Scheduling in a Database-Based Distributed Operating System

by

Shana Mathew

S.B. Electrical Engineering and Computer Science
Massachusetts Institute of Technology, 2020

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2021

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2021

Certified by .....
Michael Stonebraker
Adjunct Professor of Computer Science and Engineering
Thesis Supervisor

Accepted by .....
Katrina LaCurts
Chair, Master of Engineering Thesis Committee

Scheduling in a Database-Based Distributed Operating System

by

Shana Mathew

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2021, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Current operating systems date from over 40 years ago and were designed for very different computing requirements, making them ill-equipped to handle serverless workloads as well as modern challenges in scalability, heterogeneity, availability, and security. Hence, we propose a radically new data-centric OS design for serverless computing. This database OS (DBOS) centralizes all cluster state in a uniform data model: database tables stored in a high-performance, distributed, main-memory database management system. Operations on this state will be performed via serverless, stateless tasks.

This thesis presents work done to build a preliminary scheduler and to implement and evaluate various global scheduling algorithms. We also demonstrate the performance of a modern DBMS in executing various scheduling operations.

Thesis Supervisor: Michael Stonebraker
Title: Adjunct Professor of Computer Science and Engineering

Acknowledgments

I would first like to thank my advisor, Professor Michael Stonebraker, for his continued support and guidance on this project, without which this work would not be possible. I would also like to extend my gratitude to Professors Matei Zaharia and Christos Kozyrakis, as well as Qian Li, Kostis Kaffes, and Peter Kraft for their valuable mentorship and feedback throughout the past year. Finally, I would like to thank my friends and family for their support and inspiration throughout this process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis will elaborate on my contributions to a group effort in initial scheduling experiments as well as individual work on implementing variations of priority-based global scheduling algorithms in DBOS, a proposed DBMS-based operating system. Chapter 1 goes over the motivation for DBOS, and Chapter 2 describes the environment and experiment configurations for the scheduling studies presented in this paper. Chapter 3 discusses the preliminary scheduling studies to explore the limits of a modern DBMS in executing scheduling operations, and Chapter 4 details further experiments in analyzing the overhead of implementations of priority-based scheduling algorithms.

1.1 Motivation

Current operating system software, which are mainly comprised of UNIX/Linux-style derivatives, date from over 40 years ago and were designed for a very different hardware configuration, consisting of uniprocessors with limited main memory, minimal disk space and poor connectivity. However, now with the ubiquitous use of cloud computing, thousands of cores, massive parallelism, several levels of memory and storage, and heterogeneous hardware, including CPUs, GPUs, and FPGAs, all need to be managed. This growth is in particular driven by the popularity of machine learning, IoT, and big data applications, which require more computing resources to

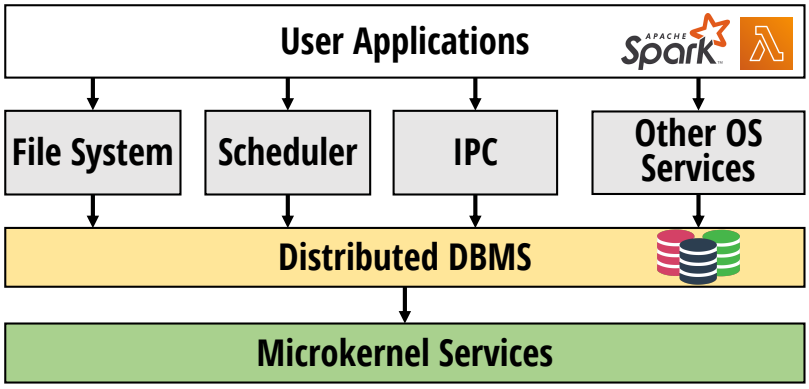


Figure 1-1: Proposed DBOS stack.

support their performance requirements [9, 14]. For example, a large shared system such as the MIT Supercloud [5, 15] has approximately 10,000 cores, a hundred terabytes of main memory, and petabytes of storage. This level of complexity makes it difficult for current-day operating systems to meet demands in scalability, hardware heterogeneity, availability, and security [9].

1.2 System Design and Advantages

In order to address these challenges, we propose a new data-centric OS design for serverless computing. This design centralizes all cluster state in a uniform data model: database tables stored in a high-performance, distributed, main-memory DBMS. Operations on this state will be performed via serverless, stateless tasks. From a high level, the design consists of a high speed, multi-node, transactional DBMS embedded into a kernel of an operating system, as shown in Figure 1-1 [16]. The OS services are DBMS applications which are either written in SQL or are user defined functions.

This design yields several advantages. Building on top of a database means that the guarantees that databases provide, such as high availability and atomic transactions, are also included. Additionally, it can be easily scaled and evolved without refactoring the entire system; the database table schemas can be easily changed, and rather than reimplementing data structures to make them scalable on multicores, only the implementations of common table operations need to be scaled.

These capabilities provided by the distributed DBMS also benefit OS services such as task schedulers, distributed filesystems, and interprocess communication (IPC). Since DBMS tables provide a consistent global view of the OS state, it is much easier to support cross-cutting operations. For example, modern task schedulers usually depend on various state data, e.g. historical performance and resource utilization, to make optimal placement decisions. In cluster managers such as Kubernetes [4] and YARN [19], this information resides on separate layers with no consistency guarantees between them and is exposed by ad-hoc APIs [16].

Similarly, our programming model should facilitate the development of novel OS services and applications. For example, since all data operations by a modern operating system, such as copying, mutating, and transmitting, can be more easily tracked and stored with DBOS, a strong data provenance system can be built. This would provide solutions to modern issues like data forging and data spills [9].

