High-Performance Database Management System Design for Efficient Query Scheduling

Deepayan Patra

CMU-CS-22-155
December 2022

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Keywords: Scheduling, Query Execution, Memory

Abstract

Decades of research in the field of database management systems (DBMSs) have focused on improving system performance. Modern analytical systems leverage innovative execution methods, such as vectorization and compilation, or enable parallelizing execution at the operator level to reduce single-query runtimes. Unfortunately, further developments to improve single-query execution performance have failed to yield significant improvements and are providing diminishing performance returns.

To extend beyond the limits of single-query performance improvements, we propose a co-design method to align database and queueing theory research in workload and architecture-aware scheduling policies. In this work, we present the addition of a scheduling component to a highly optimized execution engine and the design of new scheduling algorithms combining awareness of query characteristics and the execution hardware. Our proposed scheduling policies order and assign query sub-tasks to compute resources to enhance performance on analytical workloads in a modern, in-memory execution environment. Further improvements to the execution framework address imbalanced data access patterns and enable locality-aware execution. By optimizing for resource efficiency, our developments decrease the average system latency by over 30%.

Acknowledgement

I am privileged to have known Andy Pavlo for almost five years now. Andy is not just my advisor for my Master's Thesis but the catalyst for my interest in computer science and systems research. I first met Andy when he served as my project advisor in a high school summer program. In the time since, Andy not only taught me about databases and all aspects of computer systems, but guided me in my career path, helped me make critical academic decisions, and initiated my involvement in the research that evolved into my Master's thesis. On top of being an exceptional educator and supportive advisor, Andy is also one of the kindest and most wonderful people I know. In various ways, since our first meeting, Andy has always looked out for my personal and professional development. No other person has provided me with so many opportunities and belief in my abilities. I am grateful that I have had and always will have Andy as a valued and trusted mentor.

I am incredibly fortunate not only to have had one research advisor but three! I learned a great deal about the principles and practice of queueing theory from Ben Berg and Mor Harchol-Balter. As a person who loves systems but appreciates theory, I'm constantly in awe of their theoretical work's effective and applicable results. Ben, my research mentor, has a great penchant for learning and applying his work to new domains — in working closely with him on this system, I had the opportunity to see this quality firsthand, one I hope to one day match myself. I particularly appreciate his help and feedback in preparing me to present this work and others. I found Mor an excellent guide to exploring queueing theory at a time when I had no prior background or expertise in this area. Her teachings were always helpful, and she made learning about a brand-new area of research very approachable.

I also want to thank Justine Sherry for taking the time to serve on my thesis committee. I had the incredible opportunity to publish my first research paper as a primary co-author due in significant part to Justine's guidance. Justine not only helped us ideate the basis of our project but helped transform it into a paper worthy of conference consideration.

I have many more people to thank for helping me on my journey as a part of this degree. First and foremost, I want to thank Emmanuel Eppinger for being the very best research partner I could have asked for. I loved the time we spent working together to build out our threading ideas in NoisePage and seeing our work grow into a research project. I owe a debt of gratitude to Tanuj Nayak, Wan Shen Lim, Matt Butrovich, William Zhang, Prashanth Menon, and Lin Ma. Thank you all sincerely for welcoming me with open arms to the CMU Database Group. You helped me learn about systems, programming, debugging, and research techniques. I appreciate all you do for the research group, and your advice and support are some of the things I'll cherish most from my time here.

I am thankful to Tracy Farbacher, Sara Golembiewski, Karen Lindenfelser, and Joan Digney for helping me in so many ways in this program. Because of your support, I had some wonderful opportunities to share my work with audiences inside and outside CMU. I also want to thank Mark Stehlik and Matt Fredrikson for supporting my decision to consider this program as I evaluated potential future career paths.

And the most important thank you of all. Words cannot express the appreciation I have for my family. My mother, Debjani Baksi, and father, Angshuman Patra, have been part of my educational journey from the very beginning. I developed my passion for learning and devotion to work by seeing the effort they put into raising me and providing for me. I would not have gotten where I am today without their care, love, guidance, and support. Thank you for all the warm embraces, patient company, and meaningful advice through my life's happiest and most challenging moments. You both are always a part of every single thing I do. I love you.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

From initial usage and popularization, database management systems (DBMSs) have been critical software systems supporting data applications. The first database software were built in the 1960s, with popularity rising in the 1970s and 1980s using E. F. Codd's relational model [39]. These systems set up data stores as relations accessible by a declarative programming framework, separating the language interface from internal data retrieval operations. In the decades since, DBMSs have expanded to various domains, with an ever-growing set of features, interface dialects, and architectures and a heavy focus on applications in data storage, analytics, and information retrieval.

The relational model's declarative framework allowed database developers to independently design the language frontend and execution backend in a database. A user's query specifies their data request without instructing the DBMS how to retrieve the data. The system internally converts each query received into an execution plan consisting of operators implemented by the execution engine. The backend can then execute the resulting plan according to an internal processing model.

Execution continues to be a first-order concern for many DBMSs to effectively support performance-sensitive applications. Decades of research in this area have resulted in performant data structures (tree indexes [44, 78], hash tables [26]), result precomputation (materialized views [24]), new query execution techniques (vectorization [35, 49, 61, 62, 66, 80] and compilation [61, 62, 63]), and new data processing algorithms (parallel execution [28, 71]).

Unfortunately, optimizing single-query execution performance no longer achieves the same factors of improvement realized by the aforementioned techniques. Comparisons of the most performant operator algorithms demonstrate similarities at the levels of instructions and instructions per cycle [71]. Comparisons of alternative execution frameworks also demonstrate limited improvement depending on query characteristics [49, 61].

As a result, our work focuses on taking a broader viewpoint on performance optimization by looking at workload-level metrics. In particular, we consider applying and improving scheduling approaches in databases as a potential solution for the limitations of single-query performance.

We first present background on the types of DBMSs we aim to improve. We focus on key developments in execution techniques that have improved single-query performance as a foundation for further work. We then introduce concepts of parallelism and scheduling in the context of databases.

1.1 Analytical In-Memory Databases

Analytical applications, analyzing data to construct dashboards and reports, are a common class of workloads. These workloads are processing-intensive, issuing high-cardinality reads, computations, and transformations. To improve user experience, databases attempt to achieve low latency and high throughput. Although these types of queries often rely on reading a relatively large amount of data for their operations, significant advancements in memory technologies now allow large datasets to reside entirely in memory. Today, high memory capacity machines are also more easily accessible on cloud providers, reducing the cost and maintenance barriers of on-premises hardware. AWS, for example, supports instances with 24TB of memory [30]. Needing less interaction with network drives or disks for storage, single-node in-memory systems are not bottlenecked by data access costs. As such, performance optimizations in such systems have focused further on eliminating CPU and memory bandwidth bottlenecks.

Recent research literature has focused on two main avenues for query performance gains: vectorization and compilation.

1.1.1 Vectorization

A database backend applies an internally defined processing model to process data and respond to a user's query. The processing model executes the operator tree of a query plan by performing the corresponding operations. The traditional implementation approach in many disk-oriented systems processes a tuple at a time and is known as the iterator processing model [43]. Each operator in the plan tree will call a **`next()`** function on its children to retrieve and process the subsequent tuple to process. An alternative model, the materialization model, instead requires each operator to process all inputs to emit a complete set of outputs, executing the plan one operator at a time.

The iterator and materialization models both have high overheads, with the former requiring many operator invocations and the latter requiring large intermediate result buffers. The vectorization model addresses these issues by expanding the iterator model to act upon a batch of tuples, reducing the invocation costs of retrieving tuples while maintaining reasonable dataset sizes.

This approach also enables CPU-level parallelism, simultaneously allowing operation on multiple tuples with SIMD instructions for many database operations. Adding two arrays element-wise, for example, can be processed in a data-parallel fashion. Figure 1.1 shows this procedure for 64-bit numerics, computing the output array four elements at a time with the **`vpaddq`** instruction. Compilers may be able to optimize some internal operations, but researchers have developed targeted SIMD algorithms for supporting other internal functionality [61, 62, 66].

1.1.2 Compilation

Another execution approach for faster evaluation is to specialize code on a per-query basis. After other database system components generate a query plan for the execution engine to operate on, a compilation framework will generate machine code kernels for the operators composing the plan. These machine code kernels are specialized to the types and operations of the query but

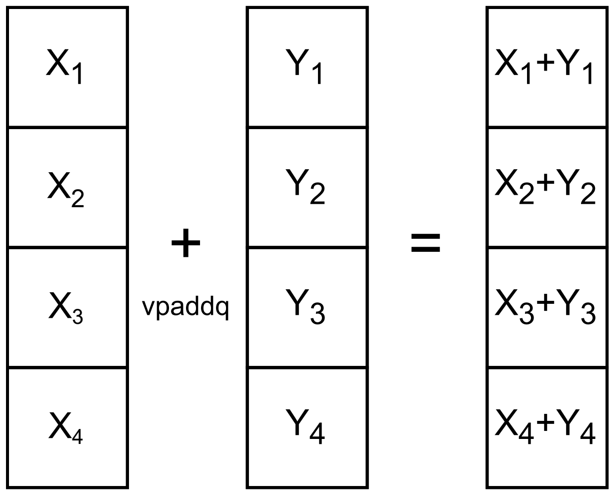


Figure 1.1: The AVX2 SIMD instruction `vpaddq` sums packed quad word values in a pair of vector registers to an output register [48].

can be reused across multiple data segments or repeated invocations of the same query. This technique attempts to elide the indirection necessary in a generic operator for different data types or execution paths and optimize the evaluation of predicates. Systems often choose between transpiling to an alternative imperative source language, such as C or C++, and then generating machine code [51, 64, 72], or converting to an alternative intermediate representation or domain-specific language, which the database can construct machine code from [61, 62, 63, 64].

Query compilation also enables pipelining a series of operations on a data segment. Certain sequences of operators can be applied in order without stalls or can be combined together. For example, operator fusion enables computing a scan operating with a filter predicate or a pair of arithmetic aggregations as a single operation. Doing so reduces cache and memory pressures as operations will not swap their working set as frequently.

1.2 Parallelism and Queueing

Parallel execution in database systems enhances the aforementioned execution frameworks to operate closer to the memory bandwidth and maximum CPU load on a single machine. Parallel processing leverages modern multi-core compute architectures to break down operations and execute these components across independent processors concurrently. Most modern databases support the parallelization of query tasks, taking advantage of improved CPU resource utilization to significantly reduce single-query latency and increase throughput.

Computational tasks generally have varying degrees of parallelism. Operations within a task may have strict dependency orderings which force serial execution. Others may be executed independently in parallel. We describe the parallelizability behaviors of a task as its parallelism characteristics [32].

The parallelism characteristics of queries in databases raise interesting questions regarding resource allocation to optimize system metrics. Queueing theory provides a framework to evalu-

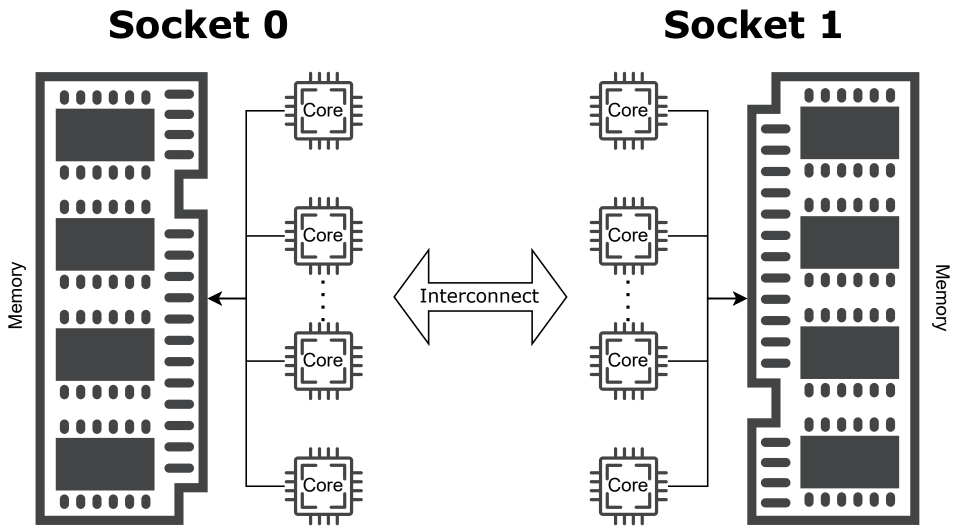


Figure 1.2: A NUMA architecture has asymmetric access speeds to the main memory segments on the chip. CPUs can access local memory efficiently while taking a performance hit to access data on a separate socket [48].

ate the performance characteristics of different scheduling approaches to take advantage of query behaviors.

1.2.1 Parallel Execution and Architecture Awareness

Prior database research has proposed many parallel algorithms for the different operators a database must support. However, the nature of operations for various computations leads to different parallelizability characteristics for each operator [28, 42, 71, 79]. Execution engine implementations can easily partition scan operators to read disparate data segments in parallel, but certain aggregation computations, for example, the median computation, may be more challenging to parallelize. Parallelism characteristics may also depend on the size of datasets, as small dataset sizes may cause the overheads of parallelization to outweigh the approach's benefits. Data synchronization and result coalescing may have non-negligible impacts that become more significant when operating on small data batches.

With modern CPU architectures supporting multiple sockets, each having a separate set of cores and memory hierarchies, data placement-aware operation also becomes more important. This layout, known as a non-uniform memory access (NUMA) architecture, has an increased cost of accessing non-local data as compared to local data. In a NUMA architecture, accessing data from a remote socket takes a latency hit as the request must traverse through the interconnect to the remote socket, retrieve the data, and return this to the requesting core. Figure 1.2 presents a diagram of a two-socket architecture connected via an interconnect, with each socket having independent memory and CPU cores. A well-behaved data placement and access pattern may avoid data access misses and interconnect traffic. All else equal, a system with an improvement in the number of remote accesses is expected to have higher overall performance [28, 54, 68, 71].

1.2.2 Queueing for Databases

Queueing theory presents a variety of solutions to analyze and solve scheduling problems in computer systems. Prior results from the theoretical community have optimized for metrics like mean latency or satisfied fairness considerations in a variety of settings [11, 70].

The database setting provides interesting insights and challenges where scheduling according to system knowledge potentially realizes significant performance improvements beyond prior system optimizations. For the theory community, the combination of sensitivity to performance results, interesting job parallelizability characteristics, and insights into job runtimes make databases an intriguing system of study to apply and develop queueing policies [32].

From a parallelizability perspective, Amdahl's law generally asserts that speedup is not perfect when work cannot be completely parallelized [46]. This behavior leads to diminishing returns for increased resource allocations of parallel tasks. Restrictions on performance gains from parallelization implies fine-grained system resource allocation can avoid resource wastage, which leads to our consideration of overall resource efficiency.

The ability to further determine job sizes or provide job size estimations increases the information scheduling components can work with. Certain applications will repeatedly run the same or similar queries, which allows for measurement and estimation to determine their runtime characteristics. A significant amount of research has otherwise focused on improving the capability of database system components to improve query size estimations without significant query knowledge [27, 55, 58, 59, 60].

1.3 Contributions

This thesis focuses on effectively implementing and evaluating scheduling policies in the NoisePage execution engine [61, 62]. We conduct a survey of scheduling policies applied in real-world DBMSs to establish the basis of prior work in this area. We then present new scheduling approaches to achieve performance improvements over some of the most commonly applied policies in databases. These techniques focus on resource efficiency by combining awareness of query execution behavior and execution hardware. We describe the details of recent theoretical work in this area and implement a series of updates to the system component to manage data layout imbalances and enable NUMA-aware query execution.

In Chapter 2, we provide background on our developments by describing salient features of NoisePage's execution engine and the scheduling approaches taken by various open-source and commercial databases. In Chapter 3, we present changes to NoisePage's query evaluation framework, newly proposed scheduling approaches, and architecture-aware advancements to address remaining performance overheads of query execution. In Chapter 4, we present an evaluation of scheduling algorithms applied to our system and demonstrate the benefits of our developments. We finally present related and future work in Chapter 5 and Chapter 6, respectively, and conclude the consideration of our subject of study in Chapter 7.

Chapter 2

Background

External architectural trends have made large in-memory databases a reality. As we claimed in Section 1.1, an in-memory DBMS shifts the bottlenecks of query processing. The most significant of these changes is eliminating the overhead of paging. When the working dataset of a DBMS fits in memory, there is no I/O stall time from accessing data. In this environment, improving the performance of query execution creates a measurable impact on the system at large. Research has correspondingly placed a greater emphasis on the optimizations necessary for high-performance query execution.

The execution engine is the key processing component in a DBMS. Once other internal components parse and transform a query into a query plan, an execution engine executes the operations specified by the plan to generate output. A great deal of work has resulted in performant algorithms for operations and processing models. We focus our developments on strengthening the complete execution pipeline, considering scheduling an area of potential improvement.

We begin this chapter by introducing the system we build upon, NoisePage, in Section 2.1. Our description serves to contextualize the state-of-the-art in optimized execution engines. We particularly highlight the research developments in query execution that enable the system's effective performance characteristics.

Section 2.2 introduces the intuition behind several scheduling techniques commonly appearing in the queueing theory literature. We emphasize the benefits and drawbacks of some of the most popular approaches implemented in real-world systems.

We then provide a survey of scheduling policies adopted in DBMSs in Section 2.3. This investigation provides a better understanding of the capabilities and scope of work of existing systems. Of the systems we considered, including many open-source and commercial databases, we find the vast majority prefer FCFS policy variants, with a few implementing other traditional policy variants. Our findings motivate further development in considering alternative scheduling approaches.

```sql
SELECT d_year,
       s_nation,
       p_category,
       SUM(lo_revenue - lo_supplycost) AS profit
FROM   date,
       customer,
       supplier,
       part,
       lineorder
WHERE  lo_custkey = c_custkey
       AND lo_suppkey = s_suppkey
       AND lo_partkey = p_partkey
       AND lo_orderdate = d_datekey
       AND c_region = 'AMERICA'
       AND s_region = 'AMERICA'
       AND ( d_year = 1997 OR d_year = 1998 )
       AND ( p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2' )
GROUP  BY d_year,
          s_nation,
          p_category
ORDER  BY d_year,
          s_nation,
          p_category;
```

Listing 1: This snippet depicts Query 4.2 from the Star Schema Benchmark, a benchmark commonly used to evaluate the performance of analytical databases [65].

