**EFFICIENT QUERY SCHEDULING**

by

Harshad Deshmukh

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2018

Date of final oral examination: 07/13/2018

The dissertation is approved by the following members of the Final Oral Committee:

Jignesh Patel, Professor, Computer Sciences, UW-Madison
Paraschos Koutris, Assistant Professor, Computer Sciences, UW-Madison
Andrea Arpaci-Dusseau, Professor, Computer Sciences, UW-Madison
Theodoros Rekatsinas, Assistant Professor, Computer Sciences, UW-Madison
Remzi Arpaci-Dusseau, Professor, Computer Sciences, UW-Madison

*To Supriya and my parents*

# ACKNOWLEDGMENTS

This journey has been possible due to so many helping hands in my life. First and foremost, my advisor Prof. Jignesh Patel. Jignesh's infectious energy and his passion for database research have shaped many ideas in my work. I find it fascinating that he can dive into low level implementation details with the same ease as he can paint the big picture. I benefited tremendously by his feedback on my writing and presentation skills. He always prioritized my well being and my family's well being over other things. I hope to emulate his industriousness, discipline and remarkable interpersonal communication skills.

I was fortunate to be in Wisconsin's database group and interact with some astounding professors. Jeff Naughton during his time in Wisconsin was like a father figure. His sharp and witty comments during the talks always made the atmosphere lively. I hope to enjoy Jeff's mentorship in my professional life. I have had numerous interactions with AnHai Doan. I am grateful for his advice and support during my job search. I thank my committee members Professors Paris Koutris, Theodoros Rekatsinas, Andrea Arpaci-Dusseau and Remzi Arpaci-Dusseau for reading my dissertation and for their helpful comments.

I am grateful to the Computer Sciences Graduate Coordinator Angela Thorp for her support in administrative matters.

I could live a debt free grad school life, thanks to the grants from the National Science Foundation. This money essentially comes from the tax dollars of hardworking American men and women. My sincere thanks to all of them.

Quickstep is a group project and is built from the contributions of several students. My work would not have been possible without their collective efforts. I would like to thank Quickstep developers including Craig Chasseur, Qiang Zeng, Shoban Chandrabose, Yinan Li, Gerald, Zuyu Zhang, Jianqiao Zhu, Rathijit Sen, Saket Saurabh, Navneet Potti, Udip Pant, and Quickstep's Apache mentor Julian Hyde.

I got a chance to do multiple internships in grad school. I am grateful to my Samsung internship colleagues Chang-Ho Choi and Suraj Waghulde for their support. I learnt a great deal at my internship with Microsoft CISL. Avrilia Floratou was a great mentor and I learnt from her the *modus operandi* of industrial research. Ashvin Agrawal was a great collaborator who taught me a great deal about rapid software prototyping. Prajakta Kalmegh and I became good friends through car pool.

Life would have been very difficult without some great friends in the database group. Yash Govind has been a great friend and confidant. Shaleen Deep was the sounding board for a lot of my silly ideas. Adalbert Gerald Soosai Raj has provided immense support through good times and bad. I developed close friendship with Hakan Memisoglu during

his time in Wisconsin. Whenever I needed help, Rogers Jeffrey Leo John would always come along with his (poor?) jokes. I learnt a great deal from the long interactions with Bruhathi on the pipelining project, as well as during my job search. Pradap Konda Venkatraman's mature demeanor and deep insights had a calming effect on me. I thank Adel Ardalan for his help on my various presentations. Paul Suganthan GC helped me find accommodation during my first summer in Wisconsin. I had a great time working with Vinitha Reddy on many course projects.

Through the years, I had the company of some wonderful office mates. They often accompanied me to Peet's for coffee breaks and helped me refresh from the work. The long chats with Hakan Memisoglu and Marc Spehlmann made walks to Gordon dining center seem like a breeze. I am grateful to Spyros Blanas for his tidbits of wisdom. An incomplete list of other officemates include Sriram Sridharan, Yueh-Hsuan Chiang, Lalitha Viswanathan, Han Li, and Dylan Bacon.

I got to know several bright and kind people in the UW over the years. Prashant Saxena was my roommate for the first two years and his sense of humor would always lighten up the atmosphere. I had several insightful discussions with Prashant over the delicious dinners he cooked. I thoroughly relished Ankit Agrawal's company during his Master's. Ankit and I could talk for hours about anything under the sun. I still remember Friday evening tea parties with Ankit and Nishant Agrawal. Sanketh Nalli remains a good friend till date. Meenakshi Syamkumar has become a close family friend. I am grateful to Ramnatthan Alagappan and Aishwarya Ganesan for their company for dinners and fun outings. An incomplete list of friends in the UW include Rohit Bhat, Junaid Khalid, Ashutosh Pandey, Swapnil Haria, Amrita Roy Chowdhury, Neha Godwal, Kushagra Garg, and Sakshi Gupta.

My under-graduation institute BITS Pilani has a special place in my life. Many friendships from BITS have accompanied me from India to the US. I have had many trips in grad school with these friends which helped me refresh.

Soumyadeep Ghosh has been a close friend ever since under-graduation days and till date he's my goto person for venting out frustration. I am grateful to Aditya Vijay's great advice in so many situations. Ankur Gupta provided great company on various trips that we went on together. Ameya Zambre was the first familiar face I saw after entering the USA. I am greateful to Nilesh Deshpande, Jhinak Sen, Vineet Pandey, Apoorv Bapat, Akash Badave, Aditya Belsare, Aditya Sharma, Ashirvad Singh, Chanchal Batra, Abhishek Jain, Subhro Sarkar, Priyanka Gupta and Pooja Chandra.

Living in Madison would be dull without a fantastic group of friends. I met a lot of people through Maharashtra Mandal in Madison. Thanks to all of them for the joyful cultural events and parties.

Family is the ultimate support system. My in laws Anant and Neha Hirurkar always supported my pursuit of PhD. My sisters Sukhada and Sampada and their husbands Akshay and Hitesh encouraged me to follow my aspirations and took great care of my parents. My nephews and niece provided great joy during my trips to India.

I dearly miss my late maternal grandmother. She was a single mother and a primary school teacher. She raised six kids on a meager income. Despite the hardships, she had a staunch belief in the power of education. She has a profound influence on my life.

I can probably never do enough to repay what my parents have done for me. They made several sacrifices to ensure that I got a high quality education. They always believed in my decisions and supported me in tough times. Whatever little success that I have achieved is a testimony to their strong foundational values. Thank you Aai and Baba from the bottom of my heart.

I got the gift of fatherhood during grad school. Being a father to an adorable little girl felt daunting at time, but ultimately was a very rewarding feeling. All the stress from working in grad school would go away with a single smile of my daughter.

Last, but by no means the least, I would like to thank my wonderful wife Supriya. She always believed in me, even when I doubted myself. In bad times or good, she always kept assuring me that I was capable of making it to the finish line. She went to such extremes that I was not allowed to crack jokes on myself! She sacrificed her professional life and single handedly managed our family life so that I could focus on my PhD. Supriya, you made all the hardships worthwhile!

# TABLE OF CONTENTS

Page

**LIST OF TABLES** . . . viii

**LIST OF FIGURES** . . . ix

**ABSTRACT** . . . xii

**1 Introduction** . . . 1

1.1 Modern Database Systems and the need for Scheduling . . . 2
1.2 Scheduling: Databases and MapReduce . . . 5

**2 Design and Implementation of Quickstep System** . . . 8

2.1 Introduction . . . 8
2.2 Quickstep Architecture . . . 12
2.2.1 Query Language and Data Model . . . 12
2.2.2 System Overview . . . 12
2.3 Storage Manager . . . 14
2.3.1 Block-Structured Storage . . . 14
2.3.2 Compression . . . 15
2.3.3 Template Metaprogramming . . . 15
2.3.4 Holistic Memory Management . . . 17
2.4 Scheduling & Execution . . . 18
2.4.1 Threading Model . . . 19
2.4.2 Work Order-based Scheduler . . . 19
2.4.3 Separation of Policy and Mechanism . . . 23
2.5 Efficient Query Processing . . . 24
2.5.1 Partial Predicate Push-down . . . 25
2.5.2 Exact Filters: Join to Semi-join Transformation . . . 26
2.5.3 Lookahead Information Passing (LIP) . . . 27
2.6 Evaluation . . . 28
2.6.1 Workload . . . 29
2.6.2 System Configuration . . . 30
2.6.3 System Tuning . . . 30
2.6.4 TPC-H at Scale Factor 100 . . . 31

Page

2.6.5 Denormalizing for higher performance . . . 34
2.6.6 Impact of Row Store vs. Column Store . . . 36
2.6.7 Template Metaprogramming Impact . . . 37
2.6.8 Impact of Optimization Techniques . . . 38
2.6.9 Elasticity . . . 39
2.6.10 Built-in Query Progress Monitoring . . . 40
2.7 Related Work . . . 40
2.8 Conclusions & Future Work . . . 43

**3 Design and Implementation of Quickstep Scheduler** . . . 44

3.1 Introduction . . . 44
3.2 Background . . . 46
3.3 Storage Management in Quickstep . . . 47
3.4 Work Orders . . . 48
3.5 Design of the Scheduler . . . 49
3.5.1 Scheduler Architecture Overview . . . 49
3.6 Thread Model . . . 51
3.6.1 Policy Enforcer . . . 52
3.6.2 Learning Agent . . . 52
3.7 Policy Derivations and Load Controller Implementation . . . 54
3.7.1 Fair Policy Implementation . . . 55
3.7.2 Highest Priority First (HPF) Implementation . . . 56
3.7.3 Proportional Priority (PP) Implementation . . . 57
3.7.4 Load Control Mechanism . . . 57
3.8 Evaluation . . . 59
3.8.1 Quickstep Specifications . . . 60
3.8.2 Experimental Workload . . . 60
3.8.3 Evaluation of Policies . . . 61
3.8.4 Proportional Priority Policy . . . 63
3.8.5 Impact of Learning on the Relative CPU Utilization . . . 65
3.8.6 Impact of Learning on Performance . . . 66
3.8.7 Experiment with Skewed and Uniform Data . . . 67
3.8.8 Load Controller . . . 68
3.9 Related Work . . . 69

**4 Revisiting Pipelining for In-memory Database Systems** . . . 71

4.1 Introduction . . . 71
4.2 Pipelining Background and Related Work . . . 75

Page

4.3 Quickstep Background . . . 77
4.3.1 Managing Storage in Quickstep . . . 78
4.3.2 Pipelining Implementation in Quickstep . . . 78
4.3.3 Pipelining and Scheduling . . . 79
4.4 Discussion on Dimensions . . . 80
4.4.1 Storage Format . . . 80
4.4.2 Block Size . . . 81
4.4.3 Parallelism . . . 81
4.4.4 Hardware Prefetching . . . 83
4.4.5 Pipeline Sequences . . . 83
4.5 Analytical Model . . . 85
4.5.1 Difference between strategies for large blocks . . . 88
4.5.2 Applying the Model to Disk Setting . . . 89
4.5.3 Micro-benchmarking . . . 89
4.6 Experimental Evaluation . . . 90
4.6.1 Workload . . . 91
4.6.2 Hardware Description . . . 91
4.6.3 Results . . . 92
4.6.4 Effect of Pipeline Sequences . . . 100
4.7 Summary of Experiments . . . 103

**5 Conclusions and Future Work** . . . 105

5.1 Lessons Learnt . . . 105
5.1.1 Importance of Abstraction and Contracts . . . 105
5.1.2 Importance of Query Scheduler for High Performance . . . 106
5.1.3 Importance of Measurement, Estimation, and Analysis . . . 106
5.2 Conclusions . . . 107
5.2.1 Future Work . . . 108

**Bibliography** . . . 110

**APPENDIX** . . . 120

# LIST OF TABLES

Table Page

2.1 **Types in Quickstep** . . . . . 12

3.1 **Interpretations of the policies implemented in Quickstep** . . . . . 55

3.2 **Description of notations** . . . . . 56

3.3 **Evaluation Platform** . . . . . 59

4.1 **Notations used for the analytical model** . . . . . 86

4.2 **Evaluation platform** . . . . . 91

# LIST OF FIGURES

Figure Page

1.1 **Deficit - Widening gap between hardware evolution and data generation** . 3