1.3 Initial Work

In order to demonstrate that reasonable performance can be offered, a prototype of the scheduler, filesystem, and IPC OS services was built. This thesis presents work done on the scheduling side: in particular, studies of the performance limitations of a DBMS when carrying out scheduling operations, as well as a comparison and analysis of different global scheduling algorithms.

1.4 Related Work

There are several applications that use declarative interfaces and DBMS concepts in system software. For example, there are filesystem checkers based on a declarative query language [10, 13] and instrumentation frameworks, like OSQuery [6], that provide a declarative interface to OS data for intuitive and performant monitoring and analytics. Cloudburst [17] and Anna [20], two distributed systems, also support building on top of DBMS technology. None of these efforts, however, propose an OS

stack with a DBMS at the bottom, similar to DBOS [16].

On the scheduling side, in DCM [18], developers specify the cluster manager's behavior declaratively, using SQL queries over cluster state stored in a relational database. The DCM compiler synthesizes a program from developer SQL specification, and encodes cluster state as an optimization problem that can be solved using off-the-shelf solvers, meaning that developers will not have to design ad-hoc heuristics themselves. DCM, however, consists of a centralized database and scheduler as opposed to a distributed DBMS; its primary goal is to efficiently schedule long-lived jobs while satisfying complex constraints. DBOS on the other hand aims to quickly schedule huge numbers of short-lived tasks, which have simpler constraints, and has more emphasis on throughput.

Chapter 2

Scheduling Studies

This chapter discusses the environment and baseline scheduling code that we developed in order to demonstrate the performance of a modern DBMS in executing scheduling operations.

2.1 Experiment Environment

2.1.1 VoltDB

In order to support the architecture and intended goals of DBOS, we prioritized the following traits when choosing our underlying DBMS. It should be a distributed Online Transactional Processing (OLTP) DBMS in order to be powerful enough to support most distributed systems operations, and should also support partitioning tables across nodes in order to scale with cluster size. An ACID-compliant transactional database would also allow for ease of development, and an in-memory database would ensure minimal transaction latency due to fast read and write operations.

VoltDB was chosen because it aligns with the above properties as well as because of the group's association with the organization, allowing for technical support to be easily acquired.

VoltDB is an in-memory database that implements SQL on top of tables which are either hash-partitioned on a user-specified partitioning column across multiple

nodes of a computer system, or replicated across all nodes and sites of a VoltDB database. Small tables which are almost never updated are a good candidate for the latter option.

VoltDB promises serializable ACID consistency, and is optimized for OLTP transactions, which move small amounts of data. In VoltDB, concurrency control is optimized for transactions accessing a single partition. Additionally, performance can be maximized by ensuring that a task will be on the same node as the partition being accessed so as to avoid network traffic during a transaction. Transactions in VoltDB are stored procedures [7] which are aggressively optimized to yield high performance.

2.1.2 MIT Supercloud

All of the experiments were run on MIT Supercloud [5], an HPC-style (High-Performance Computing) shared cluster. The compute nodes in the cluster have 40-core dual-socket Intel® Xeon® Gold 6248 2.5GHz CPUs, 378 GB of memory, and a Mellanox ConnectX-4 25Gbps NIC.

2.2 Synthetic Scheduler

In order to measure the performance of VoltDB when executing scheduling operations, a synthetic C++ scheduler was developed.

Several bash scripts were created in order to ensure consistency in the configuration and initialization of the scheduler among all developers in the group. They initialize an instance/cluster of the enterprise version of VoltDB setup on MIT Supercloud according to the same configuration, load SQL files into VoltDB to create tables, and compile and load the Java stored procedures. VoltDB stored procedures are ways to access the database using standard SQL syntax and perform other functions on the return values while maintaining ACID transaction guarantees. Developers can specify that these procedures are partitioned on a specific column of a table; this ensures that the procedure executes within that specified partition of the database [7].

The scheduler itself schedules tasks but does not execute them and assumes that

any worker can execute a fixed number of tasks simultaneously. The worker task capacity, as well as other system parameters are specified via command line arguments.

After each task has been scheduled, the scheduler uses the latency metrics built into the VoltDB client interface to record the time required to process the operation. These metrics are then aggregated for each measurement interval to yield the overall latency and throughput.

Chapter 3

Preliminary Experiments

This chapter covers initial studies conducted as a group to determine VoltDB's performance and identify limitations in executing scheduling operations. It also goes over the implementation and evaluation of various FIFO (First-in, First-out) synthetic scheduler designs.

3.1 Identifying Limitations

In order to explore VoltDB's limits, we decided on a simple strawman design, consisting of just the worker table in Figure 3-2. Workers are evenly distributed among partitions when the scheduler is set up. Each worker is identified by a unique ID and its partition key (Pkey), and has a task capacity as mentioned earlier in Section 2.2. In this design, only a single table transaction occurs per task, meaning it ensures the maximum single-threaded performance possible. While separate bookkeeping would be necessary in order to maintain task information and handle scheduler failures, this design shows the upper bound of VoltDB's performance.

Figure 3-1 shows a stored procedure implementing the transaction used to select a worker with a mix of SQL and imperative code. This procedure is used for the following experiments, and is partitioned on the Pkey column of the Worker table.

```
SelectWorker(k, wID) {
  SELECT WorkerID, Capacity FROM Worker WHERE PKey=k AND Capacity > 0 LIMIT 1;
  UPDATE Worker SET Capacity=Capacity - 1 WHERE WorkerID=wID;
}
```

Figure 3-1: SelectWorker transaction.

```
CREATE TABLE Task (
    TaskID INTEGER NOT NULL,
    WorkerID INTEGER NOT NULL,
    State INTEGER NOT NULL,
    PKey INTEGER NOT NULL
);
PARTITION TABLE Task ON COLUMN PKey;
```

```
CREATE TABLE Worker (
    WorkerID INTEGER NOT NULL,
    Capacity INTEGER NOT NULL,
    PKey INTEGER NOT NULL
);
PARTITION TABLE Worker ON COLUMN PKey;
```

Figure 3-2: Partitioned Task and Worker table schemas.

