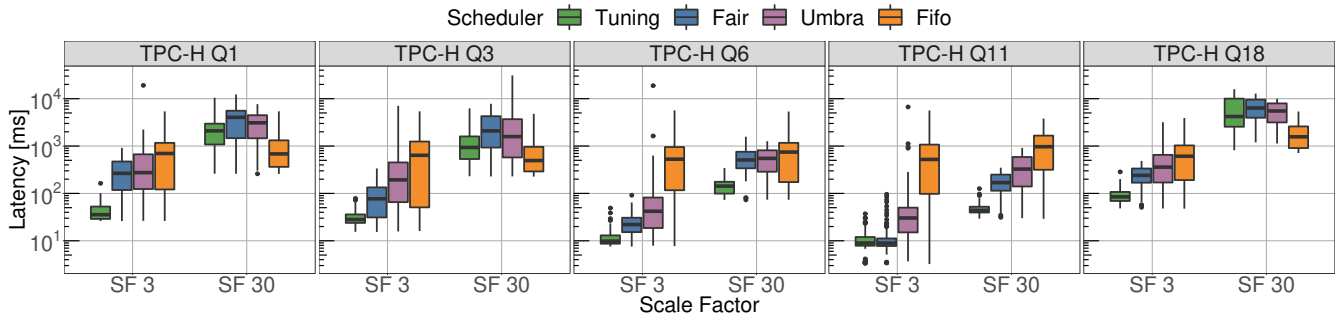


Figure 8: Detailed latency characteristics for selected TPC-H queries at load 1.0. For each scheduler, all data points are taken from the same experiment. In other words, all queries are running simultaneously within the same workload.

benefits can thus be attributed to our self-tuning infrastructure. Comparing this to the original Umbra scheduler, we are able to improve the geometric mean of the query latencies at SF3 by more than 4.5x at full load. Since these queries make up three out of four requests, this significantly improves the user experience. On top of that, we consistently outperform fair scheduling when considering the long running queries at SF30. At load 0.8 we already obtain a 15% improvement with respect to the geometric mean. At full load, we are able to achieve an improvement of almost 2x. At first glance, this might be somewhat surprising. After all, we try to optimize the priority decay in order to treat short running requests in a preferential fashion. This effect is driven by two main factors. First, the latencies of TPC-H queries at SF30 differ by up to an order of magnitude. Thus, the very short running requests at SF30 are still being treated preferentially. Second, even the latency characteristics of the long running requests are improved slightly. This is mainly due to short running queries quickly leaving the system. This leads to fewer active queries at any given time, which reduces scheduling overhead and cache pressure, since workers stick to long running requests more often.

We can also observe that FIFO scheduling is extremely undesirable for mixed analytical workloads. For short running requests, we consistently outperform FIFO scheduling by more than 5x. At load $\alpha \geq 0.95$, the performance difference grows to more than 10x. This is because at high load, the latency of short queries is dominated by their wait time in the preceding FIFO queue.

We now want to turn our attention to the latency distributions of individual queries. Figure 8 breaks down the results at full load based on query type and scale factor. We restrict ourselves to five representative TPC-H queries. The other queries behave similarly.

We first want to study Q1 and Q3. When executed in isolation at SF3, both of them are short running. As such, we can expect our scheduler to treat them preferentially. Compared to fair scheduling, we are able to improve the mean relative slowdown over the base latency by 6.8x and 2.8x for Q1 and Q3, respectively. Even at full load, the queries only suffer an average performance penalty of roughly 50% compared to isolated execution. Furthermore, tuning has a significant impact on the tail latencies of short running requests. We can improve the maximum slowdown by 5.6x for Q1 and 4.2x for Q3. Even under high load, the database is able to retain near optimal latencies for these queries. This leads to far more predictable performance. In addition, we are able to slightly improve mean and tail latencies for the long running requests. While Q18 behaves

similarly, tuning has a weaker impact. This can be attributed to the higher base latency compared to Q1 and Q3.

Q6 and Q11 are both very short running. Interestingly, the positive effects of our tuning scheduler are now more noticeable at SF30. This is because the longer running queries at SF30 will now have, on average, more overlap with other concurrent requests. Thus, they are able to benefit more strongly from preferential treatment. For both requests we are able to improve the mean slowdown by more than 3.4x. For Q11, we can reduce the maximum slowdown by 2.7x. This factor grows to 4.5x when looking at Q6.

Finally, we can see that Umbra’s original scheduler suffers from reduced performance and an extremely heavy latency tail for short running requests. This leads to unpredictable system performance. Once there are more active queries than there are workers, some requests will receive no CPU time over extended periods of time. This especially affects short running queries. For those, we are able to improve latency tails by more than an order of magnitude.