2.1 **A waterfall chart showing the impact of various techniques in Quickstep for query 10 from the TPC-H benchmark running on a 100 scale factor database. RS (Row Store), and CCS (Compressed Column Store) are both supported in Quickstep (see Section 2.3.1). Basic and Selection are template metaprogramming optimizations (described in Section 2.3.3), which relate to the efficiency of predicate and expression evaluation. LIP (Lookahead Information Passing, described in Section 2.5.3) is a technique to improve join performance. Starting with a configuration (Basic + RS), each technique is introduced one at a time to show the individual impact of each technique on this query.** . . . . . 10

2.2 **Evaluation of the expression `discount*price`.** . . . . . 16

2.3 **Plan DAG for the sample query** . . . . . 20

2.4 **Query plan variations for SSB Query 4.1** . . . . . 27

2.5 **Comparison with TPC-H, scale factor 100. Q17 and Q20 did not finish on PostgreSQL after an hour.** . . . . . 32

2.6 **Comparison with denormalized SSB, scale factor 50.** . . . . . 34

2.7 **Impact of storage format on performance for SSB scale factor 100** . . . . . 35

2.8 **Impact of template metaprogramming: Changes in total time and individual query times** . . . . . 37

2.9 **Impact of Exact Filter and LIP using SSB at scale factor 100.** . . . . . 38

Figure Page

2.10 **Prioritized query execution. QX.Y(1) indicates that Query X.Y has a priority 1. Q4.2 and Q4.3 have higher priority (2) than the other queries (1).** . . . . . 39

2.11 **Query progress status. Green nodes (0-5) indicate work that is completed, the yellow node (6) corresponds to operators whose work-orders are currently being executed, and the blue nodes (7-13) show the work that has yet to be started.** . . . 41

3.1 **Overview of the scheduler** . . . . . 50

3.2 **Interactions among scheduler components** . . . . . 54

3.3 **CPU utilization of queries in fair policy** . . . . . 60

3.4 **Standard deviation of CPU utilization of active queries for fair policy** . . . 61

3.5 **CPU utilization in HPF policy. Note that $a.b(N)$ denotes a SSB query $a.b$ with priority $N$** . . . . . 62

3.6 **CPU allocation for proportional priority policy. Note that $a.b(N)$ denotes a SSB query $a.b$ with priority $N$** . . . . . 64

3.7 **Ratio of CPU utilizations $\frac{Q4.1}{Q1.1}$ with and without learning implementation** . 65

3.8 **Impact of learning on the throughput** . . . . . 66

3.9 **Comparison of predicted and observed time per work order** . . . . . 67

3.10 **Load control: An SSB query $a.b$ with priority $N$ is denoted as $a.b(N)$** . . . . 68

4.1 **Interplay between scheduling strategies and pipelining behavior. A sample pipeline of a filter operator ($\sigma$) and a probe operator (P) for a hash join is shown on the left. On the right are two possible interleaving of the work orders of these two operators, resulting in pipelining and non-pipelining schedules** . . . . . 79

4.2 **Results of the micro-benchmarking experiments. The size of the base table is 8 GB.** . . . . . 90

4.3 **Performance comparison of probe hash operator when it is the first consumer operator in a pipeline** . . . . . 93

Figure Page

4.4 **Execution times of operator pipelines in TPC-H query plans using pipelining and non-pipelining strategies with block size 2 MB** . . . . . 94

4.5 **Execution times of TPC-H queries with base tables in column store format and block size 2 MB** . . . . . 95

4.6 **Distribution of time spent in each TPC-H query among its operators** . . . . 96

4.7 **Execution times of TPC-H queries with base tables using row store format and block size of 2 MB** . . . . . 97

4.8 **Scalability of two different probe operators in TPC-H Q07, compared with the ideal scalability. Also shown are the execution times for the probe operator with poor scalability in various settings** . . . . . 98

4.9 **Performance of row store format with and without hardware prefetching. Shown above are total TPC-H execution times and median work order execution times for selection, probe and build hash operators in Q07 with various block sizes** . . . . . 99

4.10 **Comparison of query performances that use two different pipeline sequences. MPS refers to the pipeline sequence output by the MPS algorithm proposed in Section 4.4.5 and another sequence in which probe input is cold, but the hash table is hot at the time of first probe** . . . . . 101

4.11 **Probe performance in cold probe sequence vs MPS algorithm output** . . . . 102

A.1 **A join query and its DAG** . . . . . 122

A.2 **Time per work order for SSB Q1.1 and Q4.1** . . . . . 125

A.3 **Scheduling queries having different work order execution times for the fair policy. The solid boxes with $Q_i$ depicts the lifetime of a work order from the Query $Q_i$** . . . . . 128

# ABSTRACT

Analytical database systems process data to find insights. Database systems today are facing challenges from multiple fronts: First, there is an unprecedented data being generated by machines and humans. Second, there is a diverse demand for analytics which includes data warehousing and predictive analytics. Third, the hardware landscape keeps evolving which means the database system need to keep up in order to reap the best out of the hardware. Finally, cloud computing challenges the many assumptions regarding deployment environments, availability of compute resources, storage and network resources that the traditional systems were built with. Given these challenges, modern database systems need to manage the resources at their disposal *efficiently* - to produce faster results, *effectively* - to fully utilize the resources and *transparently* - to be accountable to its users.

We present a query scheduler designed for the Quickstep database system that imparts efficiency, effectiveness and transparency to the resource management of the system.

In the first chapter we present the makeup of the Quickstep system. We describe a task abstraction called *work orders*, which is used by the scheduler for co-ordinating the execution of a query. Through work orders, Quickstep scheduler gets a fine grained control over query execution, and thus allows the system to exploit large amounts of parallelism offered by the modern hardware.

Next we showcase the usefulness of the Quickstep scheduler as it solves two challenges: resource governance and performance. In the second chapter, we employ the Quickstep scheduler for allocating resources such as CPU to concurrent users of the system. The

resource allocation can understand high level policies such as fair and priority-based, and enforces them while allocating the resources. The scheduler has a learning component, which constantly monitors the resource consumption happening in the system, anticipates the future resource requirements and adjusts the resource allocation so as to meet the policy goals. Our experiments demonstrate that we are able to meet these policy goals.

In the final chapter, we highlight the impact of scheduling techniques on query performance. We turn to pipelining, which is a well known query processing technique, used since the times of disk-based systems. We revisit the importance of pipelining in the in-memory systems and examine the role of various parameters such as block size, parallelism, storage format and hardware prefetching. We find that the performance of queries with and without pipelining does not differ as radically as it does in the disk-based environment. We explain the reasons for this similarity of performance between strategies through empirical evaluation as well as an analytical model. Our study points to a reduced role of pipelining in the in-memory environments for systems using a block-based query processing approach.

# Chapter 1

# Introduction

Computer programs need computing, storage and communication resources to function. Usually the availability of such resources is limited and the system has to share them among multiple users. Resource management is an important aspect of running a system. Scheduling falls under the umbrella of resource management. It can be viewed as a governance model that governs how resources are shared among users and how various parts of the system function together. The governance model is designed so as to meet goals about resource management and performance of the system. For example, an operating system uses resources such as memory, CPUs, and disk. It allows multiple processes to use these resources at the same time through sophisticated scheduling techniques.

Scheduling has been studied extensively in many communities including computer science theory, computer systems, and industrial engineering [25, 31, 32, 54, 70, 78, 101, 123, 125]. Typically a scheduling problems involves finding a schedule with a certain objective. The desired solution is the one that either meets the objective or gets close to meeting it. Many scheduling problems are difficult in nature. For example, the problem of scheduling $N$ jobs each with a unit execution time on $k$ processors such that the makespan of the schedule is minimized, is NP complete [123]. Some of the standard techniques used to solve scheduling problems are: using domain specific heuristics, dynamic programming, greedy algorithms, and approximation algorithms.

In this dissertation, we establish the role of scheduling in modern database systems. We show that scheduling can play a critical role in the functioning of the database system and can be used to meet various objectives such as resource allocation and query performance.

Before understanding the scheduling problem itself, we dive into the *modern* aspect of modern database systems.

## 1.1 Modern Database Systems and the need for Scheduling

We distinguish traditional database systems and modern database systems based on three aspects which are closely related to the scheduling problem: modern hardware, growing data volumes, and deployment environments. We discuss these aspects below and explain how they are related to the scheduling problem.

**Modern hardware:** Over the past few years hardware has drastically evolved. Modern hardware includes large main memories, Non-Uniform Memory Access (NUMA) patterns and large number of CPU cores. Traditional database architecture comes from a time when memories were smaller, data were mostly resident on disks and multi-core parallelism was uncommon. Therefore modern systems have high availability of two resources: CPU parallelism and memory. Scheduling involves managing resources, therefore we believe that there is a place for schedulers in modern database architecture.

**Growing data volumes:** We are in the era of big data and witnessing tremendous amount of data generation. To process the large amount of data efficiently, data processing engines must effectively leverage all the hardware features. However, what we are experiencing is a growing *deficit* between the pace of hardware performance improvements and the pace that is demanded of data processing kernels to keep up with the growth in data volumes.

Figure 1.1 illustrates this deficit issue by comparing improvements in processor performance (blue line) with the growth rate of data (green line), using the number of pages indexed by Google as an illustrative example. The processor performance improvements presented in Figure 1.1 are measured by the highest reported CINT2006 benchmark result for Intel Xeon chips from [118]. We use the number of pages indexed by Google (using estimates made by [119]) as a proxy for data growth. Figure 1.1 does not show the

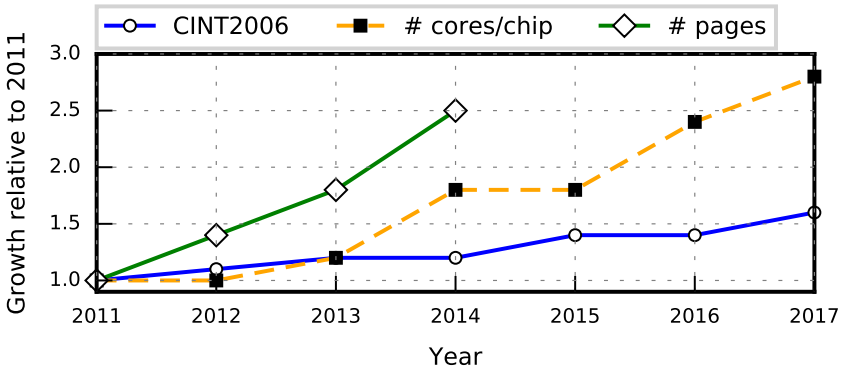



Figure 1.1: **Deficit - Widening gap between hardware evolution and data generation**

increase in the number of queries (which is about 2.5X for Google search queries from 2011–14), and the increase in the complexity of queries as applications request richer analytics. These aspects make the deficit problem worse. The figure also shows the maximum number of cores per chip used in reported CINT2006 results over time. Interestingly (and not shown in the figure), both the minimum and the average amount of memory per chip in the reported CINT2006 results has grown by ≈4X from 2011 to 2017.

The presented data growth rate is conservative for many organizations, which tend to see a far higher rate of increase in the data volume; for example, Facebook's warehouse grew by 3X in 2014 [98]. Figure 1.1 also shows (using a dotted orange line with squares) the growth in the number of cores per processor over time. As one can observe, the number of cores per processor is rising rapidly. In addition, since 2011 the main memory sizes are also growing rapidly, and there is an increasing shift to larger main memory configurations.

In addition to the data growth, there is a growing interest in finding more insights from the data as quickly as possible. In addition to the traditional online analytical processing (OLAP), there is an increased interest in advanced analytics [69] (which involves machine learning algorithms), both from academic community as well as industry. Thus, there is a critical need for in-memory data processing methods that *scale-up* to exploit the full (parallel) processing power that is locked in commodity multi-core servers today.