3.1.1 Scalability of Table Sizes

In order to measure the scalability of table sizes, an experiment was conducted on a single VoltDB partition with 8 parallel scheduler threads. As the number of workers was increased, resulting in more rows in the Worker table, the capacity of each worker was decreased inversely. The total capacity of the system was fixed at 2 million (2×10^6) tasks, which seemed like a reasonable upper-bound. This can be modeled by the following simple equation: $c = \frac{2 \times 10^6}{n}$ for $n \geq 1$ where c is the capacity of a single worker and n is the integer number of workers.

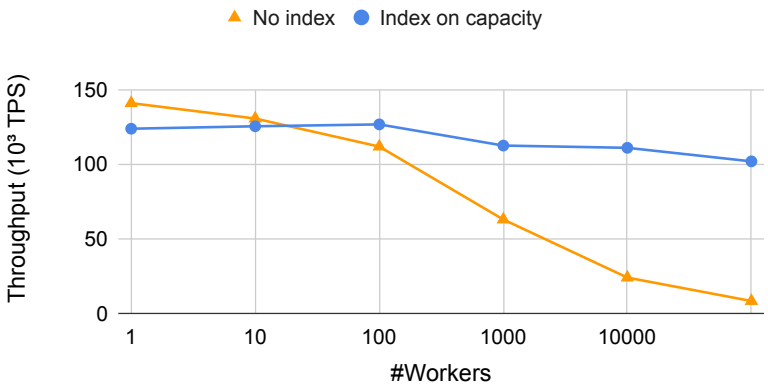
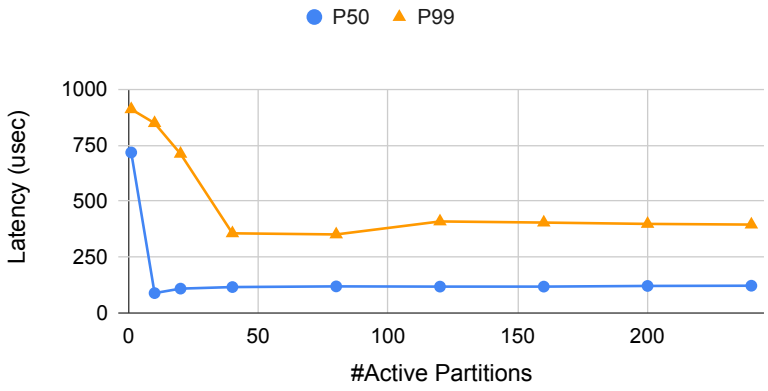


Figure 3-3: Scalability of table sizes.

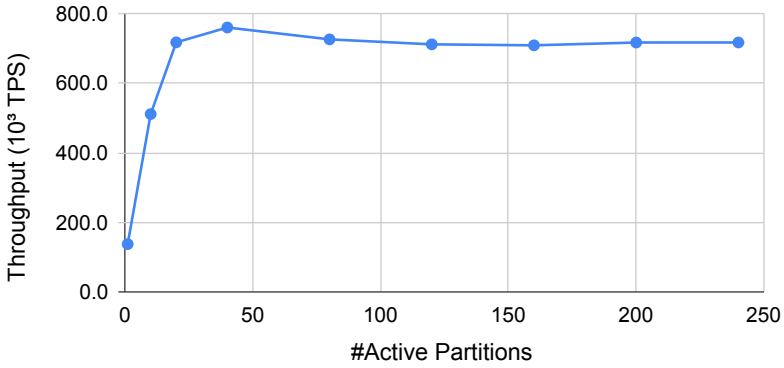
The experiment was also run with and without indexing on the capacity column; the results are as shown in Figure 3-3. While performance does decrease due to more rows in the Worker table, indexing greatly improves scalability. There is, however, a small overhead for maintaining the index; this is seen by the throughput of the experiment run with the index on capacity being smaller than the throughput of the experiment with no index when there are less than ten workers. Despite this, the benefits of the overhead outweigh its cost once there are more than 12 workers.

3.1.2 Scalability of Multiple Partitions

In order to measure the scalability of multiple partitions, an experiment was conducted in which the number of active partitions was varied from 1 to 240. The term



(a) Latencies vs. #active partitions



(b) Throughput vs. #active partitions

Figure 3-4: Scalability of multiple partitions.

active partitions is defined as $min(\#\text{worker groups}, \#\text{VoltDB partitions})$. This study was run on 40 VoltDB partitions on an 80-core Supercloud server with 100 parallel scheduler threads and 8000 workers. Each worker was configured such that they had maximum task capacity.

As seen in Figure 3-4, a maximum throughput of 760k transactions/second is reached when the number of worker groups match the number of partitions. The figure also shows the P50 and P99 latencies. The P99 (99th percentile) graph shows in how many microseconds 99% of the tasks will be scheduled. Likewise, the P50 (50th percentile) shows the median latency, or the time by which 50% of tasks will be completed. When the maximum throughput is reached, the P50 and P99 latencies are 116 and 356 sec, respectively.