5.3 Scheduling Overhead

We now investigate the scheduling overhead on systems with higher core counts. The experiments in this section are run on a four socket machine with Intel Xeon E7-4870 v2 CPUs. In total, the machine has 60 physical cores, 120 virtual cores and 1TB of RAM.

Figure 10 breaks down the scheduling overhead when increasing the core count. At core count n , we schedule $50 \cdot n$ queries at the same time. For each query, we randomly sample from TPC-H SF3 and SF30 in the same way as in Section 5.2. We then measure the overhead of the different phases until all queries finished successfully. We also *disable* the optimizations at high load outlined at the end of Section 2.3. The numbers thus represent the worst-case overhead. We can see that the total scheduling overhead is negligible. For low core counts it is around 0.05%. It drops to roughly 0.02% when utilizing 120 cores. This is because the relative tuning overhead drops significantly. After all, we restricted this phase to a single core. Meanwhile, the overhead for updating the masks to incorporate new queries increases linearly with the number of cores. When utilizing all 120 cores the overhead induced by this phase is roughly 0.005%. The same holds for the local work of each thread when incorporating these updates into its scheduling state. As we add more cores, the query throughput increases almost linearly. This also leads to more state updates. Finally, we can see that the finalization phase causes almost no overhead.

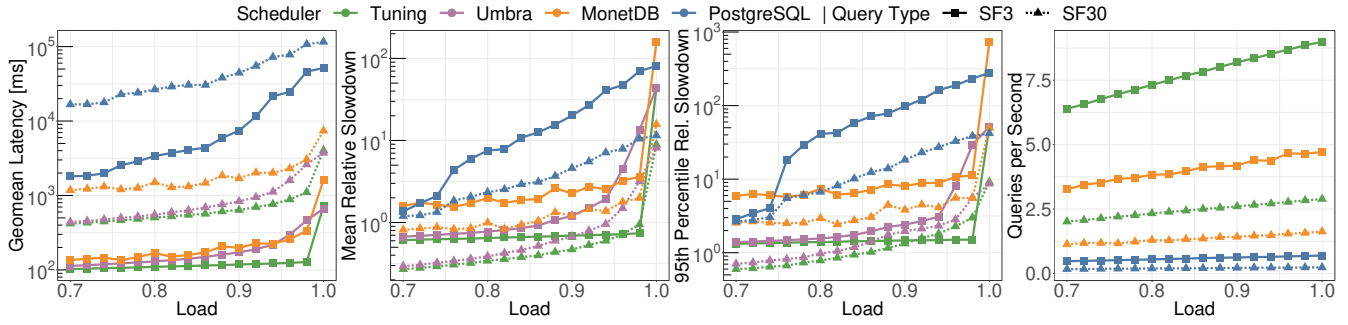


Figure 9: Latencies for different systems under increasing load. The workload consists of queries at SF3 and SF30 running simultaneously. We break down latencies at scale-factor granularity. Queries per second are identical for Tuning and Umbra.

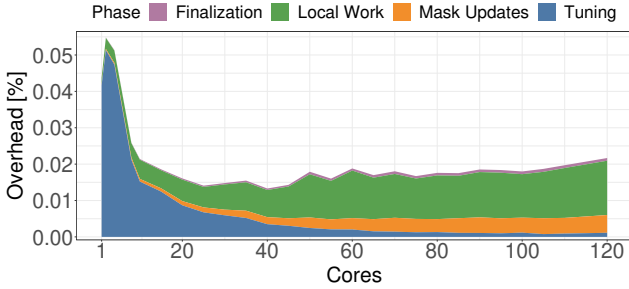


Figure 10: Scheduling overhead with increasing core count.

5.4 Comparison to Other Systems

We now evaluate the performance of our self-tuning scheduler compared to other systems. Designing a fair experiment in this case is significantly harder than before. Query latencies under load are not only determined by the scheduling policy. Rather, they are closely intertwined with the design of the execution engine. We found that for many TPC-H queries, PostgreSQL only saturates a small subset of the available cores. If we utilize the same metrics as in Section 5.2, this will lead to skewed performance results. PostgreSQL will be heavily underutilized at the calculated full load. Instead, we require a metric that captures how query performance deteriorates as the system approaches oversubscription. We say that the system becomes oversubscribed once the mean slowdown of queries within a workload exceeds 50. We define this point as full load $\alpha = 1.0$. This value directly corresponds to an expected arrival rate λ_{max} . For an arbitrary load α' , we can compute the target arrival rate $\alpha' \cdot \lambda_{max}$.