**Deployment environments:** The most common deployment method few years ago was "on-premise database", in which an enterprise would host and maintain the database software on private servers. Deployment concerns such as failover handling, installing additional capacity, upgrading hardware would be taken care by the in-house database administrator.

Today cloud databases have drastically changed the *modus operandi* of database deployment. Cloud providers host and manage databases on cloud platforms and abstract away all the deployment pain points from the users. The database engine may be hosted on a number of hardware platforms or virtual environments in the cloud.

Cloud databases promise the quality of their service in the form of a Service Layer Agreement (SLA). The most common form of SLA today is in terms of availability, which quantifies the extent to which the cloud database service would be available in a given time window. As the extent of availability increases, the cost to the customer increases. We now discuss the implications of such a SLA in terms of resource management. To meet the SLA, the database service may need to elastically add more nodes. Occasionally to reduce operational expenses and to lower the energy consumption, the database service may need to downsize the number of operational nodes. Some cloud databases provide differentiated service guarantees to users based on the Service Layer Agreement (SLA). To guarantee the SLA for all users, the database service may require to reshuffle resources from one customer to another.

Considering the above challenges, the resource management layer of the cloud database needs to be flexible enough to withstand dynamicity in the resource availability. Hence the scheduler should have built in resource allocation flexibility, helping the resource management of the database become dynamic.

Next we take a deeper look into the scheduling problem in database systems. The scope of our work is relational analytics (warehouse settings) where data primarily resides in memory. In relational analytics, questions on data are posed via *queries*, that are made up of relational operators. To get an idea of the problem of scheduling in database systems,

we compare traditional database systems with MapReduce [36] based systems such as Hive[1] [122] in the next section.

## 1.2 Scheduling: Databases and MapReduce

We focus on two aspects of scheduling to compare databases and MapReduce frameworks: a) an abstraction for tasks b) connecting resource management with relational operators' semantics.

MapReduce is a parallel data processing framework where a job is broken up in several parallel map and reduce tasks. MapReduce programs consists of only map tasks and reduce tasks. The mappers and reducer tasks form the scheduling abstractions for the MapReduce framework.

In databases, a query is made up of several relational operators such as selection, join, aggregation. Each logical operator may have multiple physical implementations. For example an equi-join operator can be performed using hash-join and sort-merge join. Given the variety in terms of relational operators and their physical implementations, databases so far did not have a common scheduling abstraction for tasks within a query. We hypothesize that the lack of a common scheduling abstraction is a reason why scheduler is not considered a first class citizen in database architecture.

By having a scheduling abstraction the database system can have a greater control over how resources are managed, and allotted to its users. The abstraction makes it easy to estimate resource availability. If the scheduling abstraction is amenable to parallelism, it may help the system to leverage parallelism offered by the hardware.

Next we discuss specialized resource management for relational operators. Database community has for long focused on finding specialized resource management techniques to improve performance of either individual relational operators or query as a whole [35, 25, 16, 83, 88]. Such a specialization is difficult to achieve in MapReduce framework, in

[1]We acknowledge that Hive now supports Tez and Spark run time environments in addition to MapReduce.

which every task is either a mapper or a reducer. It is difficult to reason about what is the logical task being performed through the mappers and reducers. For example a mapper could belong to the probe phase of a join or aggregation operator.

Through our work, we would like to achieve the best of both (databases and MapReduce framework) worlds. We want scheduler as a first class primitive in the database architecture and the ability to perform operator centric resource management in query execution. We now present the contributions of this dissertation.

**Design and Implementation of the Quickstep Database System:** The work in this thesis is carried out in the Quickstep Database System [1, 99, 100]. We begin by presenting the design and implementation of Quickstep.

Quickstep targets high performance for in-memory relational analytics. We describe the various components of the system including storage manager, query optimizer, query execution engine (which consists the query scheduler), and the expression evaluation module. Through benchmarking against other systems, we show that Quickstep is faster than other systems, often by an order of magnitude.

**Design and Implementation of the Quickstep Scheduler:** The core focus of this dissertation is Quickstep's query scheduler. We describe the *nuts and bolts* of the scheduler. We propose a novel task abstraction called *work orders* used for query processing in Quickstep. The work done in a query is broken up in several work orders. Thus the work orders abstraction is used by all the relational operators implemented in Quickstep and it enables the system to exploit the large degree of parallelism offered by modern hardware.

**Using the Quickstep Scheduler to meet different objectives:** We showcase the versatility of the Quickstep scheduler by demonstrating how it can be used to meet various objectives. Specifically we show the Quickstep scheduler in two scenarios. First, we show how the scheduler can allocate resources such as CPU to concurrent queries through well defined policies such as fair and priority-based policies. We propose a probabilistic

scheduling approach, which gives the resource administrators a knob to allocate resources to concurrently running queries in the system. We believe that this work can be useful in the age of cloud databases, where the *pay as you go* philosophy for resource consumption can become the norm.

Second, we analyze the impact of scheduling strategies on query performance. For this analysis we focus on *pipelining*, a well known technique in database systems that deals with the interaction of two adjacent relational operators in a query plan DAG. Disk-based database systems prefer pipelined query processing over non-pipelining query processing. We question this preference in the in-memory based systems and show that the gap between pipelining and non-pipelining is not as wide. We study the connection between pipelining performance and parameters such as degree of parallelism, storage formats, block size and hardware prefetching. We compare the performance of TPC-H queries with and without pipelining. We come up with a surprising conclusion that pipelining does not really help query performance significantly. We show the reasons for why these two strategies result in a similar performance through an analytical model and through extensive empirical evaluation on Quickstep.

Readers are advised that the content in this dissertation is based on previously published papers [37, 38, 100] and another manuscript under submission. Large sections of this dissertation are taken directly from these papers. Quickstep is an open source project. Interested readers can find its source code at: `https://github.com/apache/incubator-quickstep`.

# Chapter 2

# Design and Implementation of Quickstep System

## 2.1 Introduction

Query processing systems today face a host of challenges that were not as prominent just a few years ago. A key change has been dramatic changes in the hardware landscape that is driven by the need to consider energy as a first-class (hardware) design parameter. Across the entire processor-IO hierarchy, the hardware paradigm today looks very different than it did just a few years ago. Consequently, we are now experiencing a growing *deficit* (illustrated in Figure 1.1) between the pace of hardware performance improvements and the pace that is demanded of data processing kernels to keep up with the growth in data volumes.

Our attempt to pay off this deficit is through a system called Quickstep. Our goal is to exploit the full (parallel) processing power that is locked in commodity multi-core servers today. We describe the initial version of Quickstep that targets single-node in-memory read-mostly analytic workloads. Quickstep uses mechanisms that allow for *high intra-operator parallelism*. Such mechanisms are critical to exploit the full potential of the high level of hardware compute parallelism that is present in modern servers (the dotted orange line in Figure 1.1).

Unlike most research database management systems (DBMSs), Quickstep has a storage manager with a block layout, where each block behaves like a mini self-contained database [27]. This "independent" block-based storage design is leveraged by a highly parallelizable query execution paradigm in which independent *work orders* are generated

at the block level. Query execution then amounts to creating and scheduling work orders, which can be done in a generic way. Thus, the scheduler is a crucial system component, and the Quickstep scheduler cleanly separates scheduling policies from the underlying scheduling mechanisms. This separation allows the system to elastically scale the resources that are allocated to queries, and to adjust the resource allocations dynamically to meet various policy-related goals.

Recognizing that random memory access patterns and materialization costs often dominate the execution time in main-memory DBMSs, Quickstep uses a number of query processing techniques that take the "drop early, drop fast" approach: eliminating redundant rows as early as possible, as fast as possible. For instance, Quickstep aggressively pushes down complex disjunctive predicates involving multiple tables using a predicate over-approximation scheme. Quickstep also uses cache-efficient filter data structures to pass information across primary key-foreign key equi-joins, eliminating semi-joins entirely in some cases. Overall, the key contributions of our work are as follows:

**Cohesive collection of techniques:** We present the first end-to-end design for Quickstep. This system brings together in a single artifact a number of mechanisms for in-memory query processing such as support for multiple storage formats, a template metaprogramming approach to both manage the software complexity associated with supporting multiple storage formats and to evaluate expressions on data in each storage format efficiently, and novel query optimization techniques. The impact of each mechanism depends on the workload, and our system brings these mechanisms together as a whole. For example, the waterfall chart in Figure 2.1 shows the contributions of various techniques on the performance of TPC-H Query 10.

**Novel query processing techniques:** We present Quickstep's use of techniques to aggressively push down complex disjunctive predicates involving multiple relations, as well as to eliminate certain types of equi-joins using *exact filters*.

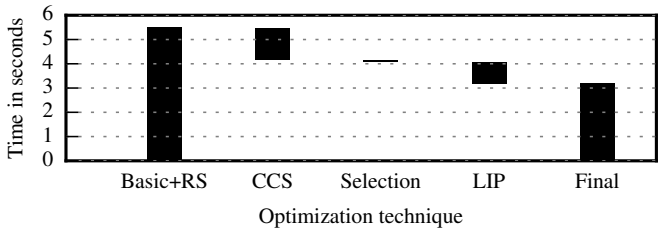



Figure 2.1: **A waterfall chart showing the impact of various techniques in Quickstep for query 10 from the TPC-H benchmark running on a 100 scale factor database. RS (Row Store), and CCS (Compressed Column Store) are both supported in Quickstep (see Section 2.3.1). Basic and Selection are template metaprogramming optimizations (described in Section 2.3.3), which relate to the efficiency of predicate and expression evaluation. LIP (Lookahead Information Passing, described in Section 2.5.3) is a technique to improve join performance. Starting with a configuration (Basic + RS), each technique is introduced one at a time to show the individual impact of each technique on this query.**

**Manageability:** The design of the system focuses on ease-of-use, paying attention to a number of issues, including employing methods such as using a holistic approach to memory management, and elastically scaling query resource usage at runtime to gracefully deal with concurrent queries with varying query priorities.

**Comparison with other systems:** We also conduct an end-to-end evaluation comparing Quickstep with a number of other systems. These system are: Spark [133, 15], PostgreSQL [103], MonetDB [60], and VectorWise [137]. Our results show that in many cases, Quickstep is faster by an order-of-magnitude, or more.

We also leverage the multiple different storage implementations in Quickstep to better understand the end-to-end impact of the popular row store and column store methods on the SSB and TPC-H queries. To the best of our knowledge, an apples-to-apples comparison of these benchmark queries does not exist. We show that overall column stores are still preferred, though the speed up overall is only about 2X. Earlier comparisons, e.g. [12], have been indirect comparisons of this aspect of storage management for the SSB benchmark across two different systems, and show far larger (6X) improvements.

**Open source:** Quickstep is available as open-source, which we hope helps the reproducibility goal that is being pursued in our community [24, 81, 82]. It also allows other researchers to use this system as a platform when working on problems where the impact of specific techniques can be best studied within the context of the overall system behavior.

The remainder of this chapter is organized as follows: The overall Quickstep architecture is presented in the next section. The storage manager in presented in Section 2.3. The query execution and scheduling methods are presented in Sections 2.4 and 2.5 respectively. Empirical results are presented in Section 2.6, and related work is presented in Section 2.7. Finally, Section 2.8 contains our concluding remarks.

| | |
|---|---|
| **Numeric types** | `INTEGER` (32-bit signed)<br>`BIGINT/LONG` (64-bit signed)<br>`REAL/FLOAT` (IEEE 754 *binary32* format)<br>`DOUBLE PRECISION` (IEEE 754 *binary64* format)<br>fixed-point `DECIMAL` |
| **Non-numeric types** | `CHAR` strings<br>variable-length `VARCHAR` strings<br>`DATETIME/TIMESTAMP` (microsecond resolution)<br>Date-time `INTERVAL`<br>Year-month `INTERVAL` |

Table 2.1: **Types in Quickstep**

## 2.2 Quickstep Architecture

Quickstep implements a collection of relational algebraic operators, using efficient algorithms for each operation. This "kernel" can be used to run a variety of applications, including SQL-based data analytics (the focus of our work) and other classes of analytics/-machine learning (using the approach outlined in [44, 134]). Our work only focuses only on SQL analytics.