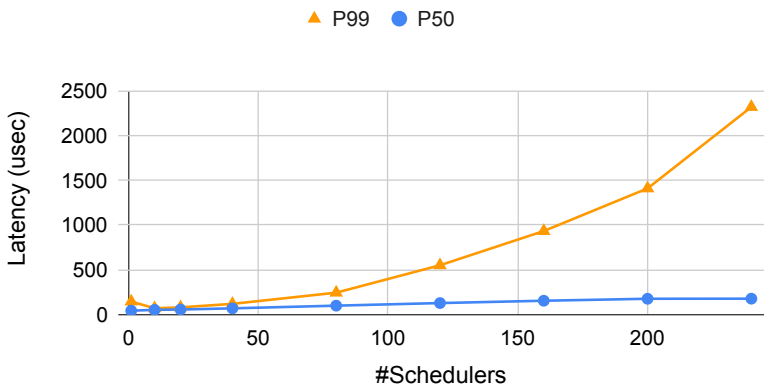
3.1.3 Scalability of Parallel Schedulers

In order to measure the scalability of parallel schedulers, an experiment was conducted in which the number of parallel scheduler threads was varied from 1 to 240. Similar to the experiment setup in Section 3.1.2, the study was run on 40 VoltDB partitions on an 80-core Supercloud server with 8000 workers. All 40 of the partitions were active as this configuration yielded the maximum performance in Section 3.1.2.

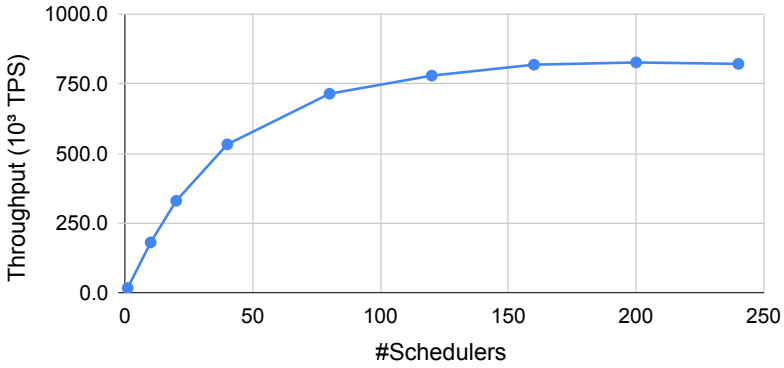
From the results in Figure 3-5, VoltDB is shown to scale well from 1 to 40 parallel schedulers; after that, the throughput flattens out and achieves a maximum of around 826k transactions/second at 200 threads.

3.2 Single-partitioned FIFO Scheduler

Next, a single-partitioned FIFO scheduler was implemented as a stored procedure. This stored procedure accesses two tables: the Worker table and Task table shown in Figure 3-2. The Worker table is the same as the one used in Section 3.1 while the Task table maintains metadata about every task scheduled, including the workerID to which the task is assigned to, the state of the task (unknown, pending, running, completed), and in which partition in the table the task is. Both tables are parti-



(a) Latencies vs. #schedulers



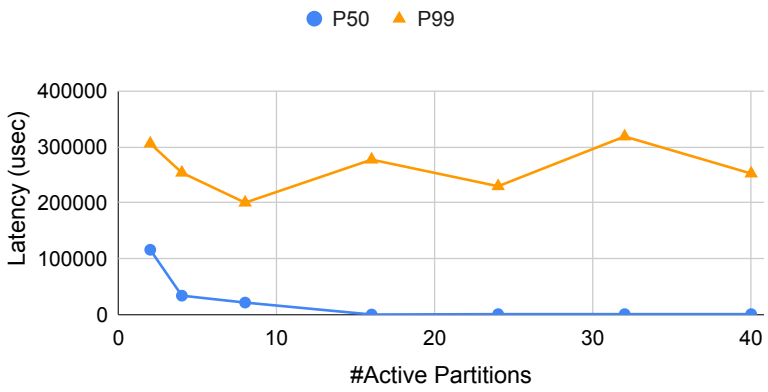
(b) Throughput vs. #schedulers

Figure 3-5: Scalability of parallel schedulers.

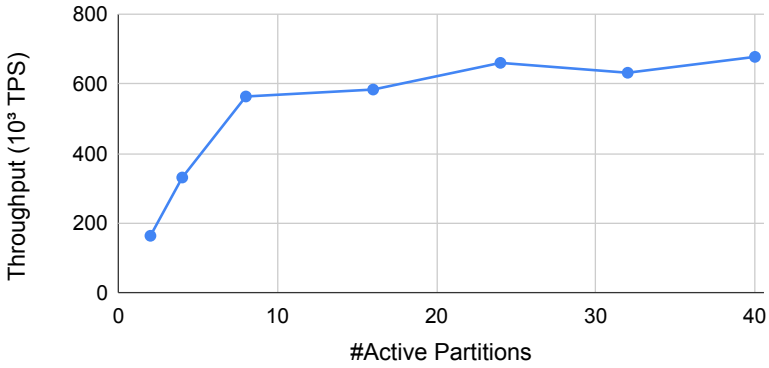
tioned on their respective partition key (Pkey) columns, and the stored procedure is partitioned on the Pkey column of the Worker table. This ensures that that the procedure executes within that particular partition of the database.

The synthetic scheduler initializes the Worker table in setup with the number of specified entries, and then invokes the stored procedure, passing the taskID of the task being scheduled and a randomly chosen partition key. The Worker table is queried for an available worker in the same partition with a capacity greater than zero. If a worker is found, the worker's capacity is decremented by one, and the task is updated. If there are no available workers, the scheduler executes the procedure on another partition, iterating until it succeeds. This procedure touches at least two partitioned tables per transaction.