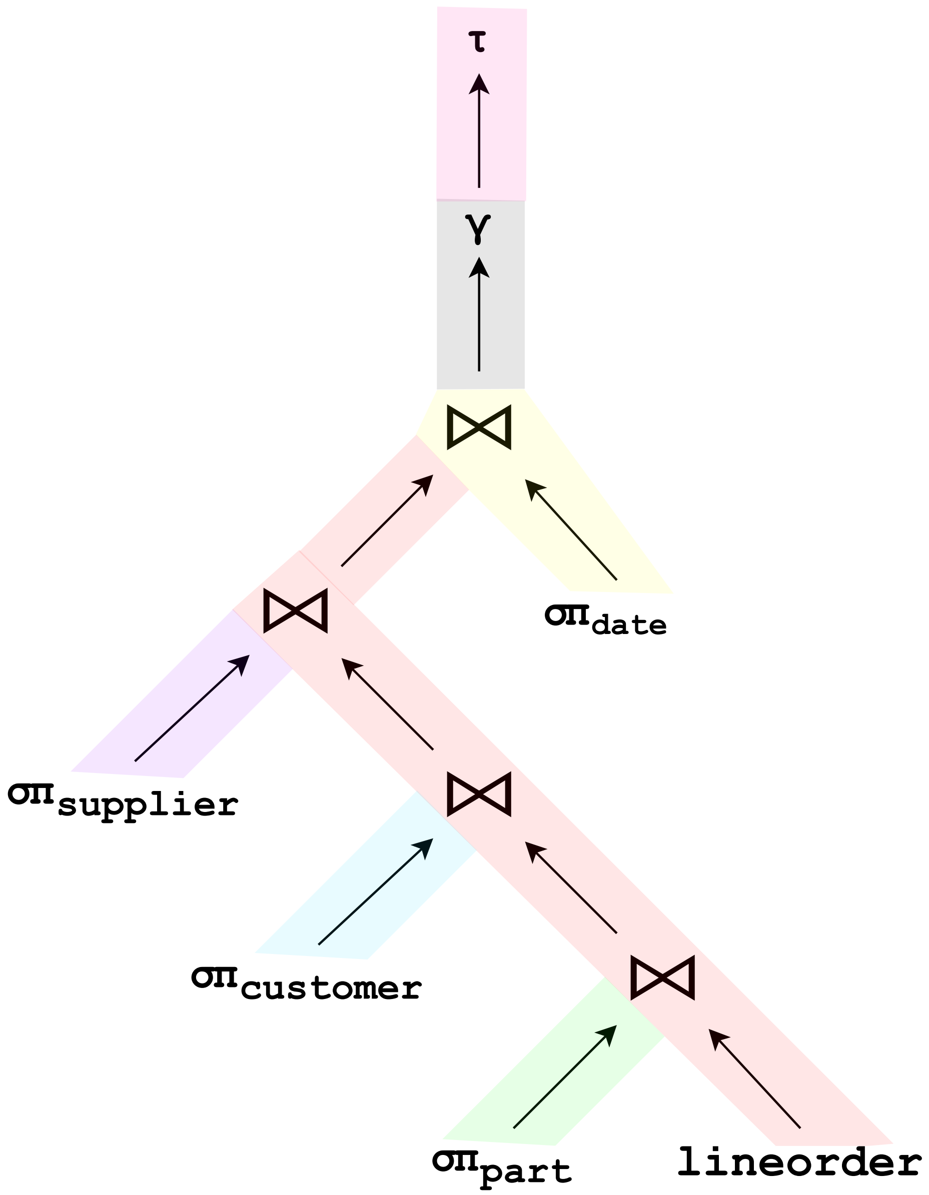


Figure 2.1: This diagram is a simplified query plan corresponding to Query 4.2 in Listing 1. In particular, we condense the representation filter and projection operators in the query plan and do not directly specify query parameters. Each individual color represents a pipeline of execution, the sequence of operations that can be run together. The role compilation plays in extending pipelines is more visible in this query plan, for example, in scanning through **`lineorder`** to join against hash tables constructed on the **`supplier`**, **`customer`**, and **`part`** tables before feeding into another join with the **`date`** table [48].

```
struct P5_State {
    join_hash_table5: JoinHashTable
    filter_manager1: FilterManager
}

fun Query0_Pipeline5_FilterClause(vp: *VectorProjection, tids:
 ↪  *TupleIdList, context: *uint8) -> nil {
    @filterEq(vp, 5, @stringToSql("AMERICA"), tids)
    return
}

fun Query0_Pipeline5_ParallelWork(q_state: *QueryState, p_state:
 ↪  *P5_State, tvi1: *TableVectorIterator) -> nil {
    for (@tableIterAdvance(tvi1)) {
        var vpi1 = @tableIterGetVPI(tvi1)
        @filterManagerRunFilters(&p_state.filter_manager1, vpi1)
        for (; @vpiHasNext(vpi1); @vpiAdvance(vpi1)) {
            var hash_val = @hash(@vpiGetInt(vpi1, 0))
            var row2 = @ptrCast(*BuildRow_Compact2,
             ↪  @joinHTInsert(&p_state.join_hash_table5,
             ↪  hash_val))
            @csWriteInteger(&row2.member0, BitCast(&row2.nulls),
             ↪  0, @vpiGetInt(vpi1, 0))
        }
    }
    return
}
fun Query0_Pipeline5_pre(q_state: *QueryState) -> nil {
    @iterateTableParallel("ssbm.customer", q_state,
     ↪  @execCtxGetTLS(q_state.exec_ctx),
     ↪  Query0_Pipeline5_ParallelWork)
    return
}

fun Query0_Pipeline5_post(q_state: *QueryState) -> nil {
    @joinHTBuildParallel(&q_state.join_hash_table1,
     ↪  @execCtxGetTLS(q_state.exec_ctx), @offsetOf(P5_State,
     ↪  join_hash_table5), q_state)
    return
}
```

Listing 2: This snippet depicts the TPL generated to construct the join hash table on the **customer** table from the query plan in Figure 2.1. **customer** is scanned with a filter only selecting tuples with a region of 'AMERICA.' These tuples are then directly hashed and inserted into the hash table constructed for the subsequent join with **lineorder**. The source code corresponding to this generated IR is then compiled via an LLVM backend into machine code.

2.1 The NoisePage Execution Engine: Vectorized and JIT-Compiled

NoisePage combines some of the most influential research ideas in query performance into a single system: pipelining, compilation, and vectorization. Prior work establishes NoisePage's execution engine as competitive compared against other state-of-the-art in-memory DBMSs when measuring single query execution time [62]. We choose to work on such a highly performant system to show the opportunity for further gains as incremental improvements in single-query performance grow limited [49, 61, 71].

NoisePage implements a query compilation framework inspired by developments in the HyPer DBMS [63]. The query plan operators define the composing units of work and their dependencies, which are organized into pipelines and then compiled into an internal, domain-specific language we name TPL.

NoisePage fuses the sequence of operators within a pipeline into an encompassing operation kernel without requiring the materialization of intermediate results. These sequences end at pipeline breakers that force the materialization of tuples before further computation; the most recognizable pipeline breakers are **`SORT`** and **`HASH JOIN`** operators.

Listing 1, Figure 2.1, and Listing 2 together present a sample of the transformation from a query to the system's intermediate representation. Listing 1 provides a sample query from a commonly evaluated analytical benchmark. This query transforms into the internal query plan visualized in Figure 2.1, highlighting the fusion of operators into pipelines. Listing 2 depicts a sample of the imperative intermediate representation generated by the engine.

The imperative intermediate representation generated from the query plan can then be executed in one of three forms:

- An interpreted mode where an internal VM processes bytecodes corresponding to the operations.
- A JIT-compiled mode in which the intermediate representation is recompiled to machine code by an LLVM backend.
- An adaptive mode where a potentially time-consuming compilation happens in the background while interpreting the query. Once the compiled function implementations are ready, they are swapped in.

This document focuses on the JIT-compiled approach, which maximizes the raw execution performance when excluding compilation time. In our example, NoisePage would compile the generated TPL code shown in Listing 2 into machine code.

NoisePage also integrates a vectorized processing model on top of the compiled engine by introducing staging points within pipelines. Intermediate results may be materialized at these staging points to enable vectorization and SIMD on batches of tuples, increasing performance compared to the tuple-at-a-time processing typical in compiled engines.

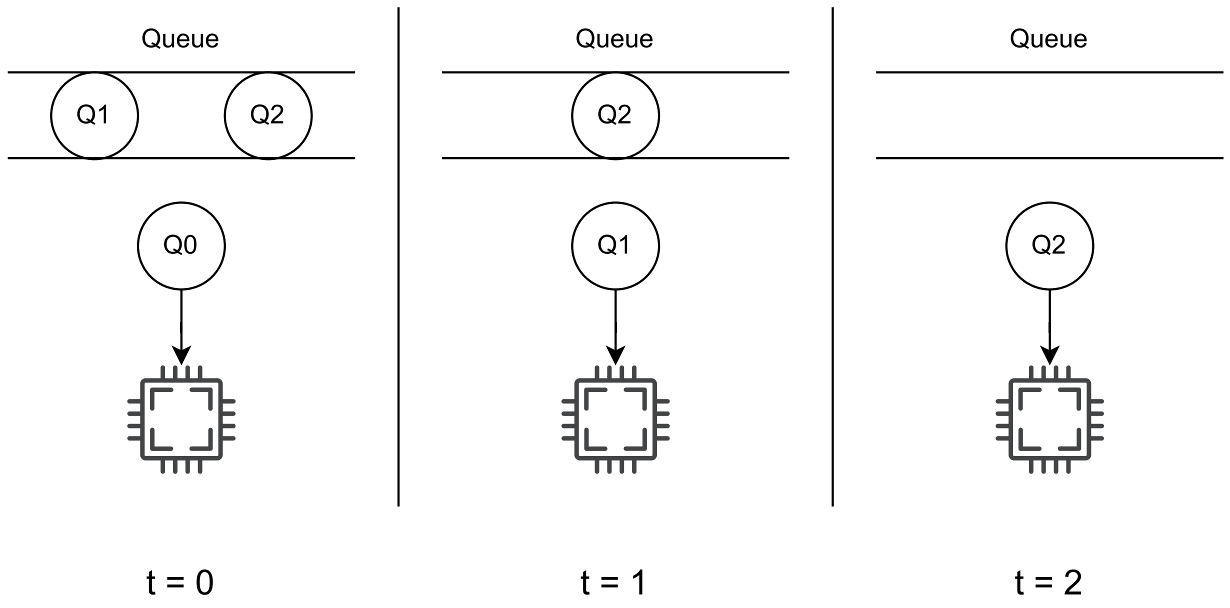


Figure 2.2: An FCFS policy serves jobs in the order they appear with the complete resources of the system. In this diagram, the system serves Q0 exclusively while Q1 and Q2 wait in a queue. The query having control of the system points to the core [48].

2.2 Queueing Theory

Although there is limited prior work on applying scheduling approaches to databases, the queueing theory literature provides many scheduling policies to improve system performance metrics including latency and fairness. This section describes the most common scheduling policies implemented in databases. For simplicity, we introduce scheduling policies in the context of a single-core system serving arbitrary compute tasks, or jobs. In the context of databases, jobs correspond to the smallest unit of work assignable to a scheduler. In some systems, this may be a complete query. In others, they may be smaller fragments such as operators or sequences of operators. We provide a survey of approaches taken in databases in Section 2.3 and describe our implementation of many of these policies in NoisePage in Section 3.2.

2.2.1 First-Come First-Served

Often, the most straightforward scheduling policy to implement is first-come-first-served (FCFS) due to minimal engineering and processing overhead. An FCFS policy serves jobs in the order of arrival [45]. A simple FCFS scheduling approach would allocate all of the processing system's resources to one job at a time, as demonstrated in Figure 2.2. Unfortunately, the FCFS policy may cause extensive queueing time for any jobs arriving after a long-running one [45].

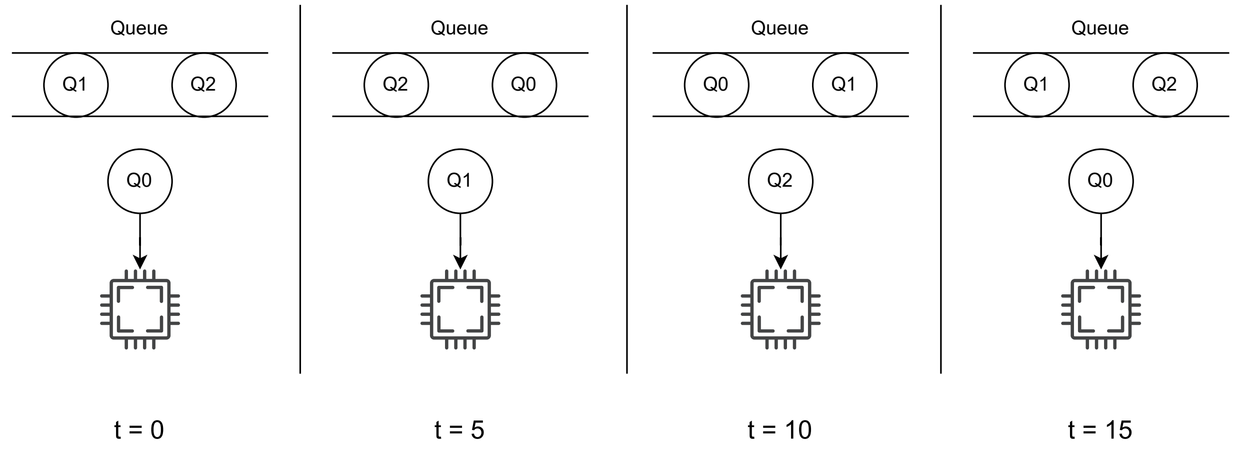


Figure 2.3: An EQUI policy divides system resources among jobs active in the system. A PS policy will instead time-share complete control of system resources when dynamic resource division is challenging or not possible. This diagram shows a PS policy applied to a system serving Q0, Q1, and Q2 concurrently where each query receives exclusive control for a 5s quantum. The query having control of the system in each quantum points to the core [48].

2.2.2 Equitable Division

Another common approach is to equally divide system resources among all jobs running in the system at any given moment. We will refer to this policy as EQUI [33]. When dividing resources among jobs is not possible, the system may look more like a time-sharing system, a policy known as processor sharing (PS). Under a PS policy, each job has complete control of system resources for a short "quantum" of time before letting the next job run, demonstrated in Figure 2.3 [45]. The procedure will continue in a round-robin fashion until all jobs complete. Although these policies are different in resource allocation behavior, they both prioritize fairness in the system. However, optimizing for fairness does not always correspond with optimizing for performance considerations.

2.2.3 Shortest Remaining Processing Time

Although the aforementioned approaches are independent of job sizes, the Shortest Remaining Processing Time (SRPT) scheduling policy attempts to reduce queueing time for shorter jobs by prioritizing their execution [45]. This policy will always prioritize running jobs with the least time left to run, reducing their latency at the cost of other jobs in the system. Queueing theory tells us that SRPT generally reduces average latency [45, 70]. This policy, however, can be challenging to implement due to significant additional instrumentation and estimation, limiting its usage.

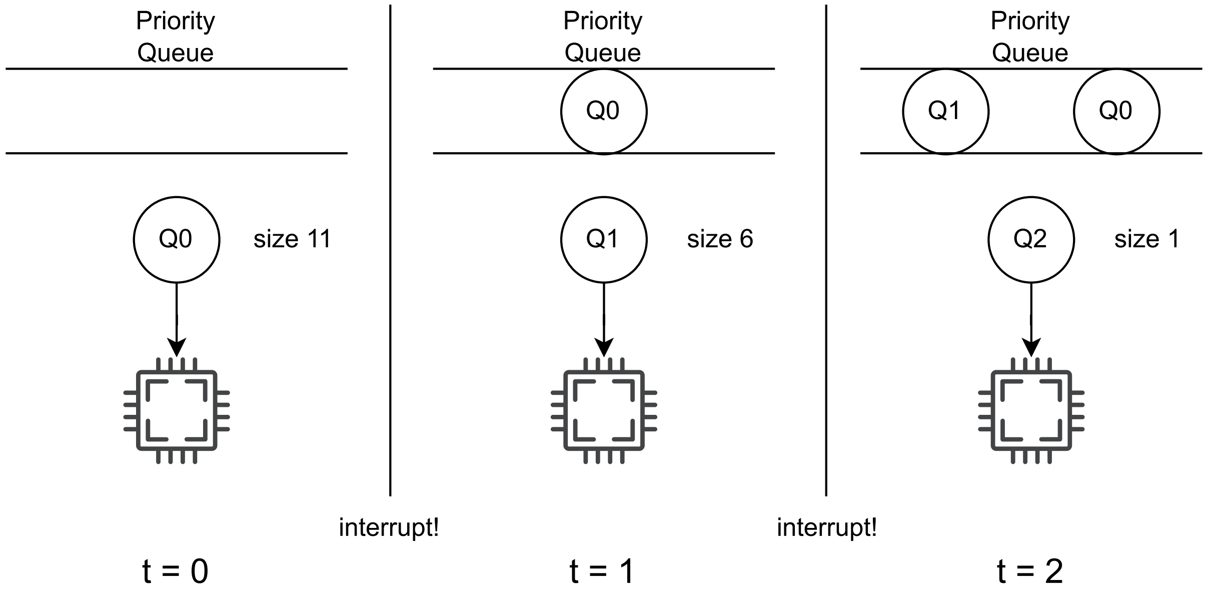


Figure 2.4: An SRPT policy prioritizes jobs with minimal work left. This policy may interrupt previously executing jobs for new, shorter jobs. This policy demonstrates that Q0 was interrupted by a smaller Q1, which was subsequently interrupted by an even smaller Q2. The query having control of the system points to the core [48].

2.2.4 Comparison

To understand the merits of each of these policies, we examine a brief example. We first introduce a few variables corresponding to concrete attributes of these policies:

$$T_S\text{: the time spent executing a job} \tag{2.1}$$

$$T_Q\text{: the time a job spends queueing} \tag{2.2}$$

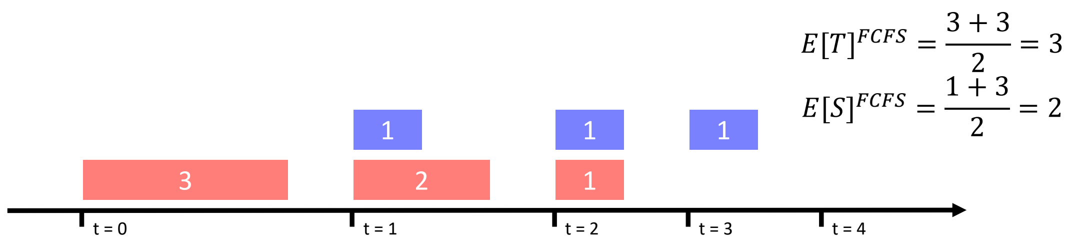
$$T = T_S + T_Q\text{: the total response time of a job} \tag{2.3}$$

$$S = \frac{T}{T_S}\text{: the slowdown of a job} \tag{2.4}$$

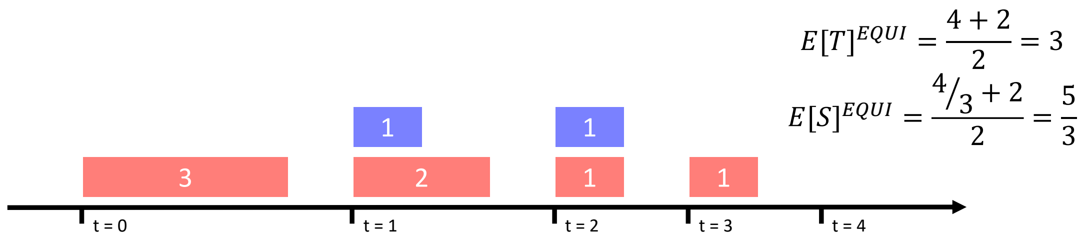
To compare these policies in one specific scenario, consider a system that receives two jobs. The first job has a runtime (or size) of 3 units and arrives at time 0. Another, which has a size of 1 unit, arrives at time 1. Figure 2.5 demonstrates the system's state at each time point under the various scheduling policies.

FCFS queueing would force the second, smaller job to wait until the larger job has finished resulting in an overall $E[T] = 3$ and $E[S] = 2$.