Since a good scheduler should leverage both intra- and inter-query parallelism to maximize performance, we measure the slowdown with respect to the *single-threaded* base latency of a query. This way, a system can actually achieve average values below one at moderate load. Nevertheless, this metric does not necessitate advanced intra-query parallelism in order to retain solid performance. Even a system that does not offer any intra-query parallelism can retain low query latencies at high load, as long as it distributes queries amongst the available cores in a smart way.

Compared to the previous experiments, we do not pre-compile queries in Umbra anymore. This is done to not give Umbra an unfair advantage over the other systems. We evaluate our scheduler against MonetDB (version 11.33) and PostgreSQL (version 11.7). In contrast to our design, both systems bind OS threads directly to

queries. Scheduling is mostly left to the operating system. MonetDB and PostgreSQL perform best when the number of concurrent queries is limited [12, 31]. Otherwise, resource contention reduces the system throughput. We thus limit the maximum number of concurrent queries to 20 for PostgreSQL and 64 for MonetDB. For PostgreSQL, this is done through PgBouncer. For MonetDB, we impose the query limit in our own code on top of its Python connector. For reference, we also include Umbra’s original scheduler outlined in Section 5.2. Each experiment is run for 20 minutes in Umbra and MonetDB. For PostgreSQL, we increase the duration to 30 minutes to account for its lower base performance. This way, each experiment schedules more than 1.5 thousand queries.

Figure 9 presents an overview of the latency characteristics of the different systems as they approach full load. We can see that our self-tuning scheduler consistently provides the best performance. On top of that, we can cope with the highest scheduling pressure. We execute 84% more queries per second than MonetDB. For PostgreSQL, the difference grows to 10x. At low load, our scheduler has a higher relative slowdown for short running queries than for long running ones. This might be surprising since we prioritize short running requests. The effect is caused by code generation, which is not parallelized. For the short running queries, code generation makes up a larger fraction of the overall latency. As a result, long running requests benefit more from hardware parallelism.

In this section, we do not focus on load factors beyond 0.96. At load 1.00, all systems are oversubscribed and suffer from arbitrary performance degradation. By only going up to load 0.96, we give all schedulers some leeway before becoming oversubscribed. This differs from the experiments in Section 5.2, where we used a different metric to investigate factors up to 1.00.

When moving from load 0.70 to 0.96, the mean slowdown of queries at SF3 increases by 2x for MonetDB and almost 30x for PostgreSQL. Meanwhile, performance of our self-tuning scheduler only deteriorates by 18%. At load 0.96, we improve the mean slowdown for these requests by 4.5x compared to MonetDB. For PostgreSQL, this factor grows to more than 65x. The effect on the 95th percentile slowdown is even stronger. At load 0.96, we outperform MonetDB by 7.1x and PostgreSQL by more than two orders of magnitude with respect to the queries at SF3. For the long running requests at SF30, we still provide the best performance. But since our self-tuning scheduler prioritizes short running requests, the improvement is not as strong. Nevertheless, we are the only system able to retain a mean slowdown below 1.0 for both query types at load 0.96.

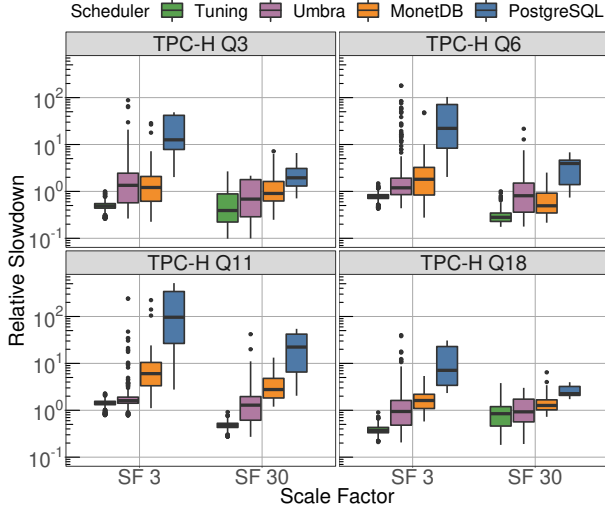


Figure 11: Detailed latency characteristics for selected TPC-H queries at load 0.96. For each system, all data points are taken from the same experiment.