The scheduler was benchmarked by varying the number of active VoltDB parti-



(a) Latencies vs. #active partitions



(b) Throughput vs. #active partitions

Figure 3-6: Performance of the single-partitioned FIFO scheduler.

tions, and keeping the number of tasks (40), number of parallel scheduler threads (100), and number of workers with maximum capacity (40) constant. The number of partitions was varied from 1 to 40, given that in Section 3.1.2 the performance flattened out after 40 partitions. The results can be seen in Figure 3-6. As the number of active partitions increase, the throughput increases in a logarithmic fashion, and starts to flatten out at 660k transactions/second at 24 partitions.

3.3 Partitioned FIFO Scheduler

In order to understand the effects that global locking has, another FIFO scheduler was implemented, again as a stored procedure, and was compared to the scheduler in Section 3.2. An experiment was conducted on this synthetic scheduler in which the frac-

tion of global transactions (multi-partitioned procedures) executed was varied from 0 to 1.0. This was implemented by simulating the probability of choosing a single-partitioned transaction according to the fraction passed in. The synthetic scheduler initializes the Worker and Task tables in setup with entries and then, according to the simulated probability, invokes either the single-partitioned stored procedure or the multi-partitioned procedure for each task.

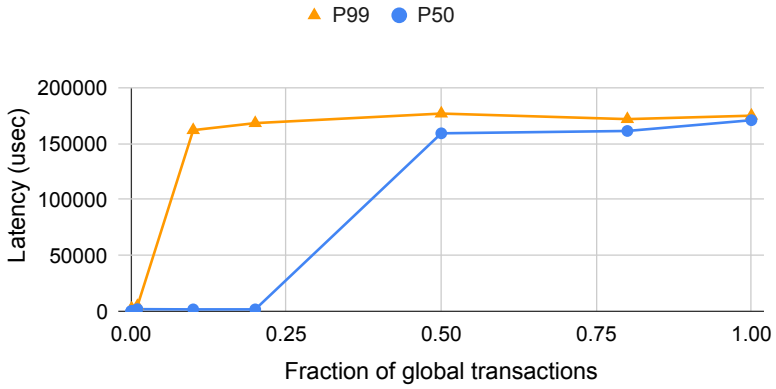
Similar to the FIFO scheduler in Section 3.2, the synthetic scheduler initializes the Worker and Task tables in setup with entries according to the appropriate command-line arguments, and then according to the simulated probability, invokes either the single-partitioned stored procedure or the multi-partitioned procedure for each task.

If it invokes the single-partitioned stored procedure, a randomly chosen partition key is passed in, and the Task table is queried for an unassigned task with that particular partition key. If one is found, the Worker table is then queried for an available worker in the same partition. If a worker is found, the worker's capacity is decremented by one, and the task is updated. All of these mentioned queries are executed in the same transaction. The scheduler executes this procedure on a randomly chosen partition, looping until it succeeds. The procedure touches at least two partitioned tables per transaction and is partitioned on the Pkey column of the Worker table.

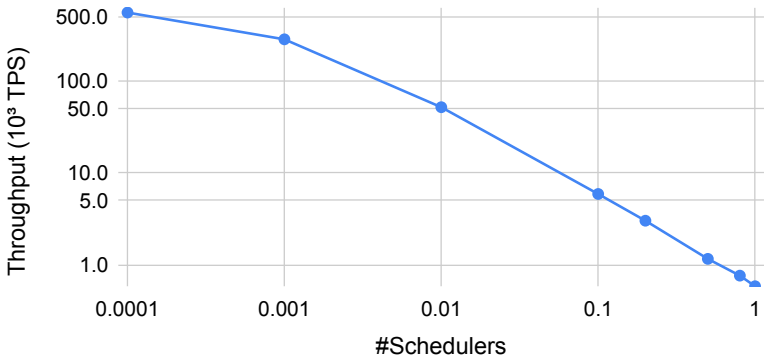
If the scheduler invokes the multi-partitioned stored procedure, the Task table is queried for an unassigned task with any partition key. If one is found, the Worker table is then queried for an available worker in any partition. Like the single-partitioned procedure, if a worker is found, the worker's capacity is decremented by one, and the task is updated. The scheduler executes this procedure on a randomly chosen partition, looping until it succeeds.

This experiment was run on 40 VoltDB partitions, with 40 tasks (1 per partition), 100 parallel scheduler threads, and 40 workers with maximum capacity. As seen in Figure 3-7, the two-table single-partitioned transaction performs well, with the P99 latency being 576 μs, and the throughput being 619k transactions/second. However, even a tiny fraction of global transaction greatly impacts the overall performance,

especially the P99 latency and throughput. Without partitioned transactions, the throughput decreases by a factor of over 1000 to 586 transactions/second. Therefore, in order to maximize performance, it is crucial that transactions limit the number of partitions they touch as much as possible.



(a) Latencies vs. fraction of global transactions



(b) Throughput vs. fraction of global transactions

Figure 3-7: Performance of the partitioned FIFO scheduler.

Chapter 4

Global Scheduling Experiments

While the preliminary experiments focused on FIFO schedulers, other scheduling algorithms are often used, such as CFS in Linux [3], proportional deadline-driven scheduling in Shinjuku [12], and delay scheduling in Spark [21].

In a distributed setting, global scheduling algorithms differ from local ones in that global scheduling algorithms specify how exactly to assign tasks to worker machines from a shared global queue, while local scheduling algorithms run on each worker and specify which tasks to run first from its local queue.

In particular, this chapter will cover the TPC-H benchmark, then different implementations of a synthetic scheduler that uses Shortest Job First (SJF), a priority-based algorithm, and how their overhead compares to FIFO and each other.