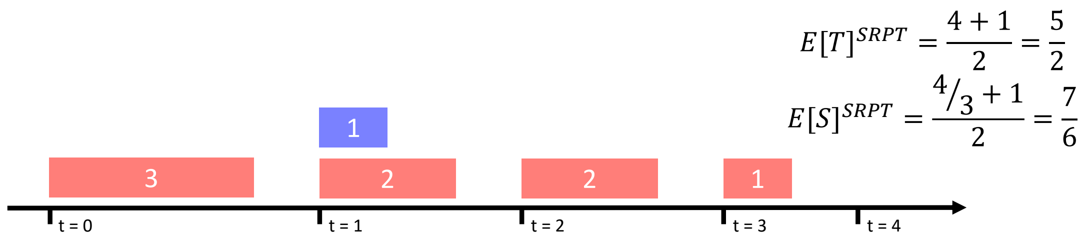
In comparison, consider a PS policy where the quantum size is one unit, and newly arriving jobs will only start processing after all previously existing jobs receive a time slice. Such a policy allows the shorter job to complete before the larger job does, reducing the overall slowdown, with $E[T] = 3$ and $E[S] = \frac{5}{3}$.


$E[T]^{FCFS} = \frac{3+3}{2} = 3$
$E[S]^{FCFS} = \frac{1+3}{2} = 2$
3
2
1
1
1
t = 0
t = 1
t = 2
t = 3
t = 4

(a) An FCFS policy will execute jobs in the order of arrival, so the bottom job will complete before the top job starts.


$E[T]^{EQUI} = \frac{4+2}{2} = 3$
$E[S]^{EQUI} = \frac{4/3+2}{2} = \frac{5}{3}$
3
2
1
1
1
1
t = 0
t = 1
t = 2
t = 3
t = 4

(b) A PS policy will allow for jobs to time-share system resources. In this example, we use a quantum size of one time unit with jobs sharing resources in the order of their arrival. The top job will be able to complete before the bottom job does.


$E[T]^{SRPT} = \frac{4+1}{2} = \frac{5}{2}$
$E[S]^{SRPT} = \frac{4/3+1}{2} = \frac{7}{6}$
3
1
2
2
1
t = 0
t = 1
t = 2
t = 3
t = 4

(c) An SRPT policy will prioritize jobs with shorter remaining runtimes so the top job will be prioritized and run before the bottom job on arrival.

Figure 2.5: A comparison of different scheduling policies in a simple system where one job of size 3 arrives at time 0, and the second job of size 1 arrives at time 1.

Finally, SRPT, by prioritizing shorter jobs at each time step, cuts down on the total response time of the system and further decreases the average slowdown of the jobs. Both metrics improve, with $E[T] = \frac{5}{2}$ and $E[S] = \frac{7}{6}$.

Though this demonstration is a tailored example, the comparison shows the potential benefits of policy changes and provides intuition behind why systems typically choose EQUI/PS or SRPT as a "next step."

2.3 Policies and Parallelism in Database Scheduling

To better understand the state-of-the-art of scheduling policies in DBMSs, we surveyed various open-source, commercial, and research-oriented databases. For each system, we outline the capa-

Table 2.1: This table summarizes the scheduling policies implemented of various open-source, commercial, and research-oriented databases.

FCFS	PS	SRPT
PostgreSQL	Databricks	Umbra
MySQL	SQL Server	Amazon Redshift
ClickHouse	ClickHouse (background)	
IBM Db2		
Oracle Database		
Snowflake		
HyPer		

bilities of systems to modify query execution orders and other unique implementation attributes of their query execution procedures. We utilized this analysis to better scope the potential for developments in this subdomain of query execution. For the remainder of this section, mentions of parallelism refer to intra-query parallelism, the parallelization of individual operators. Most databases support inter-query parallelism, the concurrent execution of multiple independent queries.

Our analysis shows a clear preference among the systems we considered for implementing more straightforward scheduling policies, particularly variations of FCFS scheduling. Only a few systems have adopted other approaches in supporting EQUI-like and SRPT-like policies. We present the systems considered that fall under each of these categories. Table 2.1 summarizes the findings of this section.

2.3.1 First-Come First-Served

Some of the most popular DBMSs today utilize FCFS scheduling. Among open-source systems, both older systems developed since the 1990s and newer systems utilize variants of FCFS scheduling. Several commercial, closed-source systems describe their scheduling policies as variants of FCFS as well.

MySQL implements a variation of FCFS. The DBMS supports two priority levels that the user can specify in queries [2, 3, 4]. Unlike most other databases, MySQL does not support executing a single query in parallel, so each query is limited to a single thread [4].

PostgreSQL (Postgres) is similarly a well-known, older open-source DBMS. Postgres executes queries in the order of arrival but does not natively support query priorities during execution. Unlike most other systems supporting parallel execution, Postgres uses processes instead of threads [7]. This parallelism approach does make the use of alternative scheduling approaches slightly more challenging, as switching processes is a heavier-weight procedure at the OS-level than switching threads of the same process, limiting the flexibility of scheduling procedures [74].

ClickHouse is a relatively new open-source DBMS developed to support fast analytical applications. This system primarily executes queries in FCFS order but can support query priorities with any integer value [1, 38].

IBM Db2 is one of the first commercial databases: the system was first released in the 1980s but is still actively developed. Db2's most recent release executes queries in FCFS order but allows granting priority levels on a per-session basis [14, 15, 16].

Oracle Database is another actively-developed, early commercial DBMS from the late 1970s. The database's scheduling policy defaults to FCFS, but the system does expose knobs for greater user control, among them queue timeouts, query priorities, and probabilistic execution, where queries are consumed from queues with a user-defined distribution [6]. Oracle's parallel execution framework determines per-query parallelization based on the amount of data to process and the degree of parallelism available on the system at runtime. Internally, the system assigns ranges based on the size of the datasets consumed and maximum parallelism level configured [5].

Snowflake is a data warehouse built for the cloud-computing era that focuses on supporting analytical workloads. Snowflake's execution model serves queries in FCFS order [12]. As a user can provision compute cluster resources independently for each workload, there is no internal concept of priority at the workload or query level [10].

HyPer was a research system built at the Technical University of Munich and subsequently acquired by Tableau. The initial version of HyPer executed queries in FCFS order and did not support concepts of query priority. HyPer, however, did take a unique approach to query parallelism. The system executes query pipelines on small data divisions in a locality-aware fashion. The system preferentially assigns operations and corresponding data segments to cores on the same NUMA node as the segment's location in memory. This approach, named "morsel-driven" parallelism, improves performance [54].

Among the systems mentioned above, many support user-specified query prioritization. This feature allows users to manually take a fine-grained approach to scheduling which specific queries or query groups have higher importance than others. Although varying resource priorities provides interesting resource-performance tradeoffs and a helpful customer feature, we assume that all queries have equal priority in our development. Doing so simplifies the resource and target modeling constraints of our solutions.

2.3.2 Equitable Division

Databricks is a cloud data platform supporting warehousing and analytical applications. The scheduling policy for tasks running on the Apache Spark scheduler in Databricks enforces approximate processor sharing for all queries running in the system. The scheduler periodically preempts query operations that exceed their runtime timeouts, with parameters for the policy made available for user management [13]. The open-source version of Apache Spark defaults to a scheduling policy of FCFS prioritization, with similar support for fair sharing [8].

Microsoft has developed its DBMS offering, SQL Server, since the late 1980s. SQL Server includes an internal layer, named SQLOS, to manage many resources typically handled by the operating system a database runs on, including scheduling [41]. This system enables SQL Server to take a cooperative, time-sharing scheduling approach for query execution. In SQL Server's scheduler framework, query tasks run for up to a quantum of 4ms, subsequently yielding to other tasks in the system [21, 41].

Interestingly, some aspects of ClickHouse's execution pipeline support a variant of PS: certain background storage layer operations break up each "task" into executable "steps" that execute in a round-robin fashion across all tasks. However, as the steps are not necessarily of equal size, the algorithm presented does not quite represent equitable sharing of resources. That said, the presented algorithm does expect tasks with shorter step sequences to complete before those with longer step sequences [1, 38].

2.3.3 Shortest Remaining Processing Time

Umbra is the successor to HyPer, under development at the Technical University of Munich [9]. Umbra's parallel execution framework is similar to HyPer's in using a morsel-based parallelism approach. However, Umbra augments this framework with an internal scheduling policy. Umbra uses an optimized approach to a technique known as stride scheduling, defining the query priorities in this algorithm inversely proportional to the expected query size. In particular, query priorities decay the longer queries take, with self-tuned or user-defined hyperparameters guiding the re-prioritization process [77]. This approach matches the goals of prioritizing queries expected to have shorter remaining runtimes.

Amazon Redshift is a cloud data warehouse built to support analytical workloads. Redshift supports an SRPT-like query scheduling framework with a feature named Short Query Acceleration. This technique prioritizes queries with an expected runtime less than a dynamically-determined threshold, where the expected runtime is calculated with internal statistics and predictive models [19, 27]. The system additionally exposes the acceleration threshold and query prioritization classes for user specification [18, 20].

Chapter 3

Methodology

In this chapter, we focus on the technical details of supporting new scheduling policies in NoiseP-age. Our work to better understand query parallelization behaviors led to the development of new scheduling policies in queueing theory. These policies interleave the execution order of pipelines across queries to enable greater workload-level efficiency. To enable these query behavior-aware scheduling techniques, we present the design and implementation of a scheduling component built into NoisePage.

We also describe techniques to add resource awareness to the system, focusing on NUMA architecture awareness. NUMA awareness is an established performance solution in the database research literature [54, 68]. However, the NoisePage system was not previously resource-aware. We detail the benefits of our combined NUMA architecture-aware scheduling approaches in further improving scheduling efficiency and system performance.

3.1 Query Parallelism and Scheduling

Even in an analytical workload consuming large amounts of data, queries are not perfectly parallelizable. Considering the parallelization characteristics of the composing pipeline operations largely explains query-level parallelizability. Regardless of the processing mode of the system, different database operations typically have different execution properties. Some operations, such as sequentially scanning through a large table with an associated filter, can easily be highly parallelized. Others, such as median aggregations, are quite challenging to implement in a parallel manner. Some operations fall somewhere in between, like many join algorithm implementations. For example, in a parallel hash-join algorithm, the probe phase may require minimal coordination, with each thread independently searching the hash table for a portion of the input. However, the build or partition phases often involve more coordination to finalize the hash table, which limits parallelization. Frequently, many operations consume small datasets which do not require the degree of parallelism offered by the system architecture. Borrowing terms from queueing theory, we refer to parallel work as elastic and non-parallelizable work as inelastic.

Figure 3.1 demonstrates the speedups observed for various queries from the Star Schema Benchmark as implemented in our system, none of which have perfect parallelism characteristics [32]. Our experiment measured query execution times in NoisePage while scaling the maximum

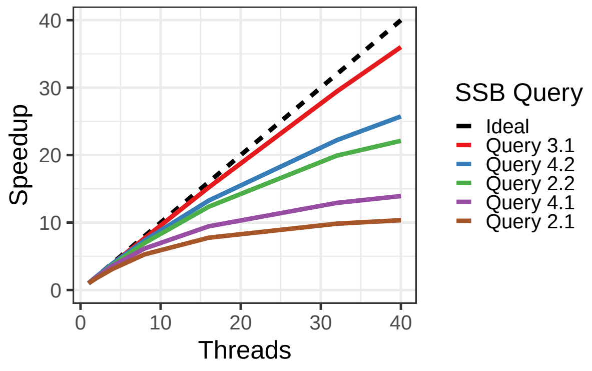


Figure 3.1: Queries in the Star Schema Benchmark are not perfectly parallelizable. A query's runtime does not scale linearly with the number of threads given to the query [32].

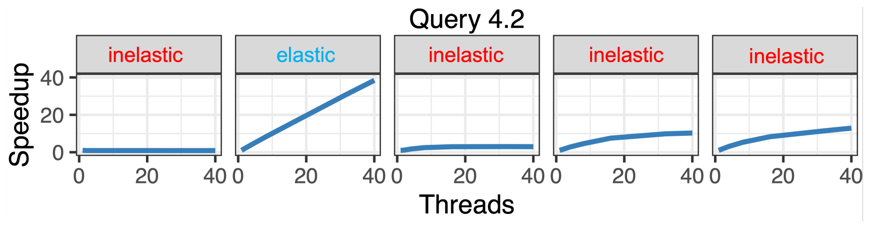


Figure 3.2: Of Query 4.2's sequence of operator pipelines, many are inelastic while some are elastic [34].

allowed parallelism of all operations in the system from 1 to 40. Though we present the results for NoisePage here, we found that other open-source and commercial systems have similar query speedup characteristics.

We instrumented NoisePage to inspect pipeline parallelization further, recording runtimes for each individual pipeline to analyze offline. Some pipelines exhibit inelastic behaviors due to the algorithms that implement their operators or because of limited dataset sizes. As an example, we depict the parallelizability characteristics of pipelines from Query 4.2 in Figure 3.2. These pipelines vary in behavior; some are highly elastic, some partially elastic, and some completely inelastic.

NoisePage's execution engine was primarily built to show single-query performance gains. The system executes pipelines generated by the compilation engine as a series of steps. Each pipeline consists of an initialization routine, often setting up state, a core logic routine, fulfilling the operation, and a teardown routine, releasing any unneeded state. Parallel execution under this framework takes the approach of fork-join parallelism on certain operations, including parts of table scans, aggregations, joins, and sorts. The system internally utilizes data parallelism in this setting by relying on an external module to partition data containers consisting of table data or intermediate results and invoke lambdas on these partitions independently. This approach was first implemented using Intel's Thread Building Blocks [22].

In placing a greater emphasis on fine-grained control of the execution pipeline with scheduling, we make two major changes: we break the original three-routine pipelines into a longer

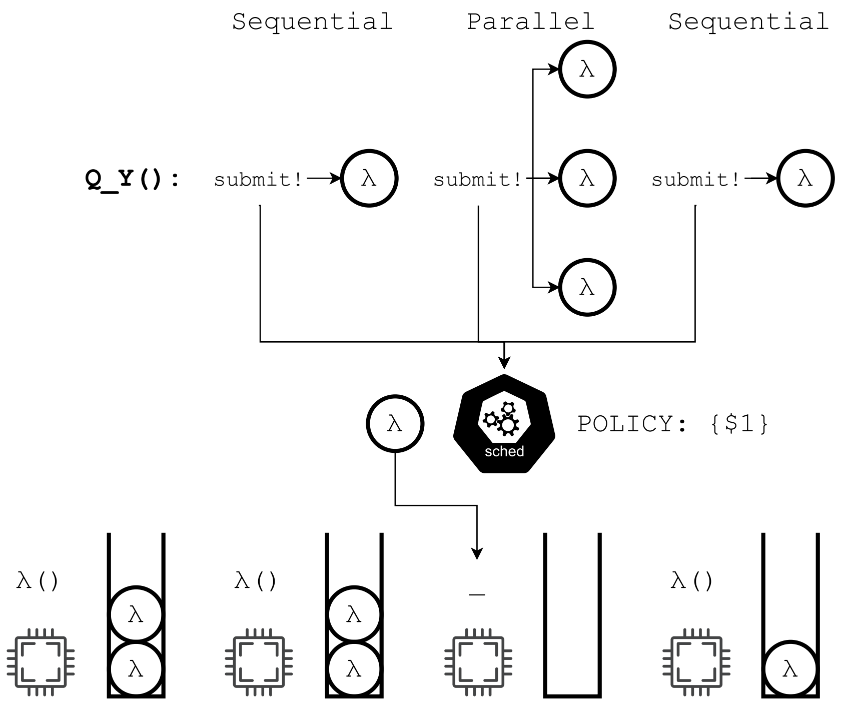


Figure 3.3: Our scheduling architecture adds staging points where parallelism levels change during which the scheduler finds the right resources for execution. Scheduler submission typically happens at the start of a query, right before the query executes a parallel step, and when continuing after a parallel step. The scheduler will assign compute resources to the tasks it receives based on the policy defined [48].

sequence of functions and directly manage all scheduling decisions in the runtime.

The former modification involves splitting the core logic of the pipeline into multiple code blocks, separating parallel components from any operator post-processing. Though this particular change may affect potential performance gains from compiler optimizations by limiting the scope of analysis provided to the compiler, doing so allows us greater introspection into the performance characteristics of each subsequence of operations. These characteristics are then maintained as additional query metadata and used in scheduling decisions.

Our latter change consists of building an internal scheduler to transform the query-level fork-join parallelism into more precise task parallelism. Previously, parallel operations submitted a data container to the external scheduling module and reserved the system's complete compute resources. Instead, the new task parallelism model submits computational tasks and associated data partitions to the scheduler, which the scheduler then manages. The scheduler can thus maintain a global view of running tasks across all active queries and dynamically allocate work to individual resources. The final subtask of a parallel component spawns a new successor task to the scheduler on completion to continue processing remaining query operations. Figure 3.3 provides a visual representation of the aforementioned scheduling architecture.

3.2 Scheduling Policies

We considered a variety of scheduling algorithms to implement in NoisePage, including those we previously presented in Section 2.2. We describe the implementations of these policies in NoisePage before introducing the novel Inelastic-First Shortest Remaining Processing Time (IF-SRPT) policy.

3.2.1 Traditional Scheduling Approaches

We add variants of the FCFS and SRPT scheduling policies to NoisePage. All scheduling algorithms use relatively similar architectures, differing mainly in their prioritization schemes, to isolate the performance effects of the policies themselves.

Our FCFS scheduler prioritizes query execution in the order of arrival. On startup, a thread is spawned and pinned to each physical core. Each thread processes its own corresponding min-heap, and submitted tasks join the heap with the shortest length across all cores. This approach enables cores to process approximately equal amounts of work with a coarse, low-compute estimation. The submission process adds tasks to the appropriate heap with a priority value of the start time of the corresponding query. Threads will then select the task with the earliest query start time in their heap, enforcing the FCFS execution order.

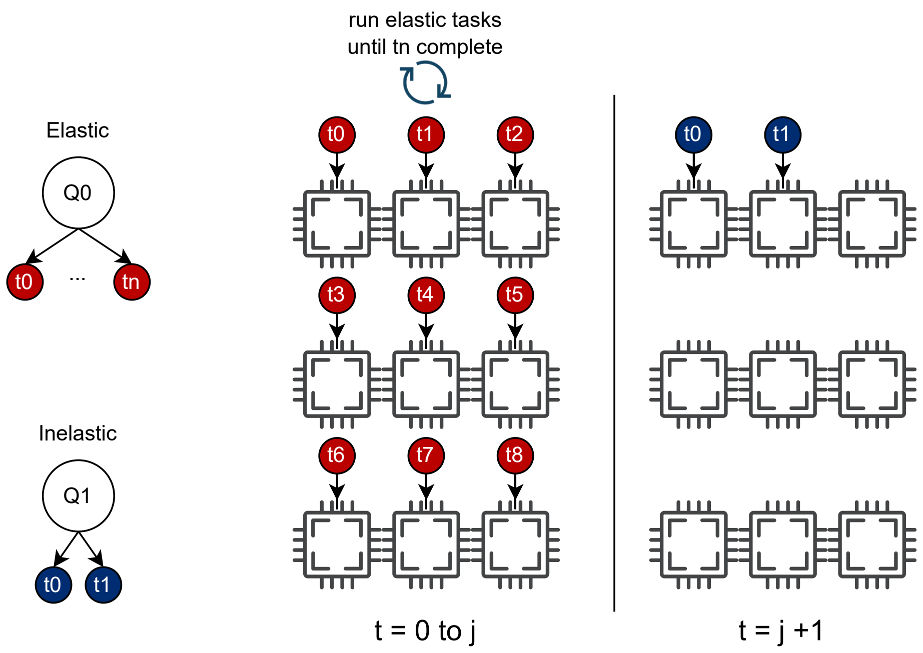
The SRPT scheduler has a similar startup approach, pinning a thread to each core but instead maintains two heaps: one for elastic tasks and another for inelastic tasks. On task submission, tasks will again join the appropriate heap with the least remaining work. However, tasks in this scheme instead have a priority value equal to the estimated remaining runtime of the query. When selecting a new task to process, threads will look at both heaps and select the task with the highest priority, thus preserving the SRPT execution order.