### 2.2.1 Query Language and Data Model

Quickstep uses a relational data model, and SQL as its query language. Table 2.1 describes the various types currently supported by Quickstep.

### 2.2.2 System Overview

The internal architecture of Quickstep resembles the architecture of a typical DBMS engine. A distinguishing aspect is that Quickstep has a query scheduler (cf. Section 2.4.2),

which plays a key first-class role allowing for quick reaction to changing workload management (see evaluation in Section 2.6.9). A SQL *parser* converts the input query into a syntax tree, which is then transformed by an optimizer into a physical plan. The *optimizer* uses a rules-based approach [48] to transform the logical plan into an optimal physical plan. The current optimizer supports projection and selection push-down, and both bushy and left-deep trees.

A *catalog manager* stores the logical and physical schema information, and associated statistics, including table cardinalities, the number of distinct values for each attribute, and the minimum and maximum values for numerical attributes.

A *storage manager* organizes the data into large multi-MB blocks, and is described in Section 2.3.

An execution plan in Quickstep is a directed acyclic graph (DAG) of relational operators. The execution plan is created by the optimizer, and then sent to the *scheduler*. The scheduler is described in Section 2.4.

A *relational operator library* contains implementation of various relational operators. Currently, the system has implementations for the following operators: select, project, joins (equijoin, semijoin, antijoin and outerjoin), aggregate, sort, and top-k.

Quickstep implements a hash join algorithm in which the two phases, the build phase and the probe phase, are implemented as separate operators. The build hash table operator reads blocks of the build relation, and builds a single cache-efficient hash table in memory using the join predicate as the key (using the method proposed in [19]). The probe hash table operator reads blocks of the probe relation, probes the hash table, and materializes joined tuples into in-memory blocks. Both the build and probe operators take advantage of block-level parallelism, and use a latch-free concurrent hash table to allow multiple workers to proceed at the same time.

For non-equijoins, a block-nested loops join algorithm is used. The hash join method has also been adapted to support left outer join, left semijoin, and antijoin operations.

For aggregation without `GROUP BY`, local aggregates for each input block are computed, which are then merged to compute the global aggregate. For aggregation with `GROUP BY`, a global latch-free hash table of aggregation handles is built (in parallel), using the grouping columns as the key.

The sort and top-K operators use a two-phase algorithm. In the first phase, each block of the input relation is sorted in-place, or copied to a single temporary sorted block. These sorted blocks are merged in the second (final) phase.

## 2.3 Storage Manager

The Quickstep storage manager [27] is based on a block-based architecture, which we describe next. The storage manager allows a variety of physical data organizations to coexist within the same database, and even within the same table. We briefly outline the block-based storage next.

### 2.3.1 Block-Structured Storage

Storage for a particular table in Quickstep is divided into many blocks with possibly different layouts, with individual tuples wholly contained in a single block. Blocks of different sizes are supported, and the default block size is 2 megabytes. On systems that support large virtual-memory pages, Quickstep constrains block sizes to be an exact multiple of the hardware large-page size (e.g. 2 megabytes on x86-64) so that it can allocate buffer pool memory using large pages and make more efficient use of processor TLB entries.

Internally, a block consists of a small *metadata header* (the block's self-description), a single *tuple-storage sub-block* and any number of *index sub-blocks*, all packed in the block's contiguous memory space. There are multiple implementations of both types of sub-blocks, and the API for sub-blocks is generic and extensible, making it easy to add more sub-block types in the future. Both row-stores and column-store formats are supported, and orthogonally these stores can be compressed. See [56] for additional details about the block layouts.

### 2.3.2 Compression

Both row store and column store tuple-storage sub-blocks may optionally be used with compression. Quickstep supports two type-specific order-preserving compression schemes: (1) simple ordered dictionary compression for all data types, and (2) leading zeroes truncation for numeric data types. In addition, Quickstep automatically chooses the most efficient compression for each attribute on a per-block basis.

Dictionary compression converts native column values into short integer codes that compare in the same order as the original values. Depending on the cardinality of values in a particular column within a particular block, such codes may require considerably less storage space than the original values. In a row store, compressed attributes require only 1, 2, or 4 bytes in a single tuple slot. In a column store, an entire column stripe consists only of tightly-packed compressed codes. We note that in the column-store case, we could more aggressively pack codes without "rounding up" to the nearest byte, but our experiments have indicated that the more complicated process of reconstructing codes that span across multiple words slows down scans overall when this technique is used. Thus, we currently pack codes at 1, 2, and 4 byte boundaries.

### 2.3.3 Template Metaprogramming

As noted above, Quickstep supports a variety of data layouts (row vs. column store, and with and without compression). Each operator algorithm (e.g. scan, select, hash-based aggregate, hash-based join, nested loops join) must work with *each* data layout. From a software development perspective, the complexity of the software development for each point in this design space can be quite high. A naive way to manage this complexity is to use inheritance and dynamic dispatch. However, the run-time overhead of such indirection can have disastrous impact on query performance.

To address this problem, Quickstep uses a template metaprogramming approach to allow efficient access to data in the blocks. This approach is inspired by the principle of zero-cost abstractions exemplified by the design of the C++ standard template library

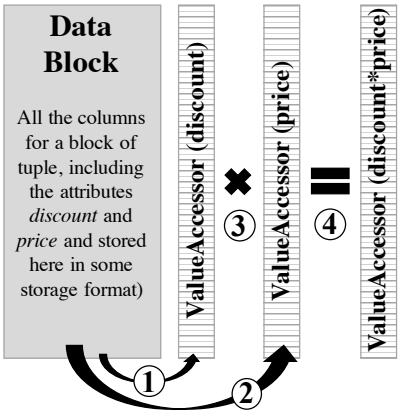



Figure 2.2: **Evaluation of the expression `discount*price`.**

(STL), in which the implementations of containers (such as vectors and maps) and algorithms (like find and sort) are separated from each other.

Quickstep has an analogous design wherein access to data in a sub-block is made via a `ValueAccessor` in combination with a generic functor (usually a short lambda function) that implements the evaluation of some expression or the execution of some operator algorithm. The various ValueAccessors and functors have been designed so that the compiler can easily inline calls and (statically) generate compact and efficient inner loops for expression and operator evaluation described in more detail below in Section 2.3.3.1. Such loops are also amenable to prefetching and SIMD auto-vectorization by the compiler, and potentially (in the future) mappable to data parallel constructs in new hardware. (We acknowledge that there is a complementary role for run-time code generation.) We describe the use of this technique for expression evaluation next.

#### 2.3.3.1 Expression Evaluation

The `ValueAccessor`s (VAs) play a crucial role in efficient evaluation of expressions (e.g. `discount*price`). Figure 2.2 illustrates how VAs work using as example an expression that is the product of two attributes. There are various compile time optimizations that control the code that is generated for VAs. When using the “Basic” optimization, the

VA code makes a physical copy of the attributes that are referenced in the expression. These are steps 1 and 2 in Figure 2.2. The vector of the two attributes are then multiplied (step 3 in the figure) using a loop unrolled by the compiler (possibly generating SIMD instructions). The output of the expression is another VA object, from which (efficient) copies can be made to the final destination (likely a sub-block in a result block).

When using the "Selection" optimization level, the code that is generated for the VAs uses an indirection to the attributes (regardless of the storage format in the block). Thus, in steps 1 and 2 in Figure 2.2, the resulting VA "vectors" contain pointers to the actual attributes. These pointers are dereferenced as needed in step 3. With the Selection optimization, copies are avoided, and if the columns are in a columnar store format the VAs are further compacted to simply point to the start of the "vector" in the actual storage block.

To understand the impact of the template metaprogramming approach we compared the code generated by the template metaprogamming (i.e. VA) approach with a standalone program that uses dynamic dispatch to access the attributes. With the dynamic dispatch option, a traditional `getNext()` interface is used to access the attributes in a uniform way regardless of the underlying storage format. For this comparison, we created a table with two integer attributes, set the table cardinality to 100 million tuples, and stored the data in a columnar store format. Then, we evaluated an expression that added both the integer attributes (on the same 2 socket system described in Section 2.6). The resulting code using the Selection optimization is 3X faster compared to the virtual function approach when using a single thread (when the computation is compute-bound, and drops to 2X when using all the (20) hardware threads when the computation is more memory-bound.

### 2.3.4 Holistic Memory Management

The Quickstep storage manager maintains a *buffer pool* of memory that is used to create blocks, and to load them from persistent storage on-demand. Large allocations of unstructured memory are also made from this buffer pool, and are used for shared run-time data

structures like hash tables for joins and aggregation operations. These large allocations for run-time data structures are called *blobs*. The buffer pool is organized as a collection of slots, and the slots in the buffer pool (either blocks or blobs) are treated like a larger-sized version of page slots in a conventional DBMS buffer pool.

We note that in Quickstep *all* memory for caching base data, temporary tables, and run-time data structures is allocated and managed by the buffer pool manager. This holistic view of memory management implies that the user does not have to worry about how to partition memory for these different components. The buffer pool employs an eviction policy to determine the pages to cache in memory. Quickstep has a mechanism where different "pluggable" eviction policies can be activated to choose how and when blocks are evicted from memory, and (if necessary) written back to persistent storage if the page is "dirty." The default eviction policy is LRU-2 [94].

Data from the storage manager can be persisted through a file manager abstraction that currently supports the Linux file system (default), and also HDFS [117].

## 2.4 Scheduling & Execution

In this section, we describe how the design of the query processing engine in Quickstep achieves three key objectives. First, we believe that separating the control flow and the data flow involved in query processing allows for greater flexibility in reacting to runtime conditions and facilitates maintainability and extensibility of the system. To achieve this objective, the engine separates responsibilities between a scheduler, which makes work scheduling decisions, and workers that execute the data processing kernels (cf. Section 2.4.1).

Second, to fully utilize the high degree of parallelism offered by modern processors, Quickstep complements its block-based storage design with a work order-based scheduling model (cf. Section 2.4.2) to obtain high intra-query and intra-operator parallelism.

Finally, to support diverse scheduling policies for sharing resources (such as CPU and memory) between concurrent queries, the scheduler design separates the choice of policies from the execution mechanisms (cf. Section 2.4.3).

### 2.4.1 Threading Model

The Quickstep execution engine consists of a single *scheduler* thread and a pool of *workers*. The scheduler thread uses the query plan to generate and schedule work for the workers. When multiple queries are concurrently executing in the system, the scheduler is responsible for enforcing resource allocation policies across concurrent queries and controlling query admittance under high load. Furthermore, the scheduler monitors query execution progress, enabling status reports as illustrated in Section 2.6.10.

The workers are responsible for executing the relational operation tasks that are scheduled. Each worker is a single thread that is pinned to a CPU core (possibly a virtual core), and there are as many workers as cores available to Quickstep. The workers are created when the Quickstep process starts, and are kept alive across query executions, minimizing query initialization costs. The workers are stateless; thus, the worker pool can *elastically* grow or shrink dynamically.

### 2.4.2 Work Order-based Scheduler

The Quickstep scheduler divides the work for the entire query into a series of *work orders*. In this section, we first describe the work order abstraction and provide a few example work order types. Next, we explain how the scheduler generates work orders for different relational operators in a query plan, including handling of pipelining and internal memory management during query execution.

The optimizer sends to the scheduler an execution query plan represented as a directed acyclic graph (DAG) in which each node is a relational operator. Figure 2.3 shows the DAG for the example query shown below. Note that the edges in the DAG are annotated with whether the producer operator is blocking or permits pipelining.