4.1 TPC-H Dataset

TPC-H is an industry standard decision support benchmark developed by the Transaction Processing Performance Council (TPC). It consists of business-related ad-hoc queries and concurrent data modifications, and is widely used to evaluate database systems in both industry and academia. While TPC-C was considered, especially because VoltDB itself uses a TPC-C like benchmark, it does not provide tools for data and query generation and is also older than TPC-H.

TPC-H was also chosen over publicly available cluster traces like the Alibaba

Cluster Trace Program [1] and the Google Borg trace [2]. Both represent workloads that include long-running tasks, and so these did not seem as representative of short-running serverless jobs, the primary target of DBOS workloads. While TPC-H also does not accurately represent the targeted workloads, it does allow users to scale the size of the generated dataset by different factors. This allowed more flexibility in generating tasks with different execution times.

Overall, TPC-H seemed like a good starter benchmark due to its ease of integration, and could easily be changed in the future if the need arose.

4.2 Data Generation and Integration

Since TPC-H doesn't provide inbuilt support for VoltDB, several modifications had to be made in order to integrate the TPC-H benchmark into the synthetic scheduler. The benchmark consists of several tables, for each of which SQL schemas had to be manually created with partitioning columns specified as in accordance to the benchmark guidelines. After running the database generator tool with a scale of 1, the files containing the dataset needed to be parsed and modified so that the inbuilt VoltDB CSV loader could load the data into the tables. Each of the 22 query templates were modified in order to be compatible with the VoltDB SQL Data Definition Language (DDL).

In order to acquire data for priority-based scheduling algorithms, the 22 queries were run individually using VoltDB sqlcmd, a built-in interactive tool that allows users to directly execute SQL statements. Their execution times were measured, the results of which can be found in Table 4.1. Only a subset of them could be run due to VoltDB not supporting certain SQL operations.

When implemented as stored procedures, many of the queries throw runtime ex-

Query Number	Q1	Q2	Q3	Q4	Q6	Q10	Q12	Q14	Q16	Q18	Q21
Execution Time (s)	7.34	.41	1.88	.86	1.44	1.63	.72	.84	.81	4.85	5.66

Table 4.1: TPC-H query execution times.

```
CREATE TABLE Task (
    TaskID INTEGER NOT NULL,
    WorkerID INTEGER NOT NULL,
    State INTEGER NOT NULL,
    ExecutionTime INTEGER NOT NULL,
    PKey INTEGER NOT NULL
);
PARTITION TABLE Task ON COLUMN PKey;
```

```
CREATE TABLE Worker (
    WorkerID INTEGER NOT NULL,
    Capacity INTEGER NOT NULL,
    PKey INTEGER NOT NULL
);
PARTITION TABLE Worker ON COLUMN PKey;
```

Figure 4-1: Partitioned Task and Worker table schemas.

ceptions. The two primary causes of this was the query not being plannable given VoltDB's architecture, and subquery expressions in the query being only supported for single-partitioned procedures [8]. Therefore, the scheduler was implemented such that it synthetically schedules tasks by using the execution times to determine their priority but does not execute the tasks themselves. While this method only allows for the overhead of global scheduling algorithms to be measured, integrating these benchmarks paves the way for future work in simulating worker execution and performing a head-to-head comparison of these global scheduling algorithms.

4.3 Shortest Job First

Under the model that one TPC-H query equates to one job, algorithms that rely on a preemptive model, such as fair scheduling, could not be chosen due to VoltDB procedures being non-preemeptive in order to ensure atomicity. Therefore, the smallest level of granularity in scheduling could only occur on a per-query basis, and so a non-preemptive algorithm was chosen to meet the constraints in the dataset and execution environment.

Shortest Job First is a popular non-preemptive, priority-based scheduling algorithm in which the scheduler executes the waiting job with the smallest execution time. The benefits of a priority-based scheduling algorithm like SJF is that it is generally efficient and work-conserving because no worker will idle while an unscheduled available task exists [11].

While the synthetic scheduler in Chapter 2 simply scheduled a task as soon as it arrived, with a priority-based algorithm, the highest priority task is unknown at task creation time and so an aggregation query must be performed in order to find the right task to schedule. The synthetic scheduler framework from Section 2.2 was augmented to support other algorithms than FIFO, and two SJF scheduler designs were implemented and benchmarked in the same manner as the single-partitioned FIFO scheduler in Section 3.2. The first design consists of one single-partitioned and one multi-partitioned transaction; the second combines these two into a single multi-partitioned transaction.

4.3.1 Multi-transactional Scheduler

The first SJF scheduler design partakes in two transactions per task at minimum. The first stored procedure is single-partitioned, and is partitioned on Pkey column of the TaskQueue table seen in Figure 4-2. A randomly chosen partition key is passed in along with the taskID and its execution time, and the procedure inserts the incoming task into the TaskQueue table.

The second stored procedure is multi-partitioned, and is partitioned on the Pkey column of the Worker table in Figure 4-1. When this stored procedure is invoked, it uses the same partition key as was passed in the first procedure. Similar to the procedure in Section 3.2, first the Worker table is queried for an available worker in the same partition. If a worker is found, then an aggregation query to select the unassigned task with the shortest execution time is performed. If a task is selected, the TaskQueue table is updated to mark the state of that task as assigned, the worker's

```
CREATE TABLE TaskQueue (
    TaskID INTEGER NOT NULL,
    ExecutionTime INTEGER NOT NULL,
    PKey INTEGER NOT NULL,
    State INTEGER NOT NULL,
);
PARTITION TABLE TaskQueue ON COLUMN PKey;
```

Figure 4-2: Partitioned Task Queue schema.