When using the SRPT scheduling algorithm, we determine phase-based estimations of the remaining query runtime with pre-processing. Our pre-processing step consists of running each query in a single-threaded fashion and recording the execution times for each pipeline. During experiments, these measurements can then be reused in parallel environment to estimate the amount of work left. Doing so allows us to maintain a relative ordering of queries based on size without adding significant statistics collection overhead during query processing.

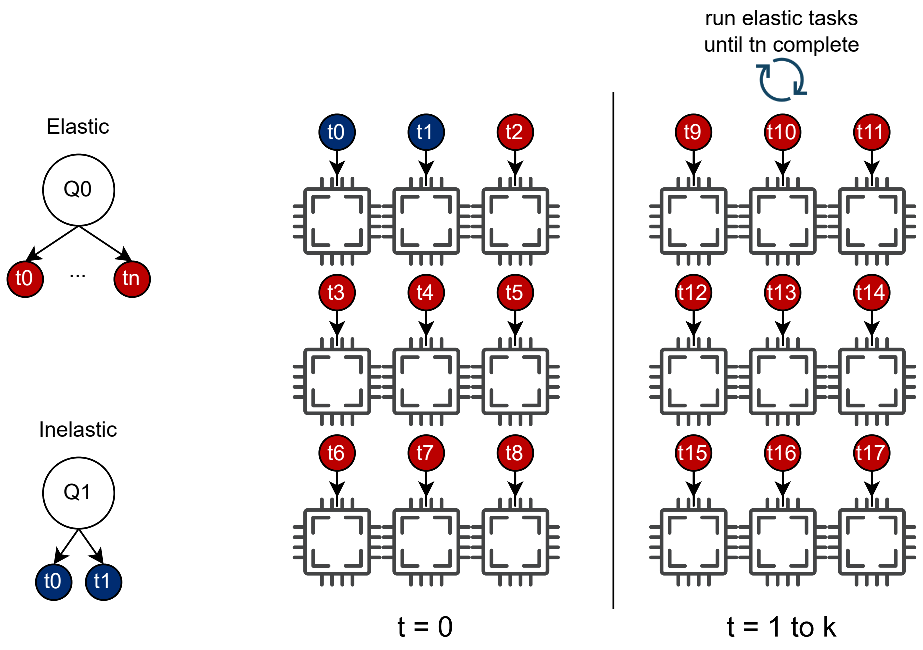
We do not currently implement the EQUI algorithm. Our data-driven parallelism technique does introduce a natural level of partitioning. However, as implemented currently, our internal scheduling algorithms have no concept of queries, only the queues containing lambda tasks they execute. As such, there is no per-query compute tracking and, thus, no way to enforce fairness across queries. The data-parallel execution framework is otherwise not particularly amenable to equitable time-based execution partitioning, which depends on the flexibility of interruption during execution. An alternative threading architecture, setting up threads per query with runtime-based preemption, may be more amenable to such an implementation.

3.2.2 Inelastic-First Shortest Remaining Processing Time

We next present the insights guiding a new scheduling approach first implemented in NoisePage as part of a related line of work in developing query-characteristic aware scheduling policies.



(a) A low-efficiency transition leaves cores unused. Running the elastic job first leaves resources under-utilized while running the inelastic job.



(b) A high-efficiency transition will keep the system utilized. Deferring the elastic job will allow more effective resource utilization.

Figure 3.4: Prioritizing the execution of inelastic work improves the utilization of a system by avoiding low resource efficiency transitions whenever possible.

The Inelastic-First Shortest Remaining Processing Time (IFSRPT) extends the SRPT algorithm to first consider parallelizability [34].

IFSRPT prioritizes inelastic pipeline tasks over elastic pipeline tasks. We previously introduced the benefits of the SRPT scheduling algorithm in Section 2.2. The added benefit of prioritizing inelastic tasks optimizes overall system utilization.

Figure 3.4 demonstrates the argument in favor of deferring parallelizable work. Consider a simplified system processing two jobs. One job is very parallelizable; it can easily use up all the compute resources in the system. The other job is not parallelizable beyond a small degree, a degree far smaller than the system resource limit. If the system were to prioritize the elastic job, many resources would remain idle as the inelastic job runs. On the other hand, if the system prioritizes the inelastic job, the elastic job could easily consume all extra resources and flexibly expand when the inelastic job finishes. The latter scenario optimizes to maintain system high utilization system by deferring elastic work whenever possible. Elastic work is effectively more valuable because it can be processed at any time to increase system utilization in a way that inelastic work cannot.

Our implementation of IFSRPT shares startup and size estimation characteristics with the SRPT algorithm but changes the task selection approach. Instead of choosing tasks across both the elastic and inelastic queues, the processing threads always exhausts the inelastic queue first. This procedure prioritizes the inelastic phases of queries and defers parallelizable work.

We also implement one additional scheduling algorithm that chooses between the IFSRPT and SRPT policies based on the load exhibited in the system. There may be no benefit to parallel work at very high load points if the system has enough inelastic work to utilize all resources fully. In these situations, IFSRPT may not increase system efficiency. A thresholding approach, which we call THRESHOLD, instead allows the scheduler to choose between these two algorithms based on the observed system load, which we estimate with the number of jobs in the inelastic queue. Each thread will use the SRPT algorithm when an excessive number of jobs are in the inelastic queue and otherwise use the IFSRPT algorithm.

3.3 Architecture Aware Execution

The central theme of our research contributions focuses on improving system efficiency to increase performance. One aspect of efficiency we look to improve upon in this system is an awareness of data placement.

NUMA systems resemble a mini-distributed system in that remote operations are less performant than local operations. The HyPer DBMS previously highlighted the effects of using a data-parallel execution method with NUMA locality. NUMA awareness improved system performance, obtaining significant speedups on their multi-socket benchmarking architecture [54].

In considering NUMA architectures, we took two performance aspects characteristics into account in developing our system:

1. Poor data layouts cause inefficiencies from access pattern skew [68]. Heavy access to remote data may trigger interconnect bandwidth bottlenecks or memory bandwidth bottlenecks on the remote node.

2. Any access to remote data is less efficient than local data due to additional incurred latency. Our results, presented later in Chapter 4, shows that NUMA-aware accesses improves cache efficiency.

3.3.1 Data Layout and Representation

The first of our aforementioned concerns suggests that altering the data layout may partially mitigate performance issues. Operating systems often expose control over memory placement with low-level libraries. In Linux, this is done with the **`libnuma`** library, which reports information about the memory layout of the system and allows fine-grained control over memory allocation [76]. NUMA allocations are fulfilled with a granularity of a system page. We modify the table data representation in the DBMS to store a roughly equal segment of data on each NUMA node with a round-robin partitioning strategy, alternating the node each data segment is placed on.

NoisePage previously stored data in a largely columnar fashion, allocating a large chunk of data for each column in a batch of tuples with a predefined batch size. An external container then maintained pointers to the columns for each batch of tuples, representing a data block. We modify the NoisePage storage format to store data in a hybrid fashion similar to PAX, allocating all attributes for a set of tuples onto a single virtual memory page but organizing attributes in a columnar fashion [25]. We additionally maintain a count of the number of tuples in each batch and internal memory management metadata per block.

The PAX hybrid storage model lowered execution overheads of DSM, or columnar storage, while improving cache efficiency over NSM, or row-based storage [25]. In a main-memory database, there is no interaction with persistent storage, so we do not incur the overheads of paging. Our primary motivation for using a PAX-like format was to more naturally align with the NUMA allocation methods, which require page-sized aligned regions. Since multiple columns in a tuple will likely be accessed together in the same task, we wanted them stored in the same NUMA region. A natural approach is to allocate these tuples together as part of the same block. A block size equivalent to a page size would automatically align with the NUMA-aware allocation specification and introduce more flexible partitioning. We present a diagram of our data layout in Figure 3.5.

As we will later present in Section 4.4, we also enabled the usage of hugepages in the DBMS. Operating system hugepages are pages of size larger than the default 4KB. Increasing the size of pages used in the database increases the amount of data that can be allocated together in a single page. Doing so improves the vectorization batch size compared to the default page size on our execution architecture and is expected to reduce the overhead of address translation.

3.3.2 Locality-Aware Scheduling

Our latter NUMA architecture concern additionally requires the use of data awareness during execution. We expect that augmenting operator implementations to execute queries in a locality-optimized manner will improve query performance.

We modified the scheduling approaches introduced in the previous section to additionally be locality aware. From a scheduling perspective, one can model locality-sensitive jobs as having

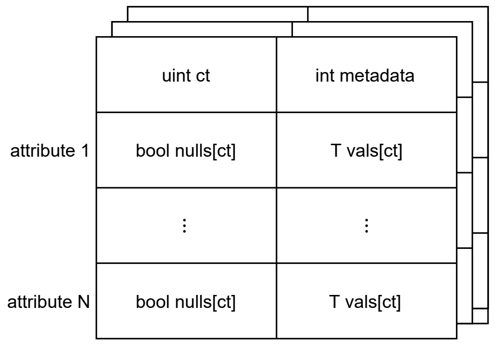


Figure 3.5: Our data layout stores tuples in a hybrid fashion, keeping attribute values contiguous in memory but storing all attributes for a batch of tuples together. In addition to the data itself, we maintain a count of tuples and some additional implementation-specific metadata per block. A table will be composed of multiple such blocks [48].

varying sizes depending on their assignment to system resources. Jobs will be "larger" as they take more time when scheduled onto suboptimal resources.

Implementing locality awareness involved tracking allocation metadata and integrating this information into the execution pipeline to prefer operating on local data. In this work, we implement locality-aware execution only for table scan operations, which compose a significant proportion of query execution. However, we envision augmenting allocations for intermediate materialized tuples or data structures in the future.

We augment the preexisting container of data blocks for each internal table structure with a map from the NUMA region to the blocks allocated in that region. Our updated table scan operations take advantage of this representation by creating executable tasks that only operate on chunks of data allocated within a single NUMA region, which then get submitted to the scheduler. The scheduler, now aware of the optimal NUMA region for execution, assigns each task to a queue on one of the cores corresponding to this region.

We also added a modified version of work-stealing in the scheduling algorithms to reduce the risk of workload imbalance. The scheduler submits query tasks to a core on an alternative socket if the queues for this core are far shorter than the local queue.

Chapter 4

Evaluation

This chapter presents the performance characteristics of the policies and system modifications we implemented in NoisePage as detailed in Chapter 3. We describe the attributes of the analytical benchmark we use in our evaluation, discuss the performance statistics that guided our exploration into system optimizations and development, and demonstrate the performance benefits of improving system efficiency.

Our experiments focus on measuring query execution-related performance changes in a DBMS. As introduced in Section 2.1, the in-memory database NoisePage neither incurs overheads of parsing and optimization nor those of persistent storage during execution. In addition, we do not consider data generation or loading costs in our benchmarks.

We run all our experiments on a dual-socket system with two 20-core Intel Xeon Gold 5218R CPUs (2.10GHz, 27.5 MiB L3 cache, with AVX512) and 187GiB of DRAM, roughly equally split across the two sockets. A remote socket has roughly 2x the distance of a local socket. NUMA regions/nodes in this system correspond directly with the sockets, so we use the terms interchangeably.

Our benchmarks are hand-written C++ programs that submit queries from the Star Schema Benchmark to NoisePage to measure the system's completion rate and the average latency of queries in the system. We additionally use profiles collected with the Linux tool **`perf`** and Intel's Performance Counter Monitor (PCM) to determine architectural bottlenecks and overheads by investigating CPU, memory, caching, and interconnect metrics in our experiments [17, 40, 57].

We conduct statistical testing to ascertain the significance of our results. On all plotted results with error bars aside from the baseline, we use Welch's t-test to determine a statistically significant change in mean values. All statistical tests compare the active experiment against both the baseline and the previous experiment of the same type. All results are statistically significant with a p-value < 0.05 for both tests unless otherwise noted.

Our evaluation demonstrates the value of architecture awareness in combination with query-aware scheduling policies grounded in queueing theory, achieving over a 30% performance improvement over the baseline. The performance experiments in this chapter will build up to this result step-by-step, going through the design decisions of our implementation and their cumulative impact.

4.1 Characteristics of the Star Schema Benchmark

DBMSs support workloads with highly variable characteristics, from short-lived point queries to long-running analytical analyses. This work focuses on determining performance on analytical, or OLAP, workloads. This category of workloads primarily consists of read-heavy queries that consume a significant amount of data and have relatively high scalability.

Our benchmarks utilize queries from the Star Schema Benchmark, a benchmark intended to simulate data warehousing applications [65]. This benchmark consists of 13 queries selecting data across tables from a uniformly generated dataset. We present the database schema in Figure 4.1 and a sample query in Listing 1. Additionally, we previously showed in Section 3.1 that queries from this benchmark are not perfectly parallelizable and that individual pipelines may be elastic or inelastic depending on implementation behaviors and dataset sizes. Our observations indicate an opportunity for performance improvement on this workload.

We use two benchmarks in our analyses to capture salient performance measures. Both benchmarks only use queries from the Star Schema Benchmark, which we configure with a scale factor of 10 ($\sim$10GB of data). As such, the workload dataset fits in memory and does not encounter disk or network-based overheads. For both benchmarks, we evaluate the scheduling policies FCFS, IFSRPT, SRPT, and THRESHOLD, with implementations as described in Chapter 3.

Spin Benchmark The first benchmark measures the maximum system throughput, the overall number of queries the system can process in a given time. To do so, we must overload the system by ensuring that the queue of queries to process is never empty.

We implement this benchmark by creating a closed-loop batch system where 2000 threads continuously send a new query to the system as the previously submitted job from that thread completes. To minimize interactions with query processing, we pin all task submission and benchmark measurement threads to a single core on the system. We record the number of completions at each second, which we then process to determine a measure of system throughput.

We set up this first benchmark to run for a total of 30 minutes per experiment to allow the system to stabilize before we capture metrics, which we empirically observe completes well within this period across all our experiments. We inspect the moving average of completions per second to determine the stability range of each experiment. Over this period, we use the average completion rate of the system as a measure of the max throughput.

Some of the scheduling policies we support prioritize shorter queries, which may inflate the reported throughput in this benchmark by increasing the number of short queries considered. We considered an alternative benchmark design to measure maximum throughput: pre-generating a large number of queries and determining throughput from the time necessary to complete all submissions. However, this alternative design gives the benchmark complete awareness of incoming queries. This design is reasonable when the execution order is FCFS but makes a significant difference when evaluating alternative priority schemes. Our short-query prioritizing policies force the completion rate to take a step-like distribution across time. This benchmark effectively classify queries into job size buckets, which will be drained one at a time, with the short job size buckets cleared first. We found the spin benchmark more reasonable to estimate maximum system throughput when queries have arbitrary arrival patterns and utilize it for this purpose.

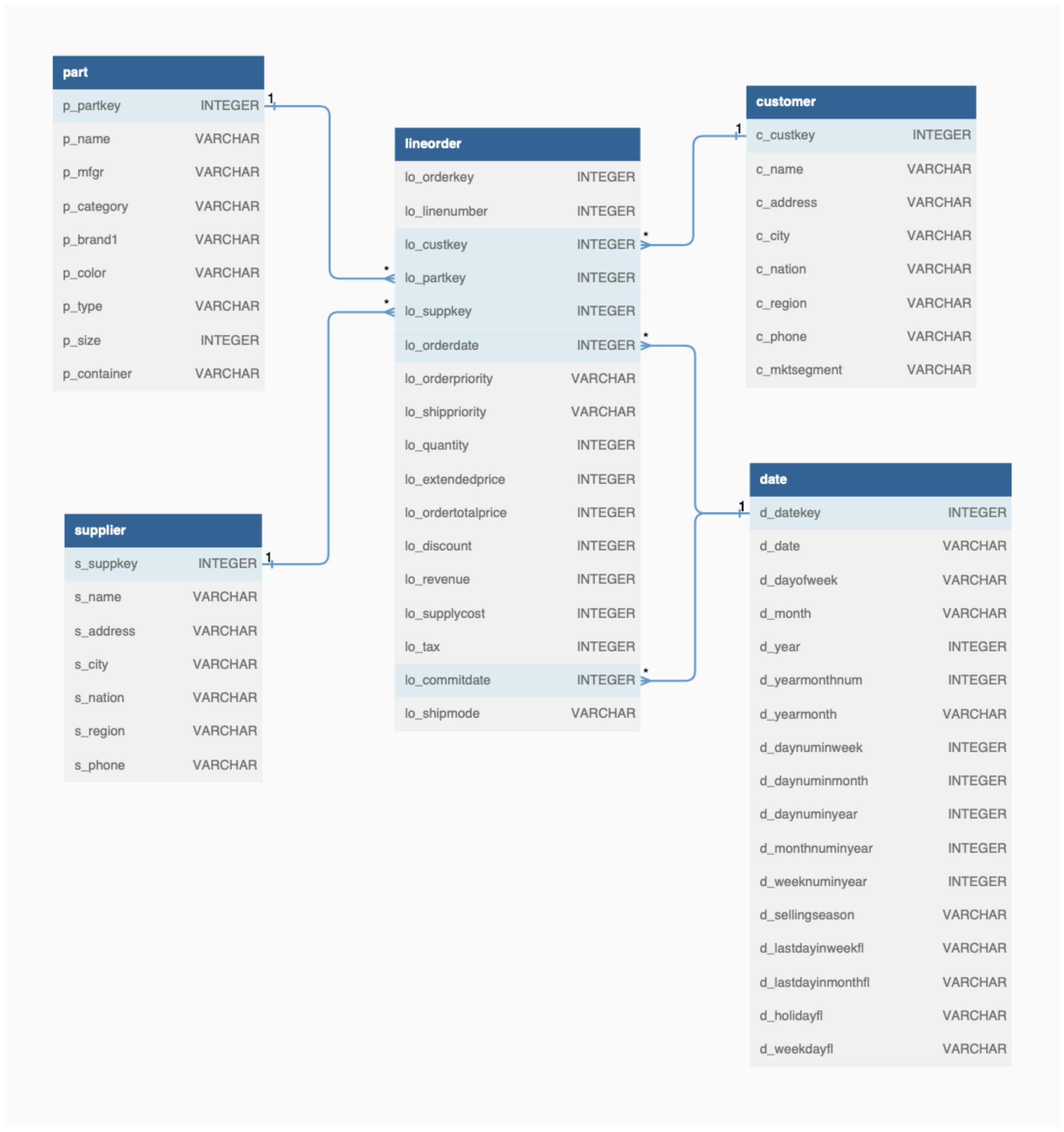


Figure 4.1: The Star Schema Benchmark contains a single fact table, **`lineorder`**, referenced by the four dimension tables **`part`**, **`supplier`**, **`customer`**, and **`date`**. This schema form is commonly used in data warehouses [47].

Table 4.1: A sample data traffic measurement over 20 seconds while running the Poisson benchmark with the IFSRPT policy. Socket 0 has significantly greater outgoing traffic than socket 1, and socket 1 has significantly greater incoming data.

Socket	Incoming Data Traffic	Outgoing Data and Non-Data Traffic
Socket 0	80G	242G
Socket 1	130G	193G

The second performance metric we are interested in is query latencies in a system under high load. This metric is the goal of our optimizations, as improved scheduling policies can significantly reduce average query latency in a stable system.

Poisson Benchmark We set up a second benchmark to submit queries with an open-system configuration. We configure a Poisson arrival process with a rate approximately corresponding to a baseline system load of 95%. We again pin task submission and benchmark measurement threads to a single core on the system. However, we now record the total latency of each query in the system, including both execution time and queueing time.