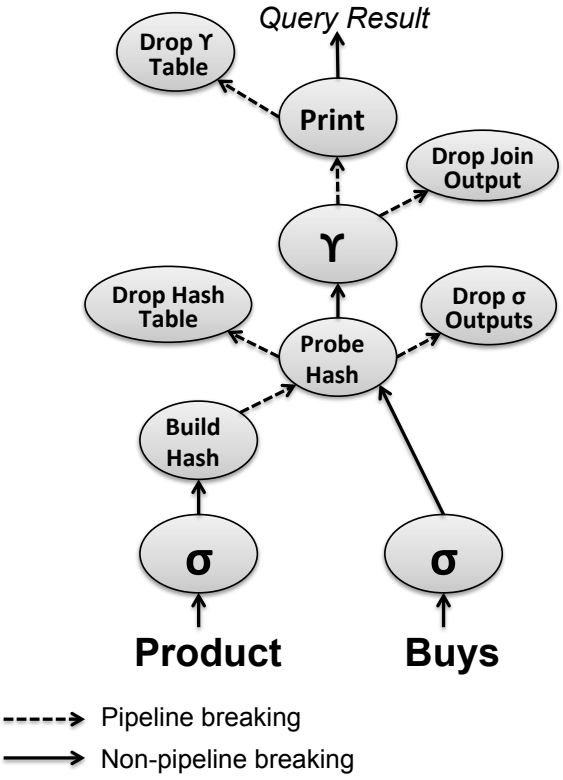



Figure 2.3: **Plan DAG for the sample query**

```
SELECT SUM(sales)
FROM   Product P NATURAL JOIN Buys B
WHERE  B.buy_month = 'March'
AND    P.category = 'swim'
```

#### 2.4.2.1 Work Order

A *work order* is a unit of intra-operator parallelism for a relational operator. Each relational operator in Quickstep describes its work in the form of a set of work orders, which contains references to its inputs and all its parameters. For example, a *selection work order* contains a reference to its input relation, a filtering predicate, and a projection list of attributes (or expressions) as well as a reference to a particular input block. A selection operator generates as many work orders as there are blocks in the input relation. Similarly, a *build hash work order* contains a reference to its input relation, the build key attribute, a hash table reference, and a reference to a single block of the input build relation to insert into the hash table.

#### 2.4.2.2 Work Order Generation and Execution

The scheduler employs a simple DAG traversal algorithm to activate nodes in the DAG. An active node in the DAG can generate *schedulable* work orders, which can be fetched by the scheduler. In the example query, initially, only the Select operators (shown in Figure 2.3 using the symbol $\sigma$) are active. Operators such as the probe hash and the aggregation operations are initially inactive as their blocking dependencies have not finished execution. The scheduler begins executing this query by fetching work orders for the select operators. Later, other operators will become active as their dependencies are met, and the scheduler will fetch work orders from them.

The scheduler assigns these work orders to available workers, which then execute them. All output is written to temporary storage blocks. After executing a work order, the worker

sends a completion message to the scheduler, which includes execution statistics that can be used to analyze the query execution behavior.

#### 2.4.2.3 Implementation of Pipelining

In our example DAG (Figure 2.3), the edge from the Probe hash operator to the Aggregate operator allows for data pipelining. As described earlier, the output of each probe hash work order is written in some temporary blocks. Fully-filled output blocks of probe hash operators can be streamed to the aggregation operator (shown using the symbol $\gamma$ in the figure). The aggregation operator can generate one work order for each streamed input block that it receives from the probe operator, thereby achieving pipelining.

The design of the Quickstep scheduler separates control flow from data flow. The control flow decisions are encapsulated in the work order scheduling policy. This policy can be tuned to achieve different objectives, such as aiming for high performance, staying with a certain level of concurrency/CPU resource consumption for a query, etc. In the current implementation, the scheduler *eagerly* schedules work orders as soon as they are available.

#### 2.4.2.4 Output Management

During query execution, intermediate results are written to temporary blocks. To minimize internal fragmentation and amortize block allocation overhead, workers reuse blocks belonging to the same output relation until they become full. To avoid memory pressure, these intermediate relations are dropped as soon as they have been completely consumed (see the Drop $\sigma$ Outputs operator in the DAG). Hash tables are also freed similarly (see the Drop Hash Table operator). An interesting avenue for future work is to explore whether delaying these Drop operators can allow sub-query reuse across queries.

### 2.4.3 Separation of Policy and Mechanism

Quickstep's scheduler supports concurrent query execution. Recall that a query is decomposed into several work orders during execution. These work orders are organized in a data structure called the *Work Order Container*. The scheduler maintains one such container per query. A *single scheduling decision* involves: selection of a query $\rightarrow$ selection of a work order from the container $\rightarrow$ dispatching the work order to a worker thread. When concurrent queries are present, a key aspect of the scheduling decision is to select a query from the set of active concurrent queries, which we describe next.

The selection of a query is driven by a *high level policy*. An example of such a policy is *Fair*. With this policy, in a given time interval, all active queries get an equal proportion of the total CPU cycles across all the cores. Another such policy is *Highest Priority First* (HPF), which gives preference to higher priority queries. (The HPF policy is illustrated later in Section 2.6.9.) Thus, Quickstep's scheduler consists of a component called the *Policy Enforcer* that transforms the policy specifications in each of the scheduling decisions.

The Policy Enforcer uses a *probabilistic framework* for selecting queries for scheduling decisions. It assigns each query a probability value, which indicates the likelihood of that query being selected in the next scheduling decision. We employ a probabilistic approach because it is attractive from an implementation and debugging perspective (as we only worry about the probability values, which can be adjusted dynamically at anytime, including mid-way through query execution).

The probabilistic framework forms the *mechanism* to realize the high level policies and remains decoupled from the policies. This design is inspired from the classical *separation of policies from mechanism* principle [71].

A key challenge in implementing the Policy Enforcer lies in transforming the policy specifications to probability values, one for each query. A critical piece of information used to determine the probability values is the prediction of the execution time of the future work order for a query. This information provides the Policy Enforcer some insight into the future resource requirements of the queries in the system. The Policy Enforcer

is aware of the current resource allocation to different queries in the system, and using these predictions, it can adjust the future resource allocation with the goal of *enforcing* the specified policy for resource sharing.

The predictions about execution time of future work orders of a query are provided by a component called the *Learning Agent*. It uses a prediction model that takes execution statistics of the past work orders of a query as input and estimates the execution time for the future work orders for the query.

The calculation of the probability values for different policies implemented in Quickstep and their relation with the estimated work order execution time is presented in [37].

To prevent the system from thrashing (e.g. out of memory), a load controller is in-built into the scheduler. During concurrent execution of the queries, the load controller can control the admission of queries into the system and it may suspend resource intensive queries, to ensure resource availability.

Finally, we note that by simply tracking the work orders that are completed, Quickstep can provide a built-in generic query progress monitor (shown in Section 2.6.10).

## 2.5 Efficient Query Processing

Quickstep builds on a number of existing query processing methods (as described in Section 2.2.2). The system also improves on existing methods for specific common query processing patterns. We describe these query processing methods in this section.

Below, we first describe a technique that pushes down certain disjunctive predicates more aggressively than is common in traditional query processing engines. Next, we describe how certain joins can be transformed into cache-efficient semi-joins using *exact filters*. Finally, we describe a technique called *LIP* that uses Bloom filters to speed up the execution of join trees with a star schema pattern.

The unifying theme that underlies these query processing methods is to eliminate redundant computation and materialization using a "drop early, drop fast" approach: aggressively pushing down filters in a query plan to drop redundant rows as early as possible, and using efficient mechanisms to pass and apply such filters to drop them as fast as possible.

### 2.5.1 Partial Predicate Push-down

While query optimizers regularly push conjunctive (`AND`) predicates down to selections, it is difficult to do so for complex, multi-table predicates involving disjunctions (`OR`). Quickstep addresses this issue by using an optimization rule that pushes down *partial predicates* that conservatively approximate the result of the original predicate.

Consider a complex disjunctive multi-relation predicate $P$ in the form $P = (p_{1,1} \wedge \cdots \wedge p_{1,m_1}) \vee \cdots \vee (p_{n,1} \wedge \cdots \wedge p_{n,m_n})$, where each term $p_{i,j}$ may itself be a complex predicate but depends only on a single relation. While $P$ itself cannot be pushed down to any of the referenced relations (say $R$), we show how an appropriate relaxation of $P$, $P'(R)$, can indeed be pushed down and applied at a relation $R$.

This predicate approximation technique derives from the insight that if any of the terms $p_{i,j}$ in $P$ does not depend on $R$, it is possible to relax it by replacing it with the tautological predicate $\top$ (i.e., `TRUE`). Clearly, this technique is only useful if $R$ appears in every conjunctive clause in $P$, since otherwise $P$ relaxes and simplifies to the trivial predicate $\top$. So let us assume without loss of generality that $R$ appears in the first term of every clause, i.e., in each $p_{i,1}$ for $i = 1, 2, \ldots, n$. After relaxation, $P$ then simplifies to $P'(R) = p_{1,1} \vee p_{1,2} \vee \ldots \vee p_{1,n}$, which only references the relation $R$.

The predicate $P'$ can now be pushed down to $R$, which often leads to significantly fewer redundant tuples flowing through the rest of the plan. However, since the exact predicate must later be evaluated again, such a partial push down is only useful if the predicate is selective. Quickstep uses a rule-based approach to decide when to push down predicates, but in the future we plan to expand this method to consider a cost-based approach based on estimated cardinalities and selectivities instead.

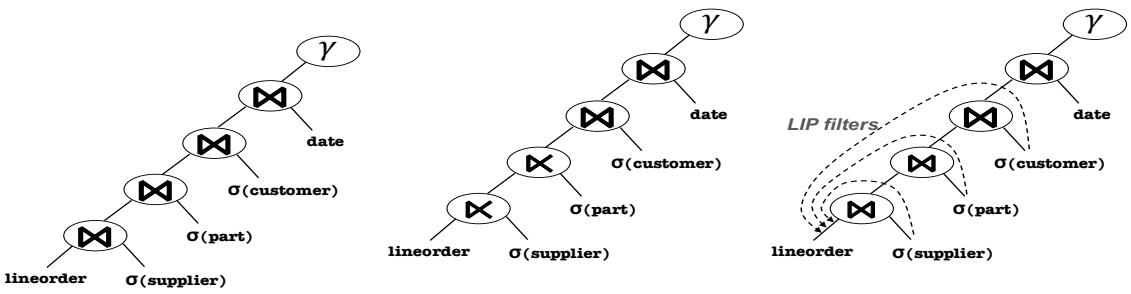
There is a discussion of join-dependent expression filter pushdown technique in [22], but the overall algorithm for generalization, and associated details, are not presented. The partial predicate push-down can be considered a generalization of such techniques.

Note that the partial predicate push down technique is complimentary to implied predicates used in SQL Server [85] and Oracle [96]. Implied predicates use statistics from the catalog to add additional filter conditions to the original predicate. Our technique does not add any new predicates, instead it replaces the predicates from another table to `TRUE`.

### 2.5.2 Exact Filters: Join to Semi-join Transformation

A new query processing approach that we introduce (which, to the best of our knowledge, has not been described before) is to identify opportunities when a join can be transformed to a semi-join, and to then use a fast, cache-efficient semi-join implementation using a succinct bitvector data structure to evaluate the join(s) efficiently. This bitvector data structure is called an *Exact Filter* (EF), and we describe it in more detail below.

To illustrate this technique, consider the SSB [95] query Q4.1 (see Figure 2.4a). Notice that in this query the `part` table does not contribute any attributes to the join result with `lineorder`, and the primary key constraint guarantees that the `part` table does not contain duplicates of the join key. Thus, we can transform the `lineorder` – `part` join into a semi-join, as shown in Figure 2.4b. During query execution, after the selection predicate is applied on the `part` table, we insert each resulting value in the join key (`p_partkey`) into an *exact filter*. This filter is implemented as a bitvector, with one bit for each potential `p_partkey` in the `part` table. The size of this bitvector is known during query compilation based on the min-max statistics present in the catalog. (These statistics in the catalog are kept updated for permanent tables even if the data is modified.) The EF is then probed using the `lineorder` table. The `lineorder` – `supplier` join also benefits from this optimization.




(a) Original query plan (b) Plan using join to semi-join transformation (c) Query plan using LIP (only)

Figure 2.4: **Query plan variations for SSB Query 4.1**

The implementation of semi-join operation using EF rather than hash tables improves performance for many reasons. First, by turning insertions and probes into fast bit operations, it eliminates the costs of hashing keys and chasing collision chains in a hash table. Second, since the filter is far more succinct than a hash table, it improves the cache hit ratio. Finally, the predictable size of the filter eliminates costly hash table resize operations that occur when selectivity estimates are poor.