Figure 11 shows a detailed latency breakdown at load 0.96. The graph corroborates our previous findings. We again analyze the mean slowdown, since this is the metric our self-tuning scheduler optimizes for. We first focus on the short running requests at SF3. Here, we are able to improve the mean slowdown over MonetDB by at least 3.5x (Q6). This factor grows to 6.4x for Q11. For PostgreSQL, we are able to improve the mean slowdown for all queries by more than 30x. We can again see that the improvements with respect to the maximum slowdown are even more pronounced. For MonetDB, the improvement ranges from 5.9x in Q18 to more than 90x in Q11. For PostgreSQL, we consistently improve latency tails by more than 30x. The effect is again most noticeable for Q11, where the improvement exceeds two orders of magnitude.

We also improve mean and maximum slowdown for the requests at SF30. For the resource intensive Q3 and Q18, our tuning scheduler still provides slightly better latency characteristics than MonetDB. Both systems outperform PostgreSQL. For the two other requests, the benefits of our tuning scheduler are extremely noticeable. Compared to MonetDB, we improve the mean slowdown of Q11 by more than 8.5x and the maximum slowdown by 14.5x. This is because the queries are extremely short running, even at SF30. As a result, our tuning scheduler treats these requests preferentially.

Overall, we have shown that our self-tuning scheduler is able to significantly outperform traditional systems at high load. For short running requests, the benefits are especially strong. This leads to predictable and high query performance.

6 RELATED WORK

Scheduling of analytical queries has not been studied extensively in the context of database systems. The work of Leis et al. on morsel-driven parallelism provides the foundation of our contributions [11]. However, they primarily focus on optimizing the latencies of isolated requests on NUMA-systems. Psaroudakis et al. investigate task scheduling in SAP HANA [22, 23]. They implement a dynamic worker count in order to improve performance when dealing with

blocking tasks in mixed OLAP/OLTP workloads. Furthermore, partitionable operations are provided with a concurrency hint based on the number of free worker threads. This is used to reduce task granularity as system load increases. How to make good use of this hint is left to the different operators, however. Our design of adaptive morsel sizes introduces a principled approach for altering task granularity in database systems. It allows for highly dynamic changes in the overall scheduling behaviour.

Existing user-space task schedulers are often designed to maximize system throughput for HPC scenarios [7, 26]. These schedulers make it hard to exploit domain knowledge in order to improve workload characteristics. Furthermore, general-purpose task schedulers benefit less from the adaptive task structure outlined in Section 3.1.

Scheduling of large analytical jobs has been studied extensively in the context of cluster computing [27, 28]. However, the scheduling problem being investigated is a different one. In this paper, we focus on how fine-grained work in a single node database can be distributed amongst the available CPU cores. In comparison, scheduling in large clusters is concerned with distributing more coarse grained tasks amongst the available nodes. One prominent example is Spark [4, 32]. Recent work by Kraska et al. proposes to use a reinforcement learning based scheduler to improve latency characteristics [10]. Such an approach makes progress guarantees for active queries extremely hard. Instead of trying to learn an optimal scheduling policy, our design optimizes hyperparameters within the constraints of a fixed scheduling policy.

While we focus on analytical workloads, scheduling transactional requests in database systems poses different challenges. Here, query performance often deteriorates due to excessive aborts and high lock contention. Performance can mainly be improved by restricting which queries are executed concurrently [21, 24].

Similar to our work, both [10] and [24] implement self-tuning capabilities in their scheduler. Nevertheless, the utilization of observed workload characteristics in order to improve scheduling decisions is not a recent idea [17, 25]. To the best of our knowledge, we are the first contribution to show how self-tuning capabilities can be used to improve scheduling for analytical database systems.


In a more broad context, our work on scheduling falls into the important area of workload management within database systems. This is a wide ranging field incorporating topics like workload classification, admission control and resource management for memory and I/O [2, 3, 13, 18, 19].

7 CONCLUSION

This paper showed how task-based parallelism can be utilized to improve the performance of modern analytical database systems. Our lock-free implementation of stride scheduling uses domain knowledge to improve scheduling decisions. This is especially attractive for morsel-driven database systems, for which we showed how the task structure can be made more robust. By making our scheduler self-tuning, we enabled it to make smart scheduling decisions for arbitrary mixed analytical workloads. This was achieved by adaptively optimizing hyperparameters based on the tracked workload. While traditional systems suffer from unpredictable and high query latencies at high load, our self-tuning scheduler alleviated these problems. It retained near optimal latencies for short running requests, while significantly improving latency tails.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their detailed comments which significantly improved the paper. We also thank Jana Giceva and Viktor Leis for their invaluable feedback.

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No 725286). 