This benchmark's design defines an arrival process for queries. Instead of running the benchmark for a fixed period, we set up the benchmark to run 45000 queries to completion and exclude the first 15000 queries as a warmup phase.

Section 4.2 through Section 4.5 will focus on the performance insights which guided our development process. Section 4.6 will summarize the changes to average latency and system metrics as a result of our complete set of improvements.

4.2 Baseline

We motivate our system improvements by discussing the initial latency results for the Poisson benchmark experiment depicted in Figure 4.2. We determined the arrival rate based on the results of the spin experiment described later in this section.

The best policy in this graph is IFSRPT, which has an $\sim$8.5% performance improvement over the naive FCFS policy. These results did not match the expectations from query simulations based on NoisePage, which predicted a performance increase of up to 47% in comparing these policies [34]. The clear performance gap necessitated further investigation.

We began by considering system metrics observed for the IFSRPT policy in this benchmark. We utilize the Intel PCM tool to monitor interconnect usage, with which we observe an imbalance in the data traffic across sockets. The traffic samples during the experiment, one presented in Table 4.1, consistently indicates a clear skew towards accessing data from socket 0. Additionally, when inspecting internal system counters with `perf`, we found a significant number of remote memory accesses, nearly matching the count of local accesses. We show these counter values in Table 4.2. These insights led us to consider the overheads of remote data in our experiment.

To compute the maximum throughput for our baseline Poisson experiment and to compare against future results, we also ran the spin benchmark for all policies. We first plot a moving

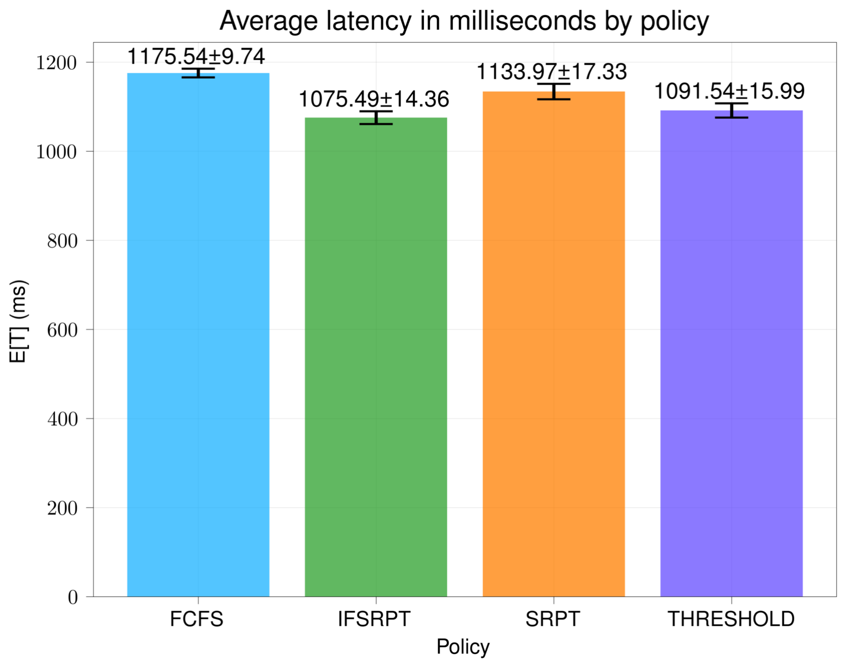


Figure 4.2: The average latency of queries across policies running the baseline Poisson benchmark shows a limited performance improvement over the naive scheduling order. The error bars indicate $2 * \text{SEM}$ $(n = 30000)$.

Table 4.2: Per-minute average local and remote memory accesses measuring the Poisson benchmark with the IFSRPT policy for four minutes.

Metric	Raw Counts	Relative Percentages
Local DRAM	1,535,689,325	57.01%
Remote DRAM	1,157,844,401	42.99%

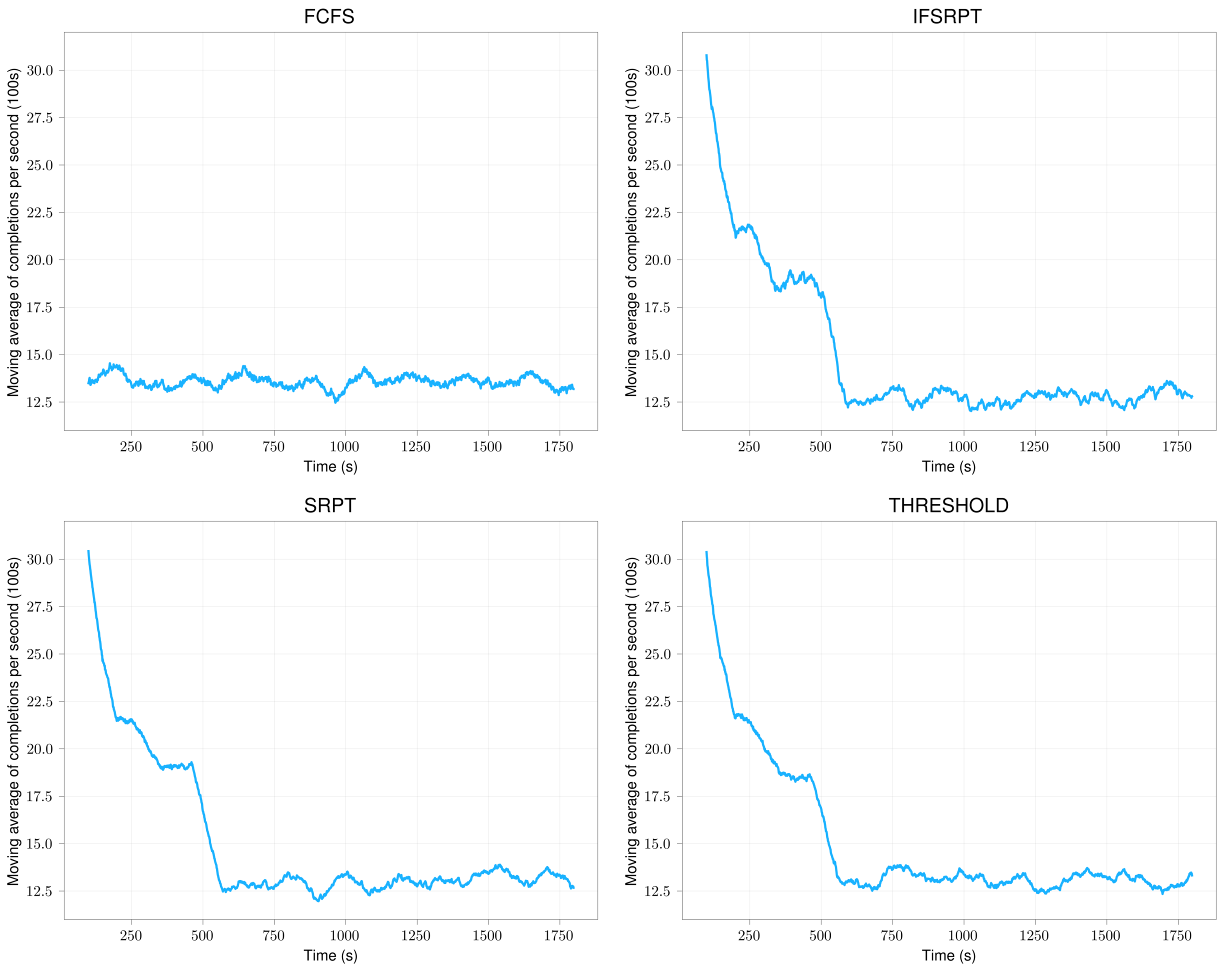


Figure 4.3: The moving averages of completions per second with a 100s window for each scheduling policy considered in the baseline spin benchmark.

average of completions per second in Figure 4.3 to determine the period of stability for each queueing policy. The moving average uses a minimum window size of 100s. The policies IF-SRPT, SRPT, and THRESHOLD all have an initial spike in completion rate before stabilizing within the first 750s. As such, we take the stable region for these policies to be the 1000-second time frame from 750s to 1750s. However, the FCFS policy is stable through more of the experiment, noting the different axes ranges presented in the figure. Accordingly, we consider an extended, 1500-second stability region from 250s to 1750s for this policy.

The stability periods we extract from this experiment similarly apply to our further experiments, so we defer the presentation of similar moving average plots to Chapter 7.

Using the stability periods defined in Table 4.3, we show the maximum throughput calculated per policy for the baseline in Figure 4.4. We observe that most policies have a maximum throughput of approximately 13 queries per second. We use this max load across all Poisson benchmark experiments to compare performance at the same load point as the baseline.

Table 4.3: The stability periods in the spin benchmark for each policy as determined from the moving average of completion rate. The stability periods are shared across experiments as they follow the same stability pattern.

Policy	**Stability Period Length (s)**	**Start Timestamp (s)**	**End Timestamp (s)**
FCFS	1500	250	1750
IFSRPT	1000	750	1750
SRPT	1000	750	1750
THRESHOLD	1000	750	1750

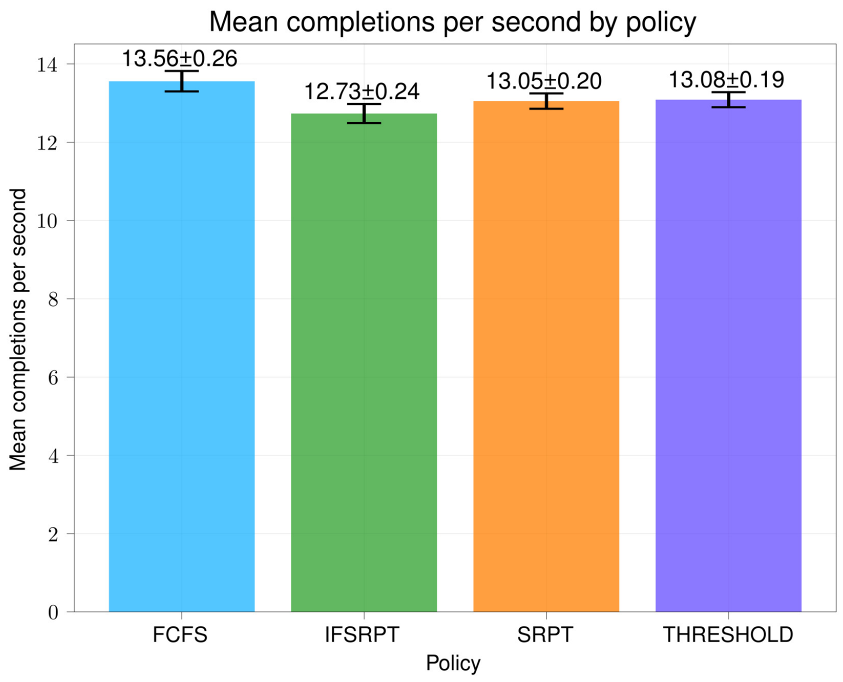


Figure 4.4: The maximum throughput per policy running the spin benchmark using the baseline system. The error bars indicate $2 * \text{SEM}$, n as defined in Table 4.3.

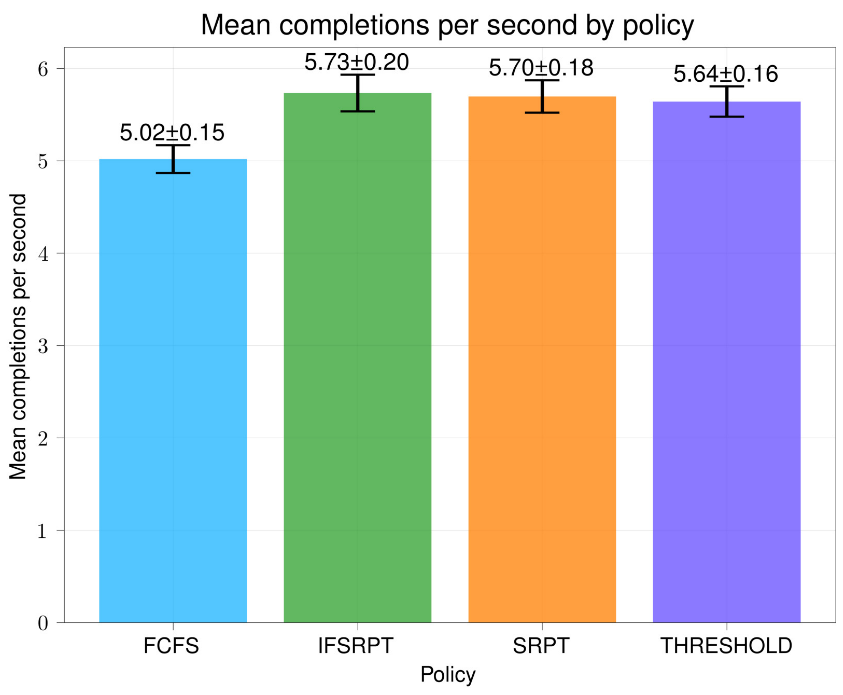


Figure 4.5: The maximum throughput per policy running the spin benchmark when striping data across NUMA nodes. The system throughput decreased by increased operation overheads from moving to our PAX-like layout and allocating data at a 4KB block size. The error bars indicate $2 * \text{SEM}$, n as defined in Table 4.3.

4.3 Resolving Access Skew

Our first step in addressing our baseline metric profiles was considering alternative allocation behavior. Our interconnect bandwidth numbers in Table 4.2 indicated a potential hotspot on socket 0. The default system allocation behavior would load the entire dataset onto the same NUMA region. Instead, we altered the loading behavior to stripe allocated data in 4KB page-size blocks across the two sockets to address this potential hotspot. We also modified the data organization to the PAX-like format described in Section 3.3.1.

However, our changes detrimentally impacted the performance of all queries in the system. In running the spin benchmark, we notice a sharp throughput drop across policies depicted in Figure 4.5 with our changes compared to the baseline, falling by over 2x.

The CPU profiles of the FCFS policy in this experiment show a higher function call overhead than the baseline, which we link to reducing the batch size. Using a page-sized block in the interest of finer granularity and alignment to the **`numa_alloc_onnode`** system call significantly reduced the number of tuples processed together [76]. For example, in the **`customer`** SSB table batches went from containing 10k tuples to containing just 35 tuples on a 4096-byte page. As a result, the internal function iterating lost some benefits of vectorization and grew from consuming 43% of cycles to 61% of cycles.

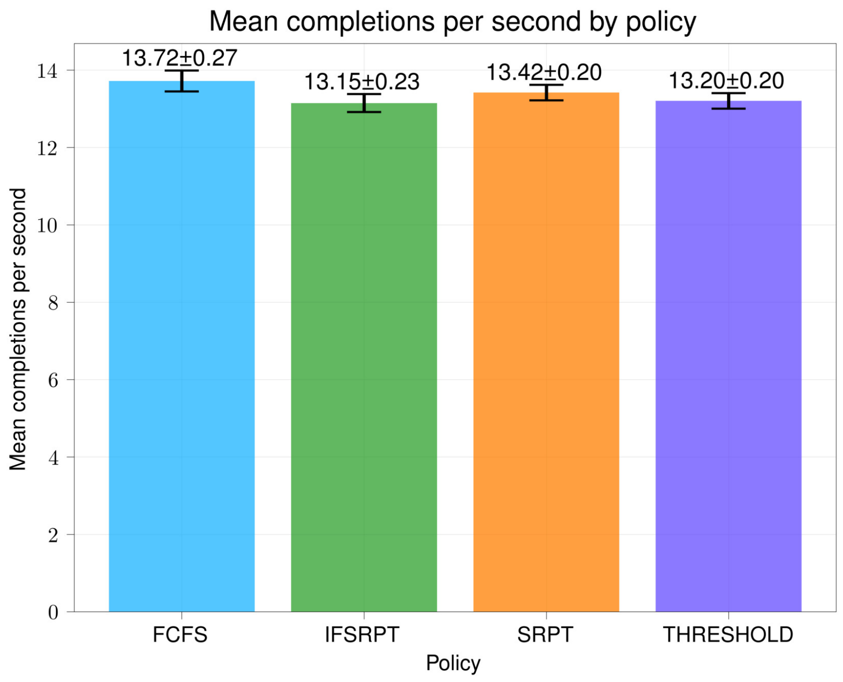


Figure 4.6: The maximum throughput per policy running the spin benchmark when using 2MB hugepages as data blocks. The system recovered the performance losses incurred from our previous modifications and even slightly improved. In this experiment, the FCFS and THRESHOLD policy max throughput changes are not statistically significant with respect to the baseline. The error bars indicate $2 * \text{SEM}$, n as defined in Table 4.3.

4.4 Increasing Block Sizes

Given the insight from our aforementioned tuples-per-block analysis, we next increase the storage capacity of our blocks by using hugepages, which we introduced in Section 3.3.1. Modifying the system to use 2MB hugepages, we were able to track $\sim$18k tuples of the **`customer`** table in a single block.

Figure 4.6 demonstrates that using hugepages recouped most of the performance degradation observed in Section 4.3. By processing data in larger batch sizes in this experiment, our CPU profiles indicate the CPU consumption of the block iteration function lowered, consuming 45% of cycles. Additionally, in spreading data across the sockets, we find that the traffic patterns of the two sockets approached similarity. Both incoming and outgoing traffic are closer across sockets 0 and 1 in Table 4.4 than in Table 4.1.

However, using hugepages when striping data across NUMA regions triggered three related metric regressions, potentially suggesting unresolved overheads: context switches, page faults, and TLB flushes, as shown in Table 4.5.

Table 4.4: A sample data traffic measurement over 20 seconds while running the Poisson benchmark with the IFSRPT policy with NUMA-aware data striping and the use of hugepages. Socket 0 and socket 1 now have far more similar traffic patterns.

Socket	Incoming Data Traffic	Outgoing Data and Non-Data Traffic
Socket 0	54G	130G
Socket 1	62G	122G

Table 4.5: Per-minute average counts for regressing metrics measuring the Poisson benchmark with the IFSRPT policy for four minutes.

Metric	Baseline	Hugepage Allocations
Context Switches	76,581	83,138
Page Faults	633,410	745,074
STLB Flushes	16,088,330	16,877,881
DTLB Flushes	311,604	346,936
ITLB Flushes	15,855,552	16,299,495

4.5 Performance Tuning

We implemented three memory-related changes to target some of the remaining observed metric disparities. Correspondence with OS experts suggested the following steps [75]:

- We turned off hyperthreading to avoid Spectre/Meltdown mitigations which may affect TLB flush rates.
- We changed our NUMA-aware allocations to request preallocated hugepages instead of using **`madvise`**.
- We avoided invocations of **`mmap`** for small allocations.

These alterations resolved most of the metric spikes from our previous development, except the number of page faults while running the benchmark. We present the improvement in these metrics in Table 4.6. These changes also slightly affect the max throughput of the system across policies, depicted in Figure 4.7.

We plot our re-evaluated latency metrics on the Poisson benchmark with the changes applied thus far in Figure 4.8. With hotspot alterations and OS-level performance tuning, in this experiment, the IFSRPT and THRESHOLD policies achieve a $\sim$9.6% and $\sim$15.2% performance improvement over FCFS, respectively. However, the FCFS result in this experiment had worse performance than the baseline, and the IFSRPT result did not exhibit a statistically significant difference. When compared with the baseline FCFS result, THRESHOLD achieves a $\sim$13.4% performance improvement. In all, our changes thus far did not improve upon our baseline result. We disproved suspicions that memory and interconnect bottlenecks were restricting performance with these results and subsequent measurements.

Table 4.6: Per-minute average counts for formerly regressed metrics measuring the Poisson benchmark with the IFSRPT policy for four minutes. Performance tuning resolved most of the degradations across metrics.