The same optimization rule also transforms anti-joins into semi-anti-joins, which are implemented similarly using EFs.

### 2.5.3 Lookahead Information Passing (LIP)

Quickstep also employs a join processing technique called LIP that combines the "drop early" and "drop fast" principles underlying the techniques we described above. We only briefly discuss this technique here, and refer the reader to related work [136] for more details.

Consider SSB Query 4.1 from Figure 2.4a again. The running time for this query plan is dominated by the cost of processing the tree of joins. We observe that a `lineorder` row may pass the joins with `supplier` and `part`, only to be dropped by the join with `customer`. Even if we assume that the joins are performed in the optimal order, the original query plan performs redundant hash table probes and materializations for this

row. The essence of the LIP technique is to look ahead in the query plan and drop such rows early. In order to do so efficiently, we use *LIP filters*, typically an appropriately-configured Bloom filter [20].

The LIP technique is based on semi-join processing and sideways information passing [18, 61, 17], but is applied more aggressively and optimized for left-deep hash join trees in the main-memory context. For each join in the join tree, during the hash-table build phase, we insert the build-side join keys into an LIP filter. Then, these filters are all passed to the probe-side table, as shown in Figure 2.4c. During the probe phase of the hash join, the probe-side join keys are looked up in all the LIP filters prior to probing the hash tables. Due to the succinct nature of the Bloom filters, this LIP filter probe phase is more efficient than hash table probes, while allowing us to drop most of the redundant rows early, effectively pushing down all build-side predicates to the probe-side table scan.

During query optimization, Quickstep first pushes down predicates (including partial push-down described above) and transforms joins to semi-joins. The LIP technique is then used to speed up the remaining joins. Note that our implementation of LIP generalizes beyond the discussion here to also push down filters across other types of joins, as well as aggregations. In addition to its performance benefits, LIP also provably improves robustness to join order selection through the use of an adaptive technique, as discussed in detail in [136].

## 2.6 Evaluation

In this section, we present results from an empirical evaluation comparing Quickstep with other systems. We note that performance evaluation is always a tricky proposition as there are a large number of potential systems to compare with. Our goal here is to compare with popular systems that allow running end-to-end queries for TPC-H and SSB benchmarks, and pick three popular representative systems that each have different approaches to high performance analytics, and support stand-alone/single node in-memory query execution. We note that a large number of different SQL data platforms have been

built over the past four decades, and a comparison of all systems in this ecosystem is beyond the scope of our study.

The three open-source systems that we use are MonetDB [60], PostgreSQL [103] and Spark [133, 15] and the commercial system is VectorWise [137]. We note that there is a lack of open-source in-memory systems that focus on high-performance on a single node (the focus of the Quickstep system). VectorWise and Hyper [63] are newer systems, and though informal claims for them easily outperforming MonetDB can often be heard at conferences, that aspect has never been cataloged before. We hope that using both VectorWise and MonetDB in our study fills part of this gap. We would have liked to try Hyper, as both VectorWise and Hyper represent systems in this space that were designed over the last decade; but as readers may be aware, Hyper is no longer available for evaluation.

Next, we outline our reasons for choosing these systems. MonetDB, is an early column-store database engine that has seen over two decades of development. We also compare with VectorWise, which is a commercial column store system with origins in MonetDB. PostgreSQL is representative of a traditional relational data platform that has had decades to mature, and is also the basis for popular MPP databases like CitusDB [33], GreenPlum [50], and Redshift [111]. We use PostgreSQL v. 9.6.2, which includes about a decade's worth of work by the community to add intra-query parallelism [104]. We chose Spark as it is an increasingly popular in-memory data platform. Thus, it is instructive just for comparison purposes, to consider the relative performance of Quickstep with Spark. We use Spark 2.1.0, which includes the recent improvements for vectorized evaluation [112].

### 2.6.1 Workload

For the evaluation, we use the TPC-H benchmark at scale factor 100 as well as the Star Schema Benchmark (SSB) at scale factors 50 and 100. Both these benchmarks illustrate workloads for decision support systems.

For the results presented below, we ran each query 5 times in succession in the same session. Thus, the first run of the query fetches the required input data into memory, and

the subsequent runs are "hot." We collect these five execution times and report the average of the middle three execution times.

### 2.6.2 System Configuration

For the experiments presented below, we use a server that is provisioned as a dedicated "bare-metal" box in a larger cloud infrastructure [113]. The server has two Intel Xeon E5-2660 v3 2.60 GHz (Haswell EP) processors. Each processor has 10 cores and 20 hyper-threading hardware threads. The machine runs Ubuntu 14.04.1 LTS. The server has a total of 160GB ECC memory, with 80GB of directly-attached memory per NUMA node. Each processor has a 25MB L3 cache, which is shared across all the cores on that processor. Each core has a 32KB L1 instruction cache, 32KB L1 data cache, and a 256KB L2 cache.

### 2.6.3 System Tuning

Tuning systems for optimal performance is a cumbersome task, and much of the task of tuning is automated in Quickstep. When Quickstep starts, it automatically senses the available memory and grabs about 80% of the memory for its buffer pool. This buffer pool is used for both caching the database and also for creating temporary data structures such as hash tables for joins and aggregates. Quickstep also automatically determines the maximum available hardware parallelism, and uses that to automatically determine and set the right degree of intra-operator and intra-query parallelism. As noted in Section 2.3.1, Quickstep allows both row-store and column-store formats. These are currently specified by the users when creating the tables, and we find that for optimal performance, in most cases, the fact tables should be stored in (compressed) column store format, and the dimension tables in row-store formats. We use this *hybrid* storage format for the databases in the experiments below.

MonetDB too aims to work without performance knobs. MonetDB however does not have a buffer pool, so some care has to be taken to not run with a database that pushes the

edge of the memory limit. MonetDB also has a read-only mode for higher performance, and after the database was loaded, we switched to this mode.

The other systems require some tuning to achieve good performance, as we discuss below.

For VectorWise, we increased the buffer pool size to match the size of the memory on the machine (VectorWise has a default setting of 40 GB). We also set the number of cores and the maximum parallelism level flags to match the number of cores with hyper-threading turned on.

PostgreSQL was tuned to set the degree of parallelism to match the number of hyper-threaded cores in the system. In addition, the shared buffer space was increased to allow the system to cache the entire database in memory. The temporary buffer space was set to about half the shared buffer space. This combination produced the best performance for PostgreSQL.

Spark was configured in standalone mode and queries were issued using Spark-SQL from a Scala program. We set the number of partitions (`spark.sql.shuffle.partitions`) to the number of hyperthreaded cores. We experimented with various settings for the number of workers and partitions, and used the best combination. This combination was often when the number of workers was a small number like 2 or 4 and the number of partitions was set to the number of hyper-threaded cores.

Unlike the other systems, Spark sometimes picks execution plans that are quite expensive. For example, for the most complex queries in the SSB benchmark (the Q4.X queries), Spark choses a Cartesian product. As a result, these queries crashed the process when it ran out of memory. We rewrote the `FROM` clause in these queries to enforce a better join order. We report results from these rewritten queries below.

### 2.6.4 TPC-H at Scale Factor 100

Figure 2.5 shows the results for all systems when using the TPC-H dataset at SF 100 (∼100GB dataset).

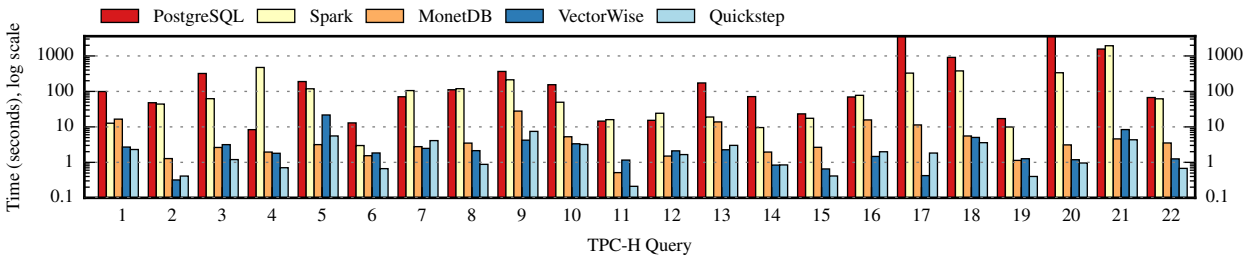



Figure 2.5: **Comparison with TPC-H, scale factor 100. Q17 and Q20 did not finish on PostgreSQL after an hour.**

As can be seen in Figure 2.5, Quickstep far outperforms MonetDB, PostgreSQL and Spark across all the queries, and in many cases by an order-of-magnitude (the y-axis is on a log scale). These gains are due to three key aspects of the design of the Quickstep system: the storage and scheduling model that maximally utilize available hardware parallelism, the template metaprogramming framework that ensures that individual operator kernels run efficiently on the underlying hardware, and the query processing and optimization techniques that eliminate redundant work using cache-efficient data structures. Comparing the total execution time across all the queries in the benchmark, both Quickstep and VectorWise are about **2X** faster than MonetDB and **orders-of-magnitude** faster than Spark and PostgreSQL.

When comparing Quickstep and VectorWise, the total run times for the two systems (across all the queries) is 46s and 70s respectively, making Quickstep ∼34% faster than VectorWise. Across each query, there are queries where each system outperforms the other. Given the closed-source nature of VectorWise, we can only speculate about possible reasons for performance differences.

VectorWise is significantly faster (at least 50% speedup) in 3 of the 22 queries. The most common reason for Quickstep's slowdown is the large cost incurred in materializing intermediate results in queries with deep join trees, particularly query 7. While the use of partial push-down greatly reduced this materialization cost already (by about 6X in query 7, for instance), such queries produce large intermediate results. Quickstep currently does

not have an implementation for late materialization of columns in join results [116], which hurts its performance. Quickstep also lacks a fast implementation for joins when the join condition contains non-equality predicates (resulting in 4X slowdown in query 17), as well as for aggregation hash tables with composite, variable-length keys (such as query 10).

On the other hand, Quickstep significantly outperforms VectorWise (at least 50% speedup) in 10 of the 22 queries. Across the board, the use of LIP and exact filters improves Quickstep's performance by about 2X. In particular, Quickstep's 4X speedup over VectorWise in query 5 can be attributed to LIP (due to its deep join trees with highly selective predicates on build-side). Similarly, we attribute a speedup of 4.5X in query 11 to exact filters, since every one of the four hash joins in a naive query plan is eliminated using this technique. The combination of these features also explains about 2X speedups in queries 3 and 11. We also see a 4.5X speedup for query 6, which we have not been able to explain given that we only have access to the VectorWise binaries. Query 19 is 3X faster in Quickstep. This query benefits significantly from the partial predicate push-down technique (cf. Section 2.5.1). VectorWise appears to also do predicate pushdown [22], but its approach may not be as general as our approach.

For the remaining 9 queries, Quickstep and VectorWise have comparable running times. We have also carried out similar experiments using the SSB benchmark; these results are reported in [99].

As noted above (cf. Section 2.6.3), Quickstep uses a hybrid database format with the fact table is stored in compressed column store format and the dimension tables in a row store format. For the TPC-H SF 100 dataset, we ran an experiment using a pure row store and pure compressed column store format for the *entire* database. The hybrid combination was 40% faster than the pure compressed column store case, and 3X faster than the pure row store case, illustrating the benefit of using a hybrid storage combination. We note that these results show smaller improvements for column stores over row stores compared to earlier comparisions, e.g. [12]; although this previous work has used indirect comparisons using the SSB benchmark and across two different systems.