REFERENCES

- [1] Andrew R Conn, Katya Scheinberg, and Luis N Vicente. 2009. *Introduction to Derivative-Free Optimization*. Vol. 8. Siam.
- [2] Dominik Durner, Viktor Leis, and Thomas Neumann. 2019. On the Impact of Memory Allocation on High-Performance Query Processing. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*.
- [3] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 592–603.
- [4] Panagiotis Garefalakis, Konstantinos Karanasos, and Peter Pietzuch. 2019. Neptune: Scheduling Suspending Tasks for Unified Stream/Batch Applications. In *Proceedings of the ACM Symposium on Cloud Computing*. 233–245.
- [5] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. 2007. Architecture of a Database System. *Foundations and Trends in Databases* 1, 2, 141–259.
- [6] IBM. 2019. The DB2 Process Model. https://www.ibm.com/support/knowledgecenter/SSEPGG_11.5.0/com.ibm.db2.luw.admin.perf.doc/doc/c0008930.html
- [7] Intel. 2007. Intel Threading Building Blocks. <https://software.intel.com/sites/default/files/m/d/4/1/d/8/tutorial.pdf>
- [8] Alfons Kemper and Thomas Neumann. 2011. HyPer: A Hybrid OLTP&OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *IEEE 27th International Conference on Data Engineering*. 195–206.
- [9] Leonard Kleinrock and Richard R Muntz. 1972. Processor Sharing Queueing Models of Mixed Scheduling Disciplines for Time Shared Systems. *Journal of the ACM (JACM)* 19, 3 (1972), 464–482.
- [10] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H Chi, Jialin Ding, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A Learned Database System. In *9th Conference on Innovative Data Systems Research, CIDR*.
- [11] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. 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*.
- [12] MonetDB Mailing List. 2012. MonetDB Maximum Concurrent Limit. <https://www.monetdb.org/pipermail/users-list/2012-February/005430.html>
- [13] Microsoft. 2017. SQL Server Resource Governor. <https://docs.microsoft.com/en-us/sql/relational-databases/resource-governor>
- [14] Microsoft. 2020. Thread and Task Architecture Guide. <https://docs.microsoft.com/de-de/sql/relational-databases/thread-and-task-architecture-guide>
- [15] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *The Proceedings of the VLDB Endowment* 4, 9, 539–550.
- [16] Thomas Neumann and Michael J Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR*.
- [17] Thu D Nguyen, Raj Vaswani, and John Zahorjan. 1996. Using Runtime Measured Workload Characteristics in Parallel Processor Scheduling. In *Workshop on Job Scheduling Strategies for Parallel Processing*. Springer, 155–174.
- [18] Stefan Noll, Norman May, Alexander Böhm, Jan Mühlig, and Jens Teubner. 2019. From the Application to the CPU: Holistic Resource Management for Modern Database Management Systems. *IEEE Data Engineering Bulletin* 42, 1, 10–21.
- [19] Oracle. 2019. Managing Resources with Oracle Database Resource Manager. <https://docs.oracle.com/en/database/oracle/oracle-database/19/admin/managing-resources-with-oracle-database-resource-manager.html>
- [20] PostgreSQL. 2019. Architectural Fundamentals. <https://www.postgresql.org/docs/12/tutorial-arch.html>
- [21] Guna Prasaad, Alvin Cheung, and Dan Suciu. 2020. Handling Highly Contended OLTP Workloads Using Fast Dynamic Partitioning. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 527–542.
- [22] Iraklis Psaroudakis, Tobias Scheuer, Norman May, and Anastasia Ailamaki. 2013. Task Scheduling for Highly Concurrent Analytical and Transactional Main-Memory Workloads. In *Proceedings of the Fourth International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures*.
- [23] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2015. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *The Proceedings of the VLDB Endowment* 8, 1442–1453.
- [24] Yangjun Sheng, Anthony Tomasic, Tieying Zhang, and Andrew Pavlo. 2019. Scheduling OLTP Transactions via Learned Abort Prediction. In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–8.
- [25] Warren Smith, Valerie Taylor, and Ian Foster. 1999. Using Run-Time Predictions to Estimate Queue Wait Times and Improve Scheduler Performance. In *Workshop on Job scheduling strategies for Parallel Processing*. Springer, 202–219.
- [26] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemariniere, Stefano Markidis, Herbert Jordan, et al. 2018. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing* 74, 4, 1422–1434.
- [27] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. 2013. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*.
- [28] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*.
- [29] Carl A Waldspurger and William E Wehl. 1994. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*.
- [30] Carl A Waldspurger and E Wehl W. 1995. Stride Scheduling: Deterministic Proportional-Share Resource Management. *Technical Memorandum MIT/LCS-TM528*.
- [31] PostgreSQL Wiki. 2014. Number of Database Connections. https://wiki.postgresql.org/wiki/Number_Of_Database_Connections
- [32] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster Computing with Working Sets. *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing*.