Metric	Baseline	Hugepage Allocations	Performance Tuning
Context Switches	76,581	83,138	26,345
Page Faults	633,410	745,074	681,913
STLB Flushes	16,088,330	16,877,881	13,053,544
DTLB Flushes	311,604	346,936	214,439
ITLB Flushes	15,855,552	16,299,495	12,461,092

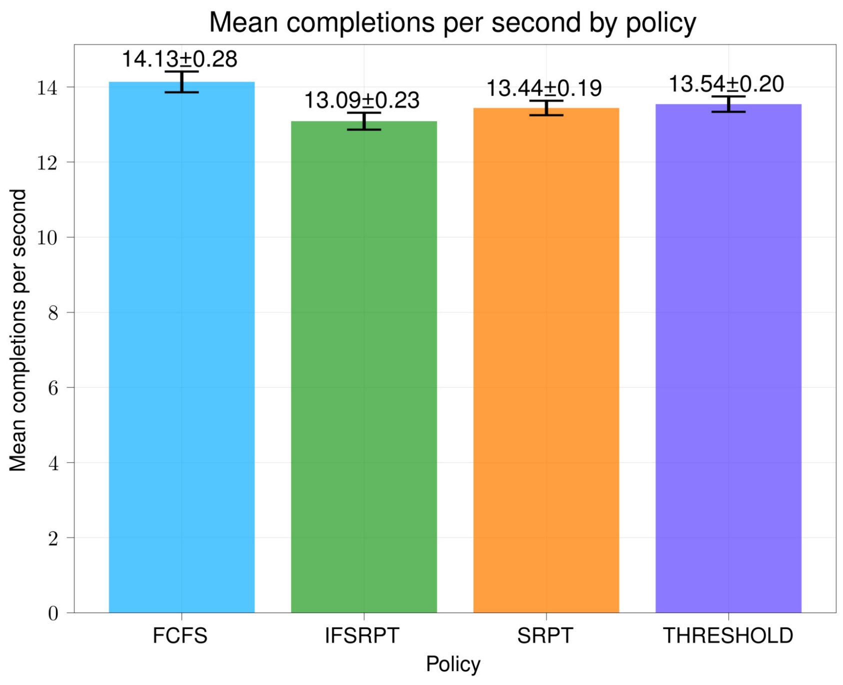


Figure 4.7: The maximum throughput per policy running the spin benchmark after applying our performance tuning changes, which slightly impacted the system. In this experiment, the IFSRPT and SRPT policy max throughput changes are not statistically significant with respect to the previous experiment. The error bars indicate $2 * \text{SEM}$, n as defined in Table 4.3.

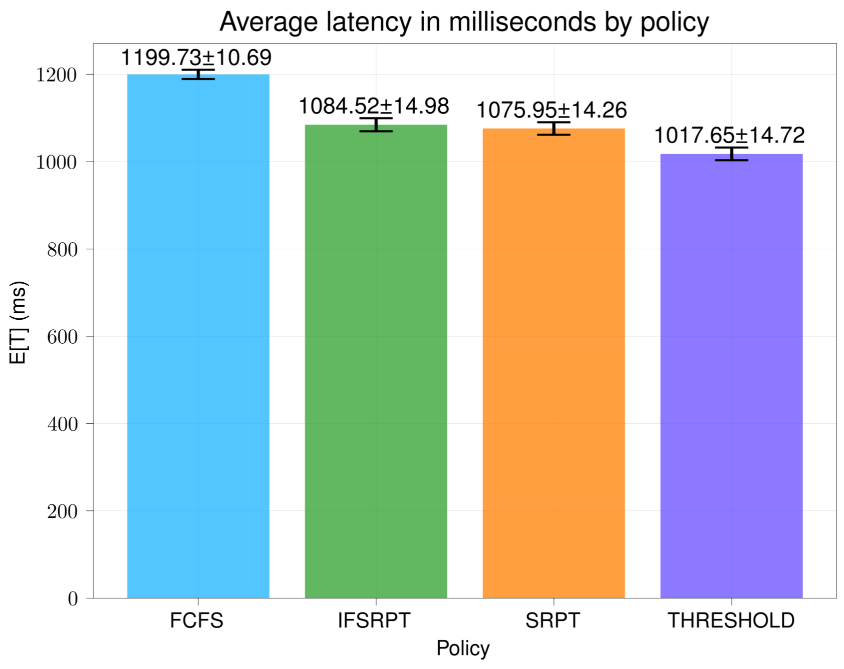


Figure 4.8: The average latency of queries in the Poisson benchmark after our performance tuning changes were applied continues to show limited performance improvement over the naive scheduling order. In this experiment, only the IFSRPT policy average latency is not statistically significant with respect to the baseline. The error bars indicate $2 * \text{SEM}$ ($n = 30000$).

Table 4.7: Per-minute average local and remote memory accesses measuring the Poisson benchmark with the IFSRPT policy for four minutes.

Metric	Raw Counts	Relative Percentages
Local DRAM	1,453,727,874	56.88%
Remote DRAM	1,101,951,249	43.12%

We still had one more metric to optimize: locality. The latest round of experiments had a similar pattern in local and remote accesses to our baseline. We again note a remote-heavy access pattern in Table 4.7.

4.6 NUMA-Aware Scheduling

Our prior developments and experiments made it clear that the system would need to support NUMA-aware scheduling of query tasks to optimize acting on local data, thereby avoiding the overheads of remote accesses.

As mentioned in Section 3.3, we implemented internal tracking structures to the database to maintain the NUMA node of each allocation and, during execution, process sequential scan operators in a locally-optimized manner. NUMA-aware scheduling noticeably impacts the max

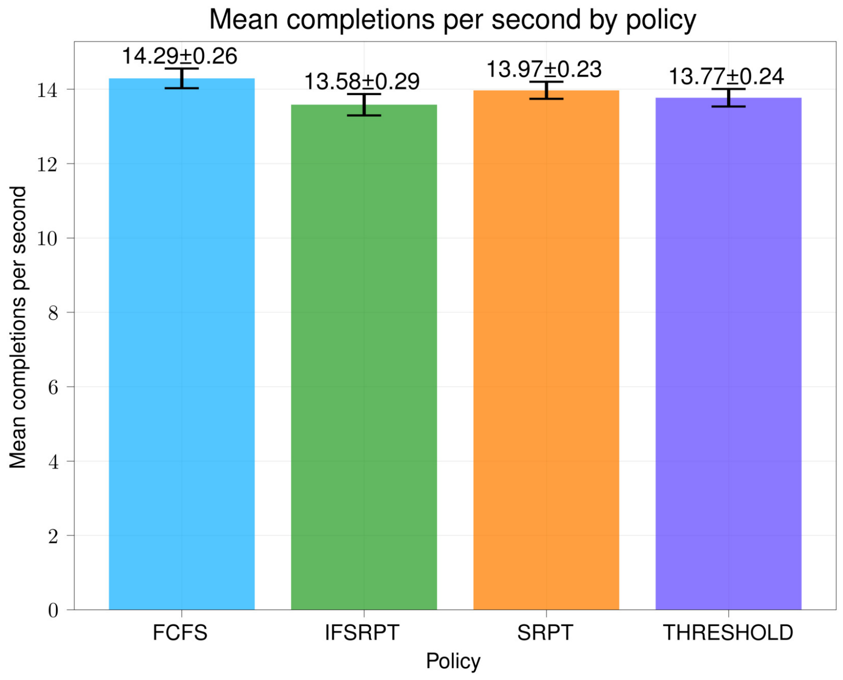


Figure 4.9: The maximum throughput per policy running the spin benchmark when using NUMA-aware scheduling. In this experiment, only the FCFS policy max throughput change is not statistically significant with respect to the previous experiment. The error bars indicate $2 * \text{SEM}$, n as defined in Table 4.3.

throughput of the system, demonstrated in Figure 4.9.

Applying NUMA-aware scheduling improved the average latency of the system when running the Poisson benchmark, as shown in Figure 4.10. Compared to the baseline FCFS result, IFSRPT achieves a $\sim$34.1% performance improvement in this experiment. FCFS also sees a $\sim$26.8% performance improvement over the policy's baseline performance.

DRAM access patterns do not directly explain the performance benefits of utilizing NUMA-aware scheduling policies. Our implementation did not dramatically affect the proportion of memory accessed locally. The proportions in Table 4.8 are similar to those in Table 4.7 and Table 4.2. Rather, NUMA-aware policies allow the system to take advantage of improved caching characteristics when optimizing most operations to act on local data. An expanded Table 4.9 demonstrates an increased hit rate in the cache hierarchy correlated with the performance improvements we observe. We hypothesize that region-independent caching can reduce cache efficiency when duplicating data in the caches of multiple regions. By consistently maintaining data locally, the caches may instead avoid data replication across sockets, improving caching effects in the system. We additionally speculate that the prioritizing scheduling policies can better take advantage of shared data requests for pipelines across invocations of the same query and potentially across several queries executed in succession due to the prioritization scheme.

Prior metrics we investigated continue improving: the values in Table 4.10 all are better than the results presented in Table 4.6. Interconnect traffic patterns also remain relatively similar,

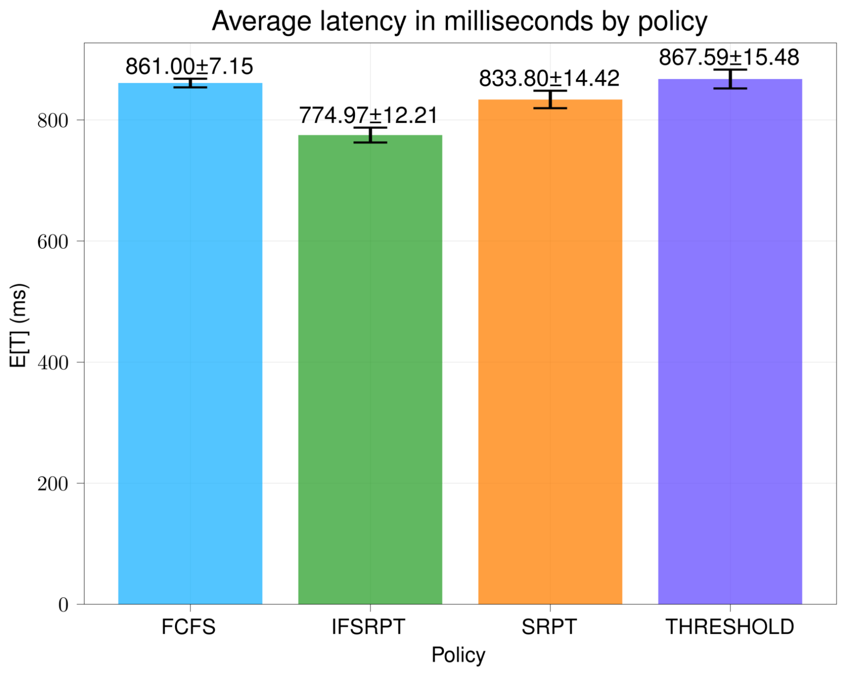


Figure 4.10: The average latency of queries running the Poisson benchmark with NUMA-aware scheduling shows a marked performance improvement on all policies other than FCFS. The error bars indicate $2 * \text{SEM}$ ($n = 30000$).

Table 4.8: Per-minute average local and remote memory accesses measuring the Poisson benchmark with the IFSRPT policy for four minutes.

Metric	Raw Counts	Relative Percentages
Local DRAM	988,338,178	53.92%
Remote DRAM	844,624,098	46.08%

Table 4.9: Per-minute average counts for cache and memory metrics measuring the Poisson benchmark with the IFSRPT policy for four minutes. Increased cache efficacy results in significantly fewer memory accesses, potentially contributing to the policy's performance improvement.

Metric	Baseline	Performance Tuning	NUMA-Aware Scheduling
L1 Cache Hit	967,936,706,093	969,387,392,256	1,026,996,656,882
L1 Cache Miss	29,945,235,199	30,479,332,002	31,023,064,344
L2 Cache Hit	16,531,638,342	16,832,982,220	17,449,109,121
L2 Cache Miss	13,336,222,568	13,646,477,477	13,846,426,909
L3 Cache Hit	9,974,854,449	10,533,353,630	11,605,222,231
L3 Cache Miss	2,931,819,490	2,761,672,642	2,012,850,592
Local DRAM	1,535,689,325	1,453,727,874	988,338,178
Remote DRAM	1,157,844,401	1,101,951,249	844,624,098

Table 4.10: Per-minute average counts for address translation metrics measuring the Poisson benchmark with the IFSRPT policy for four minutes.

Metric	Baseline	Performance Tuning	NUMA-aware Scheduling
Context Switches	76,581	26,345	25,664
Page Faults	633,410	681,913	623,400
STLB Flushes	16,088,330	13,053,544	11,599,572
DTLB Flushes	311,604	214,439	207,466
ITLB Flushes	15,855,552	12,461,092	10,517,490

Table 4.11: A sample data traffic measurement over 20 seconds while running the Poisson benchmark with the IFSRPT policy with NUMA-aware scheduling policies. Socket 0 and socket 1 continue to have similar traffic patterns.

Socket	Incoming Data Traffic	Outgoing Data and Non-Data Traffic
Socket 0	54G	128G
Socket 1	62G	120G

shown in Table 4.11.

However, within the NUMA-aware Poisson latency benchmark, IFSRPT outperforms FCFS by $\sim$10%, still far short of the $\sim$47% mark hoped for from simulations. In optimizing noticeable system metrics, addressing potential bottlenecks, and augmenting the scheduling algorithms for local performance, we hoped we could resolve behavior that simulations did not capture. That

Table 4.12: We summarize our mean latency results across experiments in this table. We report the mean latency in ms as well as the percentage change from the baseline FCFS result. Our results show the performance gains achieved by NUMA awareness as well as the benefits of query characteristic aware scheduling.

Policy	**Baseline**	**Performance Tuning**	**NUMA-aware Scheduling**
FCFS	1175.54 (N/A)	1199.73 (+2.1%)	861.00 (-26.8%)
IFSRPT	1075.49 (-8.5%)	1084.52 (-7.7%)	774.97 (-34.1%)
SRPT	1133.97 (-3.5%)	1075.95 (-8.5%)	833.80 (-29.1%)
THRESHOLD	1091.54 (-7.1%)	1017.65 (-13.4%)	867.59 (-26.2%)

said, our results for NUMA-aware IFSRPT in Figure 4.10 outperform the baseline FCFS policy by well over 30%, and IFSRPT outperforms the next most performant scheduling algorithm by $\sim 7.1\%$, so we do see significant value in designing efficient scheduling algorithms combining queueing theoretic principles with architecture awareness.

4.7 Summary

We present a summary of our experimental results in Table 4.12. Our initial experiments demonstrated the value of scheduling in NoisePage, achieving up to an 8.5% mean latency improvement with the IFSRPT policy over FCFS. Our data layout changes and subsequent choices to utilize hugepages and address memory-related metrics did not drastically change this result. The THRESHOLD policy attains the best result in this experiment with a 13.4% performance improvement over the baseline FCFS result. Finally, adding architecture awareness to the scheduling framework allows the IFSRPT policy to achieve a performance improvement of 34.1% over the baseline FCFS policy.

Our experiments support the hypothesis that resource efficiency is an important consideration in query execution which can be improved with the application of scheduling and awareness of the system architecture. In addition to the technical details derived in this work, we contribute a methodology for evaluating performance results derived from scheduling in databases.

Our work presented a targeted insight into the Star-Schema Benchmark. This analytical benchmark is one of many, and the benchmark itself can be configured with varying dataset sizes. Our work may have even more favorable performance characteristics when increasing the variability of job sizes, for example, when interleaving SSB instances with scale factors 10 and 100, respectively. We believe the query characteristic-aware scheduling policies would further outperform the baseline as the workload variability increases. We leave these investigations to future experimentation and analysis.

Chapter 5

Related Work

Our work in applying theoretical and architectural principles to DBMS design builds upon a significant history of work in the field. We discuss a brief history of research in database scheduling, resource awareness, and query processing frameworks in this chapter.

5.1 Database Scheduling

We previously outlined scheduling policies implemented by open-source, commercial, and academic DBMSs in Chapter 2. Wagner et. al. presented developments in scheduling in the Umbra DBMS, also building upon the data-parallel and compilation-based execution model introduced by the HyPer system. Their scheduling implementation shows a stride scheduling approach with self-tuned hyperparameters to both outperform alternative scheduling policies implemented in Umbra and the default behaviors of other systems. Like ours, their component evaluated performance characteristics of scheduling policies in a system under high load [54, 77]. To the best of our knowledge, Amazon Redshift is the only other system to utilize a performance-based prioritization scheme applied to query scheduling [19, 27]. These developments similarly demonstrated the value of an SRPT-like algorithm. Psaroudakis et. al. present a scheduler infrastructure that supports user-defined priorities, but this work focuses on on dynamically determining task concurrency than on prioritization [67].

Scheduling considerations have more recently been considered in optimizing system metrics beyond performance. Landgraf et. al. pressent work on search algorithms to determine pipeline execution orders to minimize memory consumption [52].

Effective models for query performance prediction can replace the naive and restrictive approach to job size estimations taken in this work. QPPNet introduced query plan-structured neural networks assembling individual operator models to predict query performance. The plan-structured architecture allows the model to reflect the behavior of query evaluation and achieves greater accuracy and efficiency through operator-level model reuse [59]. MB2 presented an offline modeling architecture implemented for the NoisePage database. The performance modeling system decomposes database operations into miniature performance models and an overarching interference model to determine interactions of these individual components [58]. Redshift also notes the use of machine learning to estimate query performance utilized in their query acceleration module [19, 27].

5.2 NUMA Awareness

Our engine improvements are inspired by the parallelism approach implemented in the HyPer DBMS by Leis et. al. “Morsel-driven” parallelism was designed to improve system efficiency by increasing query elasticity, allowing compute resources to be efficiently reassigned, maintaining locality, and avoiding straggler effects [54]. Liu et. al. present a NUMA-aware execution engine, Zen+, to support transactional workloads on non-volatile memory. Zen+ takes a similar approach to our NUMA-aware partitioning and execution scheme, but supports dynamically reassigning partitions to alternative nodes [56].

Psaroudakis et. al. focus on data partitioning and task assignment strategies to fully extract NUMA awareness’s benefits, showing that naive partitioning approaches leave room for growth. Their work demonstrates the potential pitfalls of excessive partitioning and work-stealing and introduces principles for automated workload-aware tuning [68].

Schuh et. al. and Balkesen et. al. took a focused approach in evaluating various database join algorithms. These works considered and took advantage of NUMA effects to improve the performance of hash and sort-merge joins and in turn improve overall execution performance. Among their contributions was a discussion on how to adjust many of the algorithms they considered to be NUMA-aware. Their takeaways indicated memory awareness was a significant contributor to performance [28, 71].

Bang et. al. more recently presented resource partitioning beyond architectural design, highlighting potential pitfalls when software access patterns are left unconsidered. Their approach provisions resources to individual data structure instances rather than tasks based on measurements of performance under different resource allocations [29].

5.3 Compilation and Vectorization

NoisePage similarly builds upon a large body of prior work in query compilation and vectorization. Compilation has a long history in database development, starting with IBM’s System R, which compiled SQL queries by assembling machine code fragments [37]. In the modern era, work on compilation began with the development of HIQUE by Krikellas et. al., an engine transpiling a query plan into a C program. The system’s code generation component generates query-specific source code for each operator and a composing method to evaluate these in sequence. The generated code is then compiled and linked into the execution component for evaluation [51]. Neumann et. al. introduced the concept of operator pipelining to query compilation to generate more expressive kernels. A pipelined execution model reduces the costs of materialization and function calls. The choice of the LLVM framework as a compilation backend additionally provides more desirable optimization behavior than alternative targets [63, 64]. In situations where compilation is expensive with respect to the runtime of the query, interpretation may be a better choice. Kohn et. al. and Menon et. al. have implemented an adaptive execution mode in HyPer and NoisePage, respectively, dynamically allowing the choice between interpretation and compilation to address this issue [50, 62].