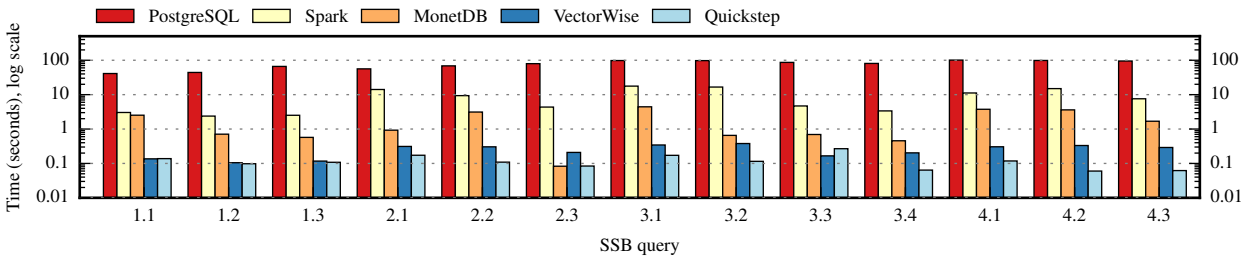



Figure 2.6: **Comparison with denormalized SSB, scale factor 50.**

### 2.6.5 Denormalizing for higher performance

In this experiment, we consider a technique that is sometimes used to speed up read-mostly data warehouses. The technique is denormalization, and data warehousing software product manuals often recommend considering this technique for read-mostly databases (e.g. [86, 59, 121]).

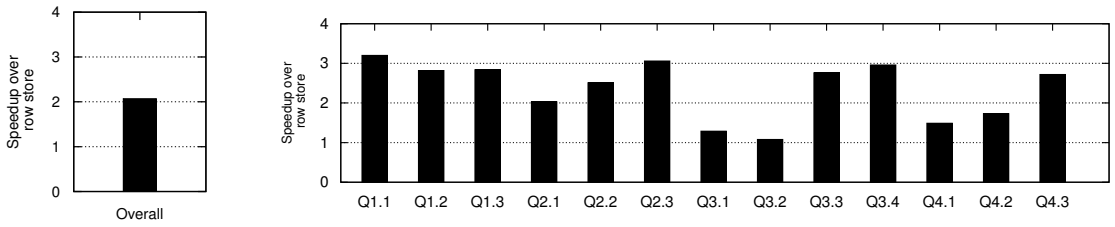
For this experiment, we use a specific schema-based denormalization technique that has been previously proposed [77]. This technique walks through the schema graph of the database, and converts all foreign-key primary-key "links" into an outer-join expression (to preserve NULL semantics). The resulting "flattened" table is called a WideTable, and it is essentially a denormalized view of the entire database. The columns in this WideTable are stored as column stores, and complex queries then become scans on this table.

An advantage of the WideTable-based denormalization is that it is largely agnostic to the workload characteristics (it is a schema-based transformation). Thus, it is easier to use in practice than selected materialized view methods.

We note that every denormalization technique has the drawback of making updates and data loading more expensive. For example, loading the denormalized WideTable in Quickstep takes about 10X longer than loading the corresponding normalized database. Thus, this method is well-suited for very low update and/or append only environments.

For this experiment, we used the SSB dataset at scale factor 50. The raw denormalized dataset file is 128GB.

The results for this experiment are shown in Figure 2.6. The total time to run all thirteen queries is 1.6s, 3.2s, 23.2s, 1,014s, and 111.9s across Quickstep, VectorWise, MonetDB, PostgreSQL and Spark respectively. Quickstep's advantage over MonetDB now increases to over an order-of-magnitude (**14X**) across most queries. MonetDB struggles with the WideTable that has 58 attributes. MonetDB uses a BAT file format, in which it stores the pair (attribute and object-id) for each column. In contrast, Quickstep's block-based storage design does not have the overhead of storing the object-id/tuple-id for each attribute (and for each tuple). The disk footprint of the database file is only 42 GB for Quickstep while it is 99 GB for MonetDB. Tables with such large schemas hurt MonetDB, while Quickstep's storage design allows it to easily deal with such schemas. Since queries now do not require joins (they become scans on the WideTable), Quickstep sees a significant increase in performance. Quickstep is also about **2X** faster than VectorWise, likely because of similar reasons as that for MonetDB. To the best of our knowledge, the internal details about VectorWise's implementation have not been described publicly, but they likely inherit aspects of MonetDB's design, since the database disk footprint is 63 GB.




(a) Overall

(b) Detailed query speedups

Figure 2.7: **Impact of storage format on performance for SSB scale factor 100**

Quickstep's speedup over the other systems also continues when working with tables with a large number of attributes. Compared to Spark and PostgreSQL, Quickstep is **70X** and **640X** faster. Notice that compared to the other systems, PostgreSQL has only a pure row-store implementation, which hurts it significantly when working with tables with a large number of attributes.

### 2.6.6 Impact of Row Store vs. Column Store

As described in Section 2.3, Quickstep supports both row store and column store formats. In this experiment, we use the multiple storage format feature in Quickstep to study the impact of different storage layouts, and specifically we compare a row-store versus a column-store layout. A notable example of such comparison is the work by Abadi et al. [12], in which the SSB benchmark was used to study this aspect, but across two *different* systems – one that was a row-store system and the other was a column-store (C-store) [120]. In this experiment, we also use the SSB benchmark, but we use a 100 scale factor dataset (instead of the 10 scale factor datase that was used in [12]).

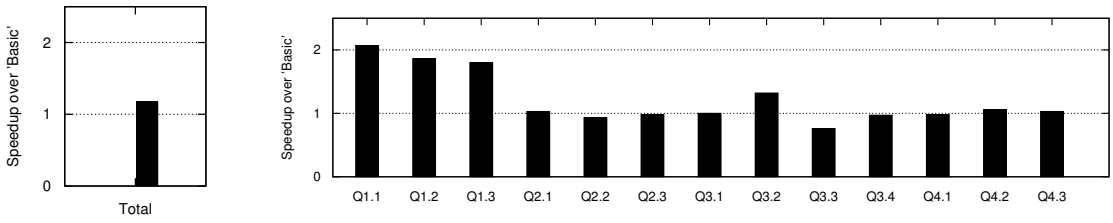
In Figure 2.7, we show the speedup of the (default) column store compared to the row store format. The use of a column store format leads, unsurprisingly, to higher performance over using a row store format. The simpler Q1.Y queries show far bigger improvements as the input table scan is a bigger proportion of the query execution time. The other queries spend a larger fraction of their time on joins, and passing tuples between the join operations. Consequently, switching to a column store has a less dramatic improvement in performance for these queries.

An interesting note is that the impact of column stores here is smaller than previous comparisons [12] which have compared these approaches across two different systems and showed about a 6X improvement for column-stores. Overall, we see a 2X improvement for column-stores, which is lower than these previous results. One key reason for the lower improvement is because column stores typically help speed up scans (select operators). However query plans are often complex with several different types of operators. Thus while considering overall query execution time, selection forms one fraction of the overall time distribution.

We have also experimented with compressed column-stores in this same setting, and find that they are slower than non-compressed column stores. Compressed column stores are still faster than row store by about 50% overall. But compression adds run-time CPU overhead which reduces is overall performance compared to regular column stores. The

results for TPC-H are similar. We present the TPC-H results in Chapter 4, where we also discuss the impact of various dimensions like storage format, parallelism and block size on the performance of pipelined query processing.

### 2.6.7 Template Metaprogramming Impact




(a) Total time

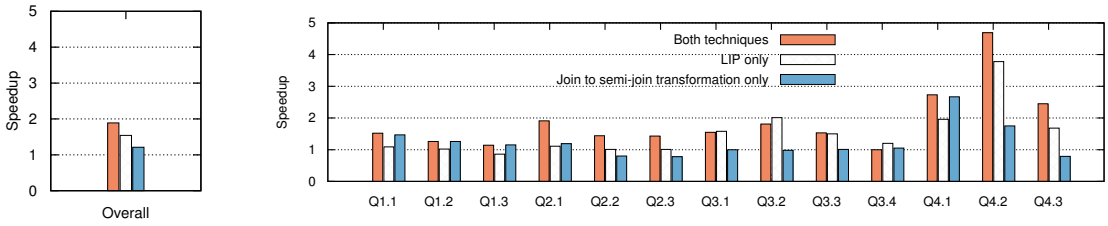
(b) Individual query execution times

Figure 2.8: **Impact of template metaprogramming: Changes in total time and individual query times**

Next, we toggle the use of the template metaprogramming (see Section 2.3.3.1), using the SSB 100 scale factor dataset. Specifically, we change the compile time flag that determines whether the `ValueAccessor` is constructed by copying attributes (Basic) or by providing an indirection (Selection). The results for this experiment are shown in Figure 2.8.

The overall performance impact of eliminating the copy during predicate evaluation is about 20%. As with the previous experiment, the benefits are larger for the simpler Q1.X queries and lower for the other queries that tend to spend most of their time on join operations and in the pipelines in passing tuples between different join operations.

The result for this experiment with TPC-H show far smaller improvements (see Figure 2.1 for a typical example), as the TPC-H queries spend a far smaller fraction on their overall time on expression evaluation (compared to the SSB queries).




(a) Overall

(b) Breakdown of speedup due to techniques

Figure 2.9: **Impact of Exact Filter and LIP using SSB at scale factor 100.**

### 2.6.8 Impact of Optimization Techniques

We described the novel optimization techniques in Quickstep optimizer in Section 2.5. In this experiment, we measure the impact of these techniques individually, viz. LIP and join to semi-join transformation.

As with the previous two sections, we use the SSB 100 scale factor dataset. (The results for the TPC-H dataset is similar.) Figure 2.9 shows the impact of these techniques over a baseline in which neither of these techniques are used. As shown in Figure 2.9, these techniques together provide a nearly 2X speedup for the entire benchmark. The LIP and semi-join transformation techniques individually provide about 50% and 20% speedup respectively. While some queries do see a slowdown due to the individual techniques, the application of both techniques together always gives some speedup. In fact, of the 13 queries in the benchmark, 8 queries see at least a 50% speedup and three queries see at least 2X speedup. The largest speedups are in the most complex queries (group 4), where we see an overall speedup of more than 3X.

These results validate the usefulness of these techniques on typical workloads. Further, their simplicity of implementation and general applicability leads us to believe that these techniques should be more widely used in other database systems.

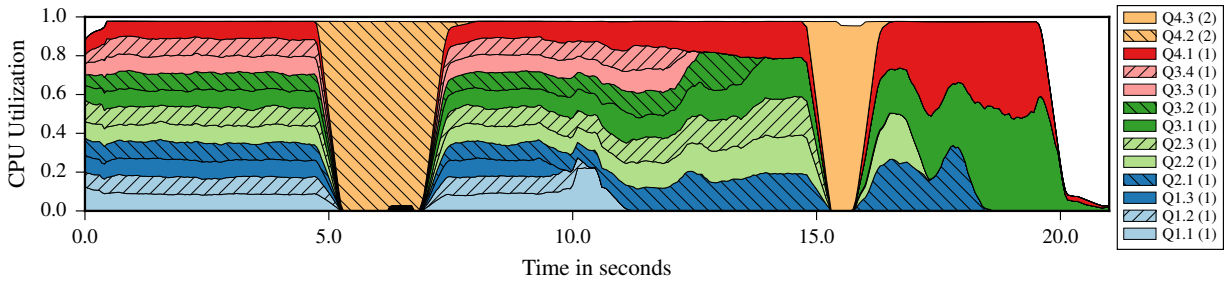



Figure 2.10: **Prioritized query execution. QX.Y(1) indicates that Query X.Y has a priority 1. Q4.2 and Q4.3 have higher priority (2) than the other queries (1).**

### 2.6.9 Elasticity

In this experiment, we evaluate Quickstep's ability to quickly change the degree of inter-query parallelism, driven by the design of its work-order based scheduling approach (cf. Section 2.4.2). For this experiment, we use the 100 scale factor SSB dataset. The experiment starts by concurrently issuing the first 11 queries from the SSB benchmark (i.e. Q1.1 to Q4.1), against an instance of Quickstep that has just been spun up (i.e. it has an empty/cold database buffer pool). All these queries are tagged with equal priority, so the Quickstep scheduler aims to provide an equal share of the resources to each of these queries. While the concurrent execution of these 11 queries is in progress, two high priority queries enter the system at two different time points. The results for this experiment are shown in Figure 2.10. In this figure, the y-axis shows the fraction of CPU resources that are used by each query, which is measured as the fraction of the overall CPU cycles utilized by the query.