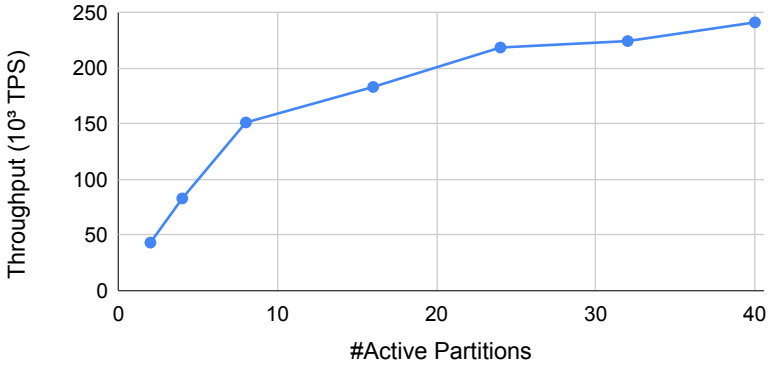


Figure 4-3: Performance of the multi-transactional SJF scheduler.

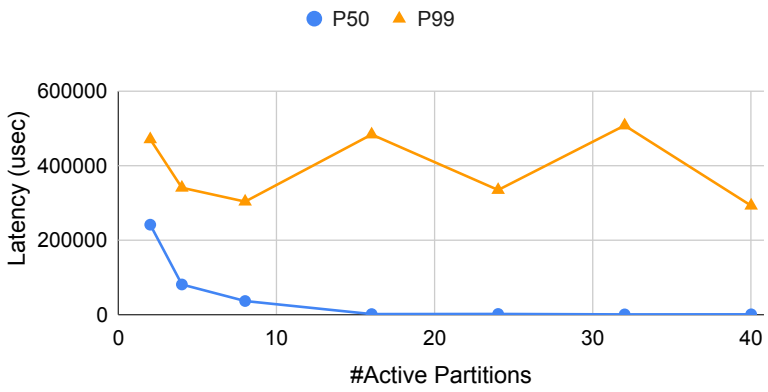
capacity is decremented by one, and then a record is created for that task in the partitioned Task Table in Figure 4-1. If there are no available workers, the scheduler executes the procedure on another partition, iterating until it succeeds.

The scheduler's performance is shown in Figure 4-3. The throughput is quite poor, reaching a maximum of 240k transactions/second at 40 active partitions—a 97% decrease from the single-partitioned FIFO scheduler. This is likely due to two reasons: the aggregation query that touched all partitions of the TaskQueue table in the second transaction, and the increased lock-contention from having a finer-grained design that uses two procedures instead of one.

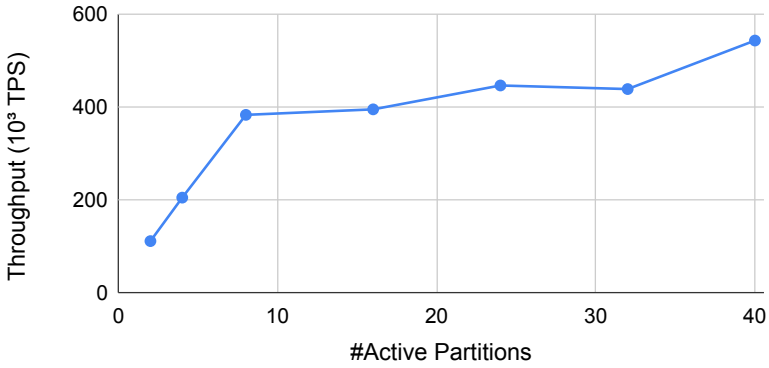
4.3.2 Single-transactional Scheduler

In order to understand the cost of the aggregation query, which is inevitable with a scheduler that uses SJF, the second scheduler design is implemented as a stored procedure that combines the two transactions from Section 4.3.1. This coarser-grained design allows us to study the effects of any lock-contention in the previous SJF scheduler. It also provides a better environment to determine the cost of an aggregation query as this design is more comparable to the single-partitioned FIFO scheduler in Section 3.2.

The stored procedure in this design is partitioned on Pkey column of the Worker table in Figure 4-1. A randomly chosen partition key is passed in along with the



(a) Latencies vs. # active partitions



(b) Throughput vs. # active partitions

Figure 4-4: Performance of the single-transactional SJF scheduler.

taskID and execution time of the incoming task, and the procedure queries the Worker table for an available worker in the same partition. If a worker is found, then the incoming task is inserted into the TaskQueue table, and then the unassigned task with the shortest execution time is queried from the TaskQueue table. If a task is found, similar to Section 4.3.1, the TaskQueue table is updated to mark the state of that task as assigned, the worker's capacity is decremented by one, and then a record is created for that task in the partitioned Task Table in Figure 4-1. Again, if there are no available workers, the scheduler executes the procedure on another partition, looping until it succeeds.

In comparison to the multi-transactional scheduler, the single-transactional scheduler, as shown in Figure 4-4, yields a 55.8% increase in throughput. This increase in performance is likely due to less lock-contention with this coarser-grained design

which consists of only one transaction.

However, compared to the single-partitioned FIFO scheduler from Section 3.2, at 40 active partitions, the single-transactional SJF scheduler has a throughput of 543k transactions/second and a P99 latency of 292 ms, compared to the single-partitioned FIFO scheduler's 677k transactions/second throughput and 252 ms latency. Therefore, the overhead of SJF, in particular the aggregation query across all partitions of the TaskQueue table, causes its throughput to be reduced by 19.8%.