Boncz et. al. were the first to demonstrate the benefits of vectorized processing with the introduction of MonetDB/X100. Vectorized processing demonstrates magnitudes of performance improvement over prior work by significantly reducing interpretation overheads and enabling batch optimizations [35]. IBM DB2 adopted vectorized processing with the BLU Acceleration layer, a columnar processing engine. BLU combined several optimizations in columnar processing, including compression, the use of SIMD instructions, and parallelism [69]. Polychroniou et. al. have since developed SIMD implementations for various operators to extract further performance gains using SIMD instructions [66]. Lang et. al. extended this prior work in presenting new, performant algorithms to take advantage of the AVX-512 instruction set [53].

Behm et. al. presented a native, vectorized query engine developed at Databricks, Photon. Photon set a performance record for the TPC-DS benchmark on a 100TB dataset [31]. The principles of vectorization also lend well to GPU hardware, which are optimized for SIMD operations. Shanbhag et. al. present a performant execution engine build for modern GPUs which can greatly outperform CPU execution [73].

Kersten et. al. also compared the performance of vectorized and compiled query engines to determine their strengths and weaknesses. Although neither approach was a clear winner over the other, both had favorable performance situations. Among other attributes, compilation techniques were shown to be more performant on compute-intensive queries, while vectorization offered lower cache miss rates [49].

Menon et. al. showed that compilation and vectorization could be effectively combined in a single engine to extract greater performance gains in building NoisePage [61].

Chapter 6

Future Work

We believe this work to be part of a fascinating line of future research in applying queueing theory to databases. We will discuss lines of future investigation based on trends gleaned from this work, a rapidly changing hardware landscape, and interesting open problems our work raises.

6.1 Scheduling with Diverse Workloads

Another open area of interest is scheduling in a transactional environment. Transactional workloads, or OLTP workloads, typically involve far shorter operations than analytics but may significantly interfere across transactions from query conflicts. These workloads raise interesting scheduling questions about optimizing system performance by relieving resource contention.

Working with query deadlines and SLAs has the potential to be an exciting area of future work. Most commercial DBMSs discussed in Section 2.3 have some notion of user-defined priority. Interesting scheduling decisions arise when considering query characteristics in addition to user-defined priorities or deadlines. Phase awareness, dynamic configuration of query scalability, and practical size estimations may improve system performance without sacrificing performance targets or tiers. Prior work also suggests interest in deadlines and priorities on transactional workloads but was primarily conducted in a different computational era [23, 36].

6.2 Scheduling and Vertical Scalability

Today, the highest core count systems offer exceptional opportunities for high-performance parallel execution. System efficiency becomes more important as the resource capabilities of a single system scale. With more resources, there is a greater risk of underutilization and inefficiency, the key targets of our optimizations in NoisePage. In particular, we expect that as the number of cores increases, IFSRPT will become more performant compared to alternative algorithms. Similarly, as the number of sockets and performance disparities increase, locality-aware scheduling will play a more significant role.

6.3 Scheduling and Horizontal Scalability

In addition to the benefits of more powerful hardware, the attractive pricing models of lower-capability hardware have made distributed query processing attractive. Scheduling on a cluster of such nodes may achieve the high resource capacity hoped from a large instance at a far lower cost. In doing so, the same resource inefficiency risks of vertical scalability arise: increasing resources increases the opportunity for inefficiency. However, the performance overheads of distributed systems are far more significant, making locality-aware execution paramount.

There are also other interesting theoretical and modeling questions in working with a distributed system. The parallelization models of query components may change drastically from those observed from a single-node system when dealing with communication costs and different execution algorithms. New query performance models may lead to more targeted theoretical results to address these changing behaviors.

6.4 Scheduling Beyond Execution

Query execution is but one part of a full-featured DBMS. We focus on in-memory systems in this document. However, the majority of databases today are still depend on disk or network communication. These environments increase performance overheads and may drastically change execution models. Scheduling algorithms may improve by being aware of these additional resources and their interactions with query processing.

Further, many databases also perform background tasks not typically on the critical path for execution. Many systems support garbage collection, checkpointing in the background or on demand, and logging for durability. Work on alternative scheduling algorithms may successfully address new challenges for each of these functions.

Chapter 7

Conclusion

With limited prior investigation into scheduling in state-of-the-art analytical systems, this work makes inroads into a relatively unexplored subdomain in database research. We presented scheduling advancements to an in-memory execution engine to improve analytical workloads. Our execution engine modifications demonstrate the value of designing scheduling algorithms to combine query characteristics and architecture awareness, achieving average workload latency improvements of 30%. We consider our work a first step in empowering future research to explore and investigate new approaches in performant DBMS design beyond the techniques learned over decades of work in optimizing single-query performance.

Bibliography

[1] Overview of clickhouse architecture. URL https://clickhouse.com/docs/en/development/architecture/. 2.3.1, 2.3.2

[2] 13.2.6 insert statement, . URL https://dev.mysql.com/doc/refman/8.0/en/insert.html. 2.3.1

[3] 13.2.10 select statement, . URL https://dev.mysql.com/doc/refman/8.0/en/select.html. 2.3.1

[4] 5.6.3.3 thread pool operation, . URL https://dev.mysql.com/doc/refman/8.0/en/thread-pool-operation.html. 2.3.1

[5] 8.1 parallel execution concepts, . URL https://docs.oracle.com/en/database/oracle/oracle-database/21/vldbg/parallel-exec-intro.html. 2.3.1

[6] 8.4 parallel statement queuing, . URL https://docs.oracle.com/en/database/oracle/oracle-database/21/vldbg/about-parallel-queuing.html. 2.3.1

[7] 15.2. when can parallel query be used? URL https://www.postgresql.org/docs/current/when-can-parallel-query-be-used.html. 2.3.1

[8] Job scheduling. URL https://spark.apache.org/docs/latest/job-scheduling.html. 2.3.2

[9] Umbra. URL https://umbra-db.com/. 2.3.3

[10] Snowflake challenge: Concurrent load and query, 2015. URL https://www.snowflake.com/blog/snowflake-challenge-concurrent-load-and-query/. 2.3.1

[11] Scheduling for efficiency and fairness in systems with redundancy. *Perform. Eval.*, 116 (C):1–25, nov 2017. ISSN 0166-5316. doi: 10.1016/j.peva.2017.07.001. URL https://doi.org/10.1016/j.peva.2017.07.001. 1.2.2

[12] How to: Understand queuing, 2020. URL https://community.snowflake.com/s/article/Understanding-Queuing. 2.3.1

[13] Task preemption, 2021. URL https://docs.databricks.com/clusters/preemption.html. 2.3.2

[14] Methods of parallel processing, 2022. URL https://www.ibm.com/docs/en/db2-for-zos/11?topic=processing-methods-parallel. 2.3.1

[15] Session priority, 2022. URL `https://www.ibm.com/docs/en/db2/11.5?topic=environment-session-priority`. 2.3.1

[16] Activity queuing, 2022. URL `https://www.ibm.com/docs/en/db2/11.5?topic=thresholds-activity-queuing`. 2.3.1

[17] *PERF(1) perf Manual*, 2022. 4

[18] Query priority, 2022. URL `https://docs.aws.amazon.com/redshift/latest/dg/query-priority.html`. 2.3.3

[19] Working with short query acceleration, 2022. URL `https://docs.aws.amazon.com/redshift/latest/dg/wlm-short-query-acceleration.html`. 2.3.3, 5.1

[20] Workload management, 2022. URL `https://docs.aws.amazon.com/redshift/latest/dg/c_workload_mngmt_classification.html`. 2.3.3

[21] Thread and task architecture guide, 2022. URL `https://learn.microsoft.com/en-us/sql/relational-databases/thread-and-task-architecture-guide?view=sql-server-ver16`. 2.3.2

[22] oneapi threading building blocks. `https://github.com/oneapi-src/oneTBB`, 2022. 3.1

[23] Robert K. Abbott and Hector Garcia-Molina. Scheduling real-time transactions: A performance evaluation. *ACM Trans. Database Syst.*, 17(3):513–560, sep 1992. ISSN 0362-5915. doi: 10.1145/132271.132276. URL `https://doi.org/10.1145/132271.132276`. 6.1

[24] Sanjay Agrawal, Surajit Chaudhuri, and Vivek R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, page 496–505, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc. ISBN 1558607153. 1

[25] Anastassia Ailamaki, David J. DeWitt, and Mark D. Hill. Data page layouts for relational databases on deep memory hierarchies. *The VLDB Journal*, 11(3):198–215, nov 2002. ISSN 1066-8888. doi: 10.1007/s00778-002-0074-9. URL `https://doi.org/10.1007/s00778-002-0074-9`. 3.3.1

[26] Victor Alvarez, Stefan Richter, Xiao Chen, and Jens Dittrich. A comparison of adaptive radix trees and hash tables. In *2015 IEEE 31st International Conference on Data Engineering*, pages 1227–1238, 2015. doi: 10.1109/ICDE.2015.7113370. 1

[27] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, TJ Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinksy, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. Amazon redshift reinvented. In *SIGMOD/PODS 2022*, 2022. URL `https://www.amazon.science/publications/amazon-redshift-re-invented`. 1.2.2, 2.3.3, 5.1

[28] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endow.*, 7(1):85–96, sep 2013. ISSN 2150-8097. doi: 10.14778/2732219.2732227. URL `https://doi.org/10.14778/2732219.2732227`. 1, 1.2.1, 1.2.1, 5.2

[29] Tiemo Bang, Ismail Oukid, Norman May, Ilia Petrov, and Carsten Binnig. Robust performance of main memory data structures by configuration. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1651–1666, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3389725. URL `https://doi.org/10.1145/3318464.3389725`. 5.2

[30] Jeff Barr. Ec2 high memory update – new 18 tb and 24 tb instances, Oct 2019. URL `EC2HighMemoryUpdateNew18TBand24TBInstances`. 1.1

[31] Alexander Behm, Shoumik Palkar, Utkarsh Agarwal, Timothy Armstrong, David Cashman, Ankur Dave, Todd Greenstein, Shant Hovsepian, Ryan Johnson, Arvind Sai Krishnan, Paul Leventis, Ala Luszczak, Prashanth Menon, Mostafa Mokhtar, Gene Pang, Sameer Paranjpye, Greg Rahn, Bart Samwel, Tom van Bussel, Herman van Hovell, Maryann Xue, Reynold Xin, and Matei Zaharia. Photon: A fast query engine for lakehouse systems. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 2326–2339, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392495. doi: 10.1145/3514221.3526054. URL `https://doi.org/10.1145/3514221.3526054`. 5.3

[32] Benjamin Berg. *A Principled Approach to Parallel Job Scheduling*. PhD thesis, 2022. URL `https://bsb20.github.io/bsberg_phd_csd_2022.pdf`. 1.2, 1.2.2, 3.1, 3.1

[33] Benjamin Berg, Jan-Pieter Dorsman, and Mor Harchol-Balter. Towards optimality in parallel job scheduling. *SIGMETRICS Perform. Eval. Rev.*, 46(1):116–118, jun 2018. ISSN 0163-5999. doi: 10.1145/3292040.3219666. URL `https://doi.org/10.1145/3292040.3219666`. 2.2.2

[34] Benjamin Berg, Justin Whitehouse, Benjamin Moseley, Weina Wang, and Mor Harchol-Balter. The case for phase-aware scheduling of parallelizable jobs. *Performance Evaluation*, 153:102246, 2022. ISSN 0166-5316. doi: https://doi.org/10.1016/j.peva.2021.102246. URL `https://www.sciencedirect.com/science/article/pii/S0166531621000638`. 3.2, 3.2.2, 4.2

[35] Peter A. Boncz, Marcin Zukowski, and Niels Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, 2005. 1, 5.3

[36] A. Buchmann, D. McCarthy, M. Hsu, and U. Dayal. Time-critical database scheduling: a framework for integrating real-time scheduling and concurrency control. In *Proceedings. Fifth International Conference on Data Engineering*, pages 470,471,472,473,474,475,476,477,478,479,480, Los Alamitos, CA, USA, feb 1989. IEEE Computer Society. doi: 10.1109/ICDE.1989.47251. URL `https://doi.ieeecomputersociety.org/10.1109/ICDE.1989.47251`. 6.1

[37] Donald D. Chamberlin, Morton M. Astrahan, Michael W. Blasgen, James N. Gray,

W. Frank King, Bruce G. Lindsay, Raymond Lorie, James W. Mehl, Thomas G. Price, Franco Putzolu, Patricia Griffiths Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system r. *Commun. ACM*, 24(10):632–646, oct 1981. ISSN 0001-0782. doi: 10.1145/358769.358784. URL `https://doi.org/10.1145/358769.358784`. 5.3

[38] Inc. ClickHouse. Clickhouse. URL `https://github.dev/ClickHouse/ClickHouse`. 2.3.1, 2.3.2

[39] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13 (6):377–387, jun 1970. ISSN 0001-0782. doi: 10.1145/362384.362685. URL `https://doi.org/10.1145/362384.362685`. 1

[40] Intel Corporation. Intel® performance counter monitor. URL `https://github.com/intel/pcm`. 4

[41] Kalen Delaney and Craig Freeman. *Microsoft SQL Server 2012 Internals*. Microsoft Press, 2013. 2.3.2

[42] David DeWitt and Jim Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, jun 1992. ISSN 0001-0782. doi: 10.1145/129888.129894. URL `https://doi.org/10.1145/129888.129894`. 1.2.1

[43] G. Graefe. Volcano/spl minus/an extensible and parallel query evaluation system. *IEEE Transactions on Knowledge and Data Engineering*, 6(1):120–135, 1994. doi: 10.1109/69.273032. 1.1.1

[44] Goetz Graefe and Harumi Kuno. Modern b-tree techniques. In *2011 IEEE 27th International Conference on Data Engineering*, pages 1370–1373, 2011. doi: 10.1109/ICDE.2011.5767956. 1

[45] Mor Harchol-Balter. *Performance Modeling and Design of Computer Systems: Queueing Theory in Action*. Cambridge University Press, 2013. doi: 10.1017/CBO9781139226424. 2.2.1, 2.2.2, 2.2.3

[46] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011. 1.2.2

[47] Holistics. dbdiagram.io, 12 2022. URL `https://dbdiagram.io/`. 4.1

[48] JGraph. diagrams.net, draw.io, 12 2022. URL `https://www.diagrams.net/`. 1.1, 1.2, 2.1, 2.2, 2.3, 2.4, 3.3, 3.5

[49] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow.*, 11(13):2209–2222, sep 2018. ISSN 2150-8097. doi: 10.14778/3275366.3284966. URL `https://doi.org/10.14778/3275366.3284966`. 1, 2.1, 5.3

[50] André Kohn, Viktor Leis, and Thomas Neumann. Adaptive execution of compiled queries. In *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pages 197–208, 2018. doi: 10.1109/ICDE.2018.00027. 5.3

[51] Konstantinos Krikellas, Stratis Viglas, and Marcelo H. Cintra. Generating code for holistic

query evaluation. *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 613–624, 2010. 1.1.2, 5.3

[52] Lukas Landgraf, Wolfgang Lehner, Florian Wolf, and Alexander Boehm. Memory efficient scheduling of query pipeline execution. In *12th Conference on Innovative Data Systems Research, CIDR 2022, Chaminade, CA, USA, January 9-12, 2022*. www.cidrdb.org, 2022. URL `https://www.cidrdb.org/cidr2022/papers/p82-landgraf.pdf`. 5.1

[53] Harald Lang, Andreas Kipf, Linnea Passing, Peter Boncz, Thomas Neumann, and Alfons Kemper. Make the most out of your simd investments: Counter control flow divergence in compiled query pipelines. In *Proceedings of the 14th International Workshop on Data Management on New Hardware*, DAMON '18, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450358538. doi: 10.1145/3211922.3211928. URL `https://doi.org/10.1145/3211922.3211928`. 5.3

[54] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, page 743–754, New York, NY, USA, 2014. Association for Computing Machinery. ISBN 9781450323765. doi: 10.1145/2588555.2610507. URL `https://doi.org/10.1145/2588555.2610507`. 1.2.1, 2.3.1, 3, 3.3, 5.1, 5.2

[55] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How good are query optimizers, really? *Proc. VLDB Endow.*, 9(3):204–215, nov 2015. ISSN 2150-8097. doi: 10.14778/2850583.2850594. URL `https://doi.org/10.14778/2850583.2850594`. 1.2.2

[56] Gang Liu, Leying Chen, and Shimin Chen. Zen+: a robust numa-aware oltp engine optimized for non-volatile main memory. volume 32, pages 123–148, 2023. ISBN 0949-877X. doi: 10.1007/s00778-022-00737-1. URL `https://doi.org/10.1007/s00778-022-00737-1`. 5.2

[57] LucaCanali. Notes and tools for measuring cpu-to-memory throughput in linux. URL `https://github.com/LucaCanali/Miscellaneous/blob/master/Spark_Notes/Tools_Linux_Memory_Perf_Measure.md`. 4

[58] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Lim, Prashanth Menon, and Andrew Pavlo. Mb2: Decomposed behavior modeling for self-driving database management systems. pages 1248–1261, 06 2021. doi: 10.1145/3448016.3457276. 1.2.2, 5.1

[59] Ryan Marcus and Olga Papaemmanouil. Plan-structured deep neural network models for query performance prediction. *Proc. VLDB Endow.*, 12(11):1733–1746, jul 2019. ISSN 2150-8097. doi: 10.14778/3342263.3342646. URL `https://doi.org/10.14778/3342263.3342646`. 1.2.2, 5.1

[60] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making learned query optimization practical. 2021. 1.2.2

[61] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. Relaxed operator fusion for in-

memory databases: Making compilation, vectorization, and prefetching work together at last. *Proceedings of the VLDB Endowment*, 11:1–13, September 2017. URL `https://db.cs.cmu.edu/papers/2017/p1-menon.pdf`. 1, 1.1.1, 1.1.2, 1.3, 2.1, 5.3

[62] Prashanth Menon, Amadou Ngom, and Andrew Pavlo Lin Ma, Todd C. Mowry. Permutable compiled queries: Dynamically adapting compiled queries without recompiling. *Proc. VLDB Endow.*, 14(2):101–113, 2020. URL `https://db.cs.cmu.edu/papers/2020/p101-menon.pdf`. 1, 1.1.1, 1.1.2, 1.3, 2.1, 5.3

[63] Thomas Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, jun 2011. ISSN 2150-8097. doi: 10.14778/2002938.2002940. URL `https://doi.org/10.14778/2002938.2002940`. 1, 1.1.2, 2.1, 5.3

[64] Thomas Neumann. Evolution of a compiling query engine. *Proc. VLDB Endow.*, 14(12): 3207–3210, oct 2021. ISSN 2150-8097. doi: 10.14778/3476311.3476410. URL `https://doi.org/10.14778/3476311.3476410`. 1.1.2, 5.3

[65] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. *The Star Schema Benchmark and Augmented Fact Table Indexing*, page 237–252. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 9783642104237. URL `https://doi.org/10.1007/978-3-642-10424-4_17`. 1, 4.1

[66] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 1493–1508, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2747645. URL `https://doi.org/10.1145/2723372.2747645`. 1, 1.1.1, 5.3

[67] Iraklis Psaroudakis, Tobias Scheuer, Norman May, and Anastasia Ailamaki. Task scheduling for highly concurrent analytical and transactional main memory workloads. 01 2013. 5.1

[68] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Scaling up concurrent main-memory column-store scans: Towards adaptive numa-aware data and task placement. *Proc. VLDB Endow.*, 8(12):1442–1453, aug 2015. ISSN 2150-8097. doi: 10.14778/2824032.2824043. URL `https://doi.org/10.14778/2824032.2824043`. 1.2.1, 3, 1, 5.2

[69] Vijayshankar Raman, Gopi Attaluri, Ronald Barber, Naresh Chainani, David Kalmuk, Vincent KulandaiSamy, Jens Leenstra, Sam Lightstone, Shaorong Liu, Guy M. Lohman, Tim Malkemus, Rene Mueller, Ippokratis Pandis, Berni Schiefer, David Sharpe, Richard Sidle, Adam Storm, and Liping Zhang. Db2 with blu acceleration: So much more than just a column store. *Proc. VLDB Endow.*, 6(11):1080–1091, aug 2013. ISSN 2150-8097. doi: 10.14778/2536222.2536233. URL `https://doi.org/10.14778/2536222.2536233`. 5.3

[70] Linus Schrage. Letter to the editor—a proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968. doi: 10.1287/opre.