Notice in Figure 2.10, at around the 5 second mark when the high priority query Q4.2 arrives, the Quickstep scheduler quickly stops scheduling work orders from the lower priority queries and allocates all the CPU resources to the high-priority query Q4.2. As the execution of Q4.2 completes, other queries simply resume their execution.

Another high priority query (Q4.3) enters the system at around 15 seconds. Once again, the scheduler dedicates all the CPU resources to Q4.3 and pauses the lower priority

queries. At around 17 seconds, as the execution of query Q4.3 completes, the scheduler resumes the scheduling of work orders from all remaining active lower priority queries.

This experiment highlights two important features of the Quickstep scheduler. First, it can dynamically and quickly adapt its scheduling strategies. Second, the Quickstep scheduler can naturally support query suspension (without requiring complex operator code such as [34]), which is an important concern for managing resources in actual deployments.

### 2.6.10 Built-in Query Progress Monitoring

An interesting aspect of using a work-order based scheduler (described in Section 2.4.2) is that the state of the scheduler can easily be used to monitor the status of a query, without requiring any changes to the operator code. Thus, there is a generic in-built mechanism to monitor the progress of queries.

Quickstep can output the progress of the query as viewed by the scheduler, and this information can be visualized. As an example, Figure 2.11 shows the progress of a query with three join operations, one aggregation, and one sort operation.

## 2.7 Related Work

We have noted related work throughout this chapter, and we highlight some of the key areas of overlapping research here.

There is tremendous interest in the area of main-memory databases and a number of systems have been developed, including [72, 21, 108, 63, 132, 13, 15, 137, 45, 97], While similar in motivation, our work employs a unique block-based architecture for storage and query processing, as well as fast query processing techniques for in-memory processing. The combination of these techniques not only leads to high performance, but also gives rise to interesting properties in this end-to-end system, such as elasticity (as shown in Section 2.6.9).

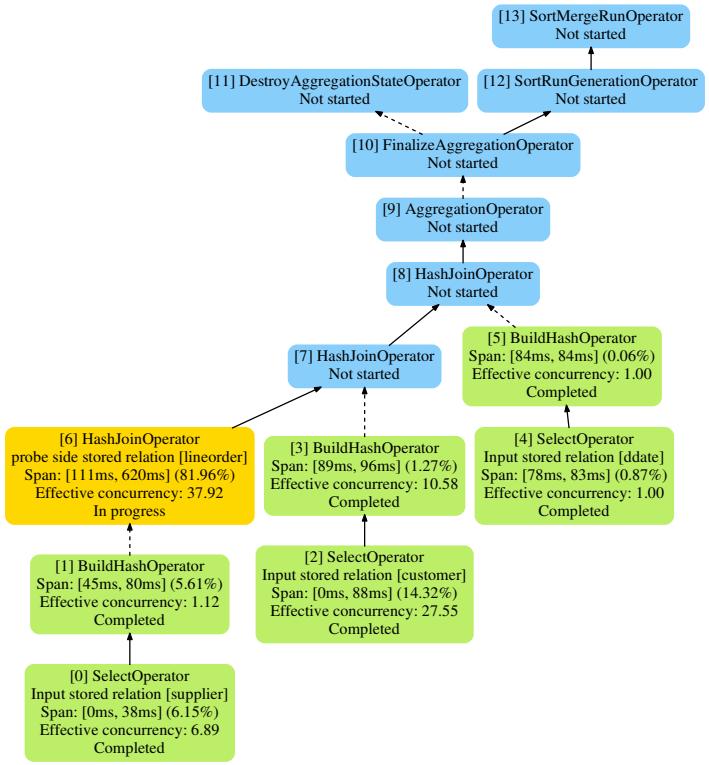



Figure 2.11: **Query progress status. Green nodes (0-5) indicate work that is completed, the yellow node (6) corresponds to operators whose work-orders are currently being executed, and the blue nodes (7-13) show the work that has yet to be started.**

Our vectorized execution on blocks has similarity to the work on columnar execution methods, including recent proposals such as [135, 109, 62, 128, 127, 76, 11, 105, 46]. Quickstep's template metaprogramming-based approach relies on compiler optimizations to make automatic use of SIMD instructions. Our method is complementary to run-time code generation (such as [62, 128, 127, 11, 105, 46, 57, 92, 15, 102, 89, 135, 109]). Our template metaprogramming-based approach uses static (compile-time) generation of the appropriate code for processing tuples in each block. This approach eliminates the per-query run-time code generation cost, which can be expensive for short-running queries. An interesting direction for future work is to consider combining these two approaches.

The design of Quickstep's storage blocks has similarities to the tablets in Google's BigTable [26]. However, tablets' primary purpose is to serve sorted key-value store applications whereas Quickstep's storage blocks adhere to a relational data model allowing for optimization such as efficient expression evaluation (cf. Section 2.3.3).

Our use of a block-based storage design naturally leads to a block-based scheduling method for query processing, and this connection has been made by Chasseur et al. [27] and Leis et al. [73]. In this work, we build on these ideas. We also leverage these ideas to allow for desirable properties, such as dynamic elastic behavior (cf. Section 2.6.9).

Philosophically, the block-based scheduling that we use is similar to the MapReduce style query execution [36]. A key difference between the two approaches is that there is no notion of pipelining in the original MapReduce framework, however Quickstep allows for pipelined parallelism. Further, in Quickstep common data structures (e.g. an aggregate hash table) can be shared across different tasks that belong to the same operator.

The exact filters build on the rich history of semi-join optimization dating back at least to Bernstein and Chiu [18]. The LIP technique presented in Section 2.5.3 also draws on similar ideas, and is described in greater detail in [136].

Achieving *robustness* in query processing is a goal for many database systems [49]. Quickstep uses the LIP technique to achieve robust performance for star-schema queries. We formally define the notion of robustness and prove the robustness guarantees provided

by Quickstep. VectorWise uses micro-adaptivity technique [107] for robustness, but their focus is largely on simpler scan operations.

Overall, we articulate the growing need for the scaling-up approach, and present the design of Quickstep that is designed for a very high-level of intra-operator parallelism to address this need. We also present a set of related query processing and optimization methods. Collectively our methods achieve high performance on modern multi-core multi-socket machines for in-memory settings.

## 2.8 Conclusions & Future Work

Compute and memory densities inside individual servers continues to grow at an astonishing pace. Thus, there is a clear need to complement the emphasis on "scaling-out" with an approach to "scaling-up" to exploit the full potential of parallelism that is packed inside individual servers.

We presented the design and implementation of Quickstep that emphasizes a scaling-up approach. Quickstep currently targets in-memory analytic workloads that run on servers with multiple processors, each with multiple cores. Quickstep uses a novel independent block-based storage organization, a task-based method for executing queries, a template metaprogramming mechanism to generate efficient code statically at compile-time, and optimizations for predicate push-down and join processing. We also present end-to-end evaluations comparing the performance of Quickstep and a number of other contemporary systems. Our results show that Quickstep delivers high performance, and in some cases is faster than some of the existing systems by over an order-of-magnitude.

Aiming for higher performance is a never-ending goal, and there are a number of additional opportunities to achieve even higher performance in Quickstep. Some of these opportunities include operator sharing, fusing operators in a pipeline, improvements in individual operator algorithms, dynamic code generation, and exploring the use of adaptive indexing/storage techniques. We plan on exploring these issues as part of future work. We also plan on building a distributed version of Quickstep.

# Chapter 3

# Design and Implementation of Quickstep Scheduler

## 3.1 Introduction

Concurrent queries are common in various settings, such as application stacks that issue multiple queries simultaneously and multi-tenant database-as-a-service environments [91, 90]. There are several challenges associated with scheduling concurrent execution of queries in such environments.

The first challenge is related to exploiting the large amount of hardware parallelism that is available inside modern servers, as it requires dealing with two key types of parallelism. The first type is *intra-query parallelism*. Modern database systems often use query execution methods that have a high intra-query parallelism [27, 74, 126]. Concurrent query execution adds another layer of parallelism, i.e. *inter-query parallelism*.

The second challenge is that workloads are often dynamic in nature. For each query, its resource (e.g. CPU and memory) demands can vary over the life-span of the query. Furthermore, different queries can arrive and depart at any time. Maintaining a level of Quality of Service (QoS) with dynamic workloads is an important challenge for the database cloud vendor.

To address these issues, we present a concurrent query execution scheduling framework for analytic in-memory database systems. We cast the goals for the scheduler framework by relating it to a governance model, since the framework *governs* the use of resources for executing queries in the system. Next, we describe some goals for a governance model and translate them in the context of our framework.

First, an ideal governance model should be *transparent*, i.e. decisions should be taken based on the guiding principles and they should be clearly understandable. In the context of scheduling, we can interpret this goal as requiring high level *policies* that can govern the resource allocation among concurrent queries. This goal also highlights the need to *separate mechanisms and policies*, a well-known system design principle [71]. The scheduler needs to provide an easy way to specify a variety of policies (e.g. priority-based or equal/fair allocation) that can be implemented with the underlying mechanisms. Ideally, the scheduler should adhere to the policy even if the query plan that it has been given has poor estimates for resource consumption.

Second, the governance model should be *responsive* to dynamic situations. Thus, the scheduler must be reactive and auto-magically deal with changing conditions; e.g., the arrival of a high-priority query or an existing query taking far more resources than expected. A related goal for the scheduler is to *control* and *predictably deal* with resource thrashing.

Finally, the governance should be *efficient* and *effective*. Thus, the scheduler must work with the data processing kernels in the system to use the hardware resources effectively to realize high cost efficiency and high performance from the underlying deployment. In main-memory database deployments, one aspect of effective resource utilization requires using all the processing cores in the underlying server effectively.

**Contributions:** We present the design of a scheduler framework that meets the above goals. We have implemented our scheduler framework in an open-source, in-memory database system, called Quickstep.

A distinguishing aspect of this paper compared to previous work is that we present a holistic scheduling framework to deal with both intra and inter query parallelism in a single scheduling algorithm. Therefore, our framework is far more comprehensive and more broadly applicable than previous work.

Our framework employs a design that cleanly separates policies from mechanism. This design allows the scheduler to easily support a range of different policies, and enables the system to effectively use the underlying hardware resources. The clean separation

also makes the system maintainable over time, and for the system to easily incorporate new policies. Thus, the system is *extensible*. The key underlying unifying mechanism is a probability-based framework that continuously determines resource allocation among concurrent queries. Our evaluation (see Section 3.8) demonstrates that the scheduler can allocate resources precisely as per the policy specifications.

The framework uses a novel learning module that learns about the resources consumed by concurrent queries, and uses a prediction model to predict future resource demands for *each* active query. Thus, the scheduler does not require accurate predictions about resource consumption for each stage of each query from the query optimizer (though accurate predictions are welcome as they provide a better starting point to the learning component). The predictions from the learning module can then be used to react to changing workload and/or environment conditions to allow the scheduler to realize the desired policy. Our evaluations underline the crucial impact of the learning module in the enforcement of policies. The scheduler has a built in load controller to automatically suspend and resume queries if there is a danger of thrashing.

Collectively, we present an end-to-end solution for managing concurrent query execution in complex modern in-memory database deployment environments.

The remainder of this paper is organized as follows: Section 3.2 describes some preliminaries related to Quickstep. The architecture of the scheduler framework is described in Section 3.5. Section 3.7 describes the formulation of the policies and the load control mechanisms. Section 3.8 contains the experimental results. Related work is discussed in Section 3.9.

## 3.2 Background

In this section we establish some prerequisites for the proposed Quickstep scheduler framework. Quickstep [100] is an open-source relational database engine designed to efficiently leverage contemporary hardware aspects such as large main memory, multi-core, and multi-socket server settings.

The control flow associated with query execution in Quickstep involves first parsing the query, and then optimizing the query using a cost-based query optimizer. The optimized query plan is represented as a Directed Acyclic Graph (DAG) of relational operator primitives. The query plan DAG is then sent to a *scheduler*, which is the focus of this chapter. The