Chapter 5

Conclusion

In this thesis, I presented contributions to a group effort in building a preliminary synthetic scheduler, as well as analyzing the limits of a modern DBMS when executing various scheduling operations. These results allowed us to quantify the effect of various configuration parameters on the throughput and latency achieved by VoltDB. Additionally, these were used to determine the optimal configuration for a scheduler, and were used in later experiments that assessed the performance of various schedulers.

I also described individual work in developing and evaluating priority-based global scheduling algorithms. The results from this allowed us to gain insights into roadblocks that may occur if more complex schedulers are implemented in the future. Because the priority-based global scheduling algorithms developed require an aggregation query that touches multiple partitions of a table in order to find the job with the highest priority, their performance was impacted. The overhead caused by this expensive operation resulted in a 19.8% decrease in performance compared to a simple FIFO scheduler which does not need to manage this extra state. Additionally, breaking up this transaction into one single-partitioned write-only procedure and one multi-partitioned procedure that performs both reads and writes does not improve this performance. In fact, it causes a 55.8% decrease in throughput in the case of the multi-transactional and single-transactional SJF schedulers, likely due to increased lock contention with this finer-grained scheme. Therefore, when implementing future

scheduling algorithms, we should be mindful of:

1. The benefit of having more procedures that are executed within a single partition vs. the cost of increased lock contention.

2. The benefit of a more complex algorithm vs. the cost of multi-partitioned transactions.

While this paper analyzes exclusively the overhead of various FIFO and SJF implementations, it sets up the framework for further work on benchmarking different scheduling algorithms given a TPC-H trace. While VoltDB does not permit many of the queries to be executed as stored procedures, once workers are integrated with the scheduler, each worker can either simulate execution by suspending its corresponding thread for the amount of time specified in Table 4.1, or can execute the query as an ad-hoc query.

In terms of future work for DBOS as a whole, the next stage will involve implementing a serverless environment in user code. This will be built on top of the existing service prototypes in IPC, filesystem, and scheduling. This will prove that OS functions can be readily and compactly coded in SQL and that our filesystem, scheduling and IPC implementations work well in a real system [16].

Bibliography

[1] Alibaba production cluster data. https://github.com/alibaba/clusterdata.

[2] Google cluster workload traces. https://github.com/google/cluster-data.

[3] Linux Scheduler. https://www.kernel.org/doc/html/latest/scheduler/index.html.

[4] Kubernetes Single Resource Api. https://kubernetes.io/docs/reference/using-api/api-concepts/#single-resource-api, 2021.

[5] MIT Supercloud. https://supercloud.mit.edu, 2021.

[6] OSQuery. https://osquery.io/, 2021.

[7] VoltDB Stored Procedures. https://docs.voltdb.com/tutorial/Part5.php, 2021.

[8] Oleg Borisenko and David Badalyan. Evaluation of SQL benchmark for distributed in-memory Database Management Systems. *International Journal of Computer Science and Network Security*, 18(10), Oct 2018.

[9] Michael Cafarella, David DeWitt, Vijay Gadepally, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, and Matei Zaharia. DBOS: A Proposal for a Data-Centric Operating System, 2020.

[10] Haryadi Gunawi, Abhishek Rajimwale, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Sqck: A declarative file system checker. pages 131–146, 01 2008.

[11] Philip L. Holman and James H. Anderson. *On the Implementation of Pfair-Scheduled Multiprocessor Systems*. PhD thesis, 2004. AAI3140333.

[12] Kostis Kaffes, Timothy Chong, Jack Tigar Humphries, Adam Belay, David Mazières, and Christos Kozyrakis. Shinjuku: Preemptive scheduling for second-scale tail latency. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 345–360, Boston, MA, February 2019. USENIX Association.

[13] Marshall Kirk McKusick, Willian N Joy, Samuel J Leffler, and Robert S Fabry. Fsck- The UNIX† File System Check Program. *Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version*, 1986.

[14] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, and et al. Scalable system scheduling for HPC and big data. *Journal of Parallel and Distributed Computing*, 111:76–92, Jan 2018.

[15] Albert Reuther, Jeremy Kepner, Chansup Byun, Siddharth Samsi, William Arcand, David Bestor, Bill Bergeron, Vijay Gadepally, Michael Houle, Matthew Hubbell, et al. Interactive supercomputing on 40,000 cores for machine learning and data analysis. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2018.

[16] Athinagoras Skiadopoulos, Qian Li, Peter Kraft, Kostis Kaffes, Daniel Hong, Shana Mathew, David Bestor, Michael Cafarella, Vijay Gadepally, Goetz Graefe, Jeremy Kepner, Christos Kozyrakis, Tim Kraska, Michael Stonebraker, Lalith Suresh, and Matei Zaharia. DBOS: A DBMS-oriented Operating System. 2021.

[17] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M Faleiro, Joseph E Gonzalez, Joseph M Hellerstein, and Alexey Tumanov. Cloudburst: Stateful Functions-as-a-Service. *arXiv preprint arXiv:2001.04592*, 2020.

[18] Lalith Suresh, João Loff, Faria Kalim, Sangeetha Abdu Jyothi, Nina Narodytska, Leonid Ryzhyk, Sahan Gamage, Brian Oki, Pranshu Jain, and Michael Gasch. Building Scalable and Flexible Cluster Managers Using Declarative Programming. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 827–844. USENIX Association, November 2020.

[19] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, Santa Clara, California, 2013. Association for Computing Machinery.

[20] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. Eliminating Boundaries in Cloud Storage with Anna. *arXiv preprint arXiv:1809.00089*, 2018.

[21] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, page 265–278, New York, NY, USA, 2010. Association for Computing Machinery.