16.3.687. URL https://doi.org/10.1287/opre.16.3.687. 1.2.2, 2.2.3

[71] Stefan Schuh, Xiao Chen, and Jens Dittrich. An experimental comparison of thirteen relational equi-joins in main memory. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1961–1976, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450335317. doi: 10.1145/2882903.2882917. URL https://doi.org/10.1145/2882903.2882917. 1, 1.2.1, 1.2.1, 2.1, 5.2

[72] Amir Shaikhha, Yannis Klonatos, and Christoph Koch. Building efficient query engines in a high-level language. *ACM Trans. Database Syst.*, 43(1), apr 2018. ISSN 0362-5915. doi: 10.1145/3183653. URL https://doi.org/10.1145/3183653. 1.1.2

[73] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1617–1632, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367356. doi: 10.1145/3318464.3380595. URL https://doi.org/10.1145/3318464.3380595. 5.3

[74] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012. ISBN 1118063333. 2.3.1

[75] Dimitrios Skarlatos and Kaiyang Zhao. private communication, May 2022. 4.5

[76] *numa(3) perf Manual*. SuSE Labs, 2007. 3.3.1, 4.3

[77] Benjamin Wagner, André Kohn, and Thomas Neumann. Self-tuning query scheduling for analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 1879–1891, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457260. URL https://doi.org/10.1145/3448016.3457260. 2.3.3, 5.1

[78] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. Building a bw-tree takes more than just buzz words. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, page 473–488, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450347037. doi: 10.1145/3183713.3196895. URL https://doi.org/10.1145/3183713.3196895. 1

[79] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. *Proc. VLDB Endow.*, 8(3):209–220, nov 2014. ISSN 2150-8097. doi: 10.14778/2735508.2735511. URL https://doi.org/10.14778/2735508.2735511. 1.2.1

[80] Marcin Zukowski, Niels Nes, and Peter Boncz. Dsm vs. nsm: Cpu performance tradeoffs in block-oriented query processing. In *Proceedings of the 4th International Workshop on Data Management on New Hardware*, DaMoN '08, page 47–54, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605581842. doi: 10.1145/1457150.1457160. URL https://doi.org/10.1145/1457150.1457160. 1

Appendix A: Moving Averages

We present here the moving maximum throughput averages for the spin benchmark for the experiments conducted in Chapter 4.

7.1 NUMA-Aware Allocations with Hybrid Data Layout

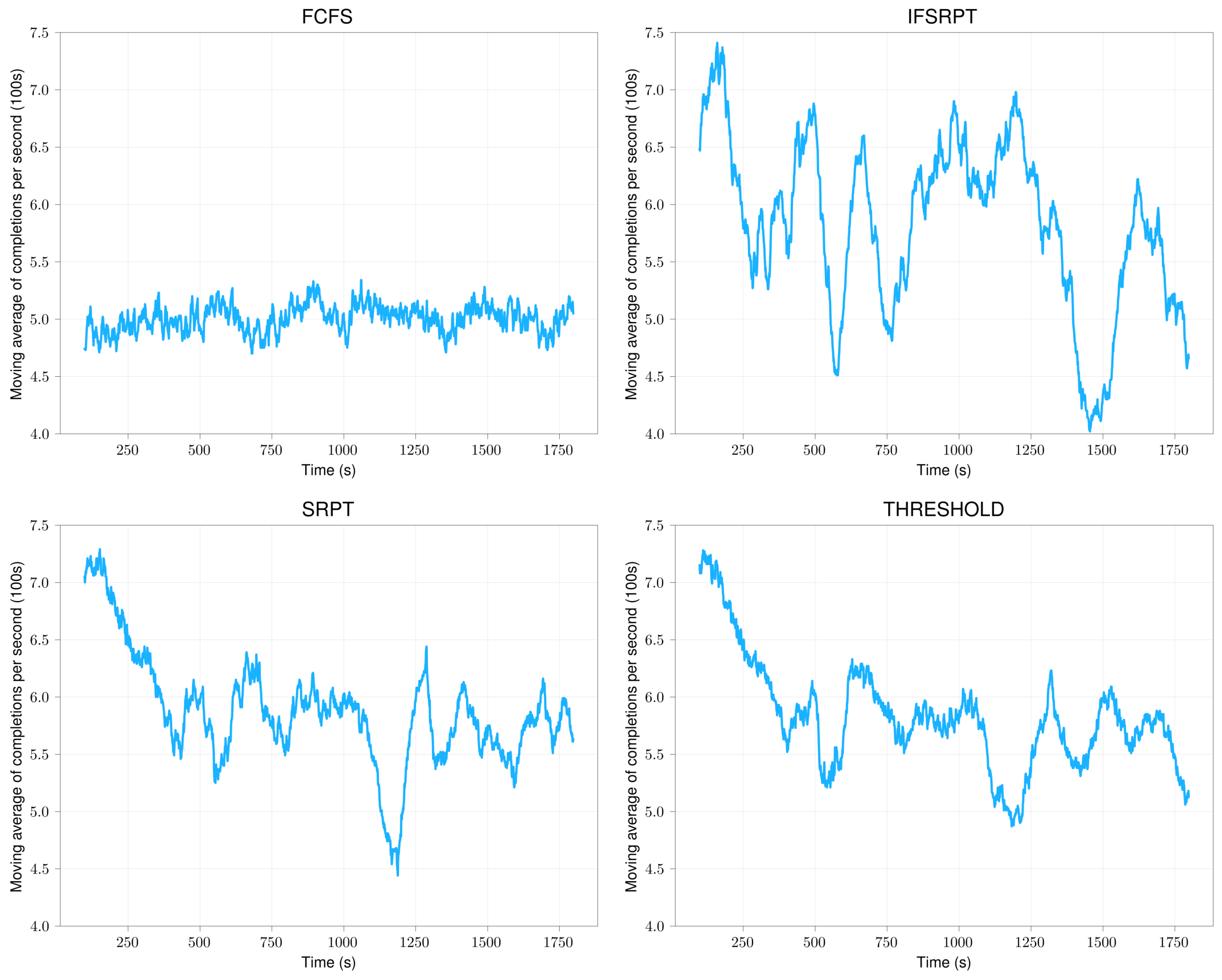


Figure 7.1: The moving averages of completions per second with a 100s window for each of the scheduling policies considered with NUMA-aware data allocation assignments and the use of a PAX memory format in the spin benchmark.

The graphs in Figure 7.1 correspond to the maximum throughput experiments run in Section 4.3. While the range of values on this experiment shifted, the stable regions still hold.

7.2 Increasing Block Sizes

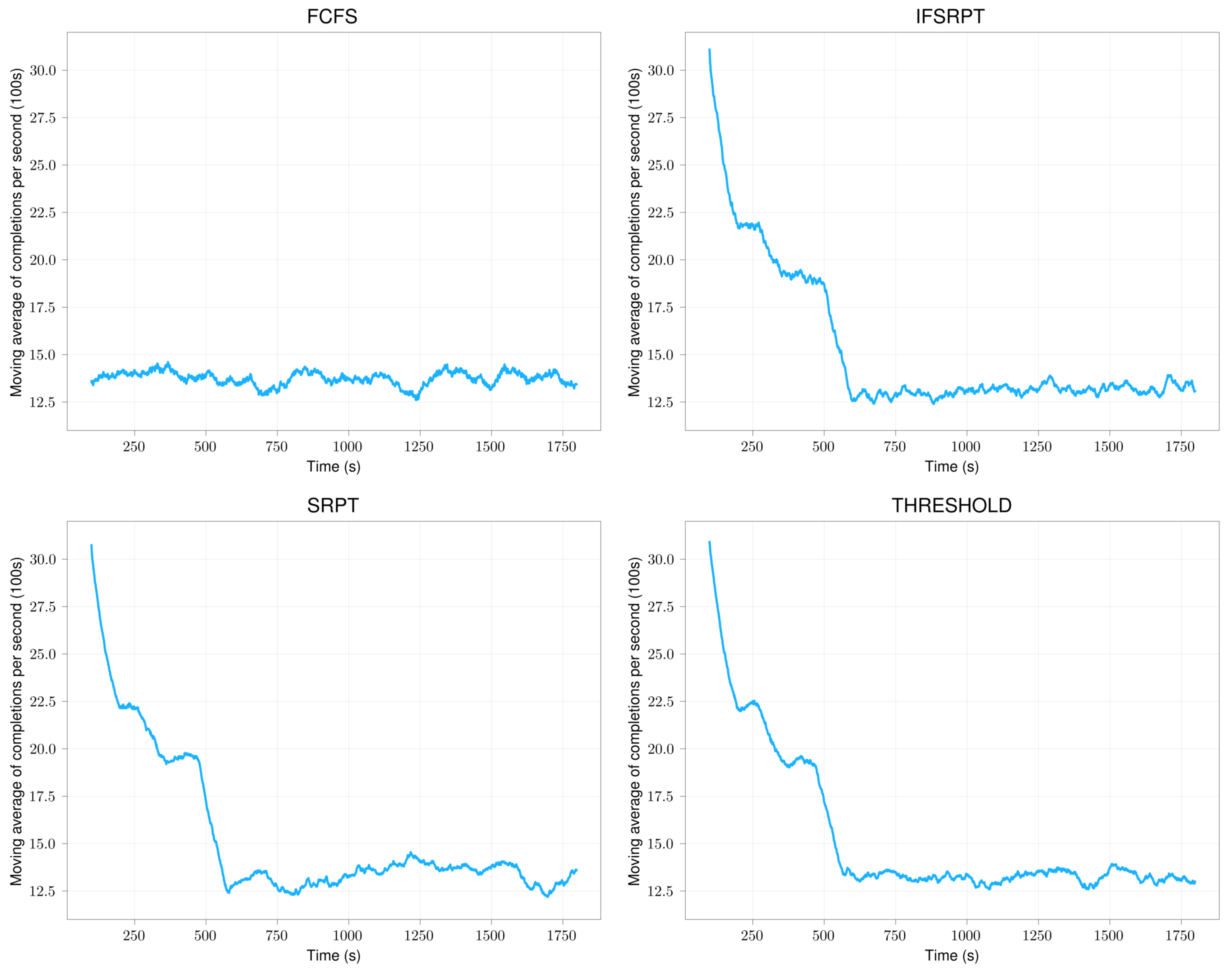


Figure 7.2: The moving averages of completions per second with a 100s window for each of the scheduling policies considered with 2MB hugepage-sized data blocks instead of default 4KB pages in the spin benchmark.

The graphs in Figure 7.2 correspond to the maximum throughput experiments run in Section 4.4.

7.3 Tackling Performance Metrics

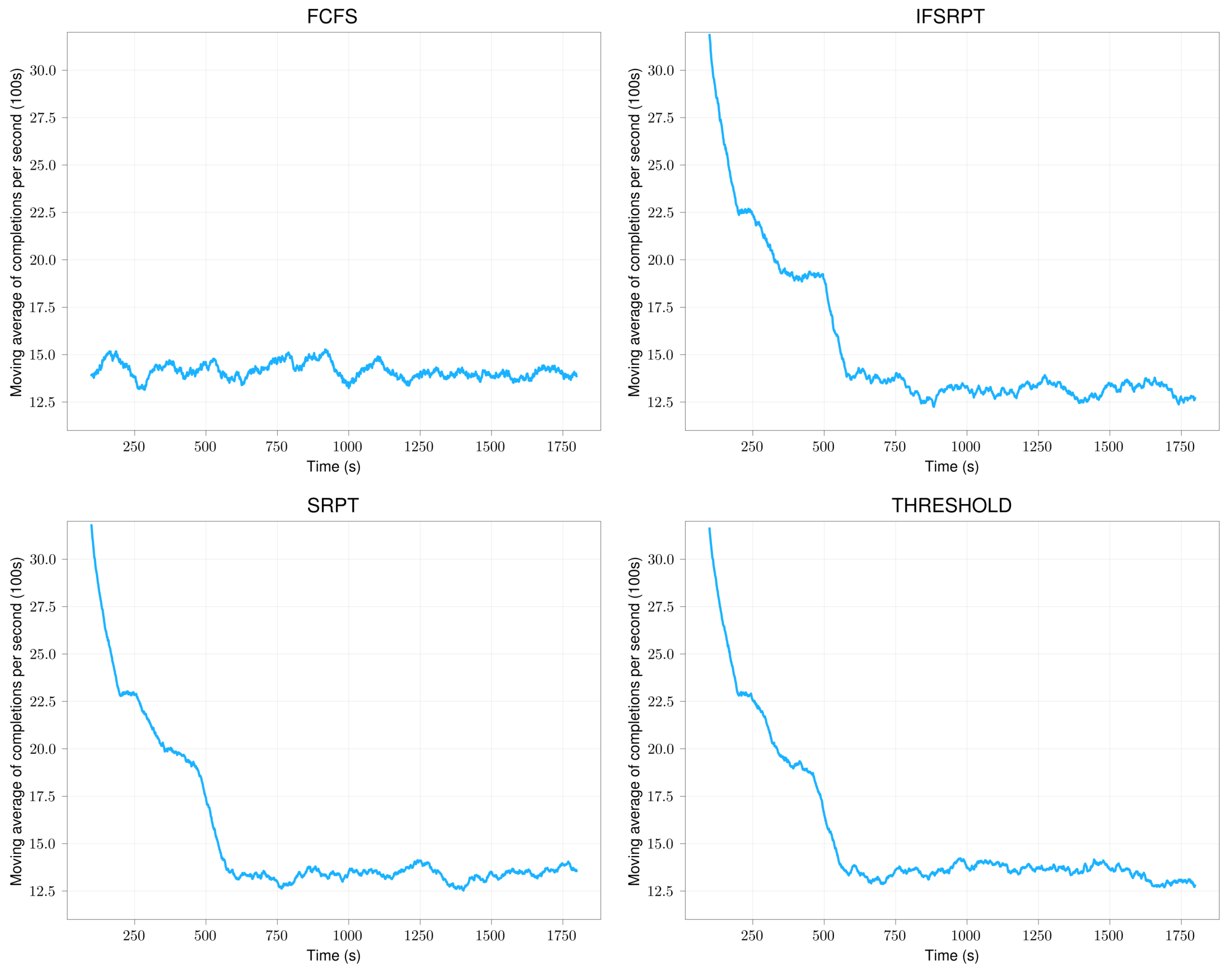


Figure 7.3: The moving averages of completions per second with a 100s window for each of the scheduling policies considered with additional performance mitigations in the spin benchmark.

The graphs in Figure 7.3 correspond to the maximum throughput experiments run in Section 4.5.

7.4 Supporting NUMA-Aware Scheduling

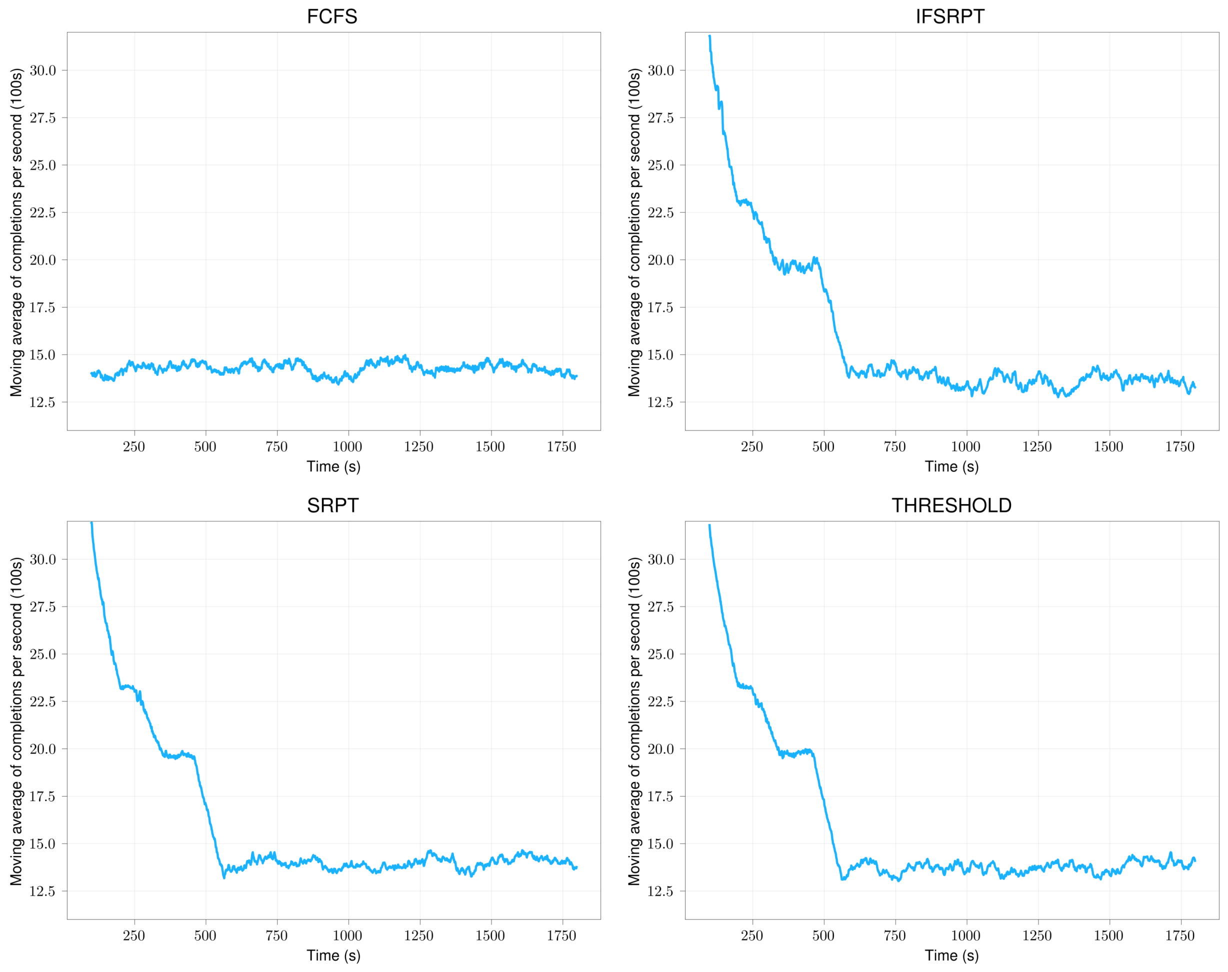


Figure 7.4: The moving averages of completions per second with a 100s window for each of the scheduling policies considered with NUMA-aware queueing in the spin benchmark.

The graphs in Figure 7.4 correspond to the maximum throughput experiments run in Section 4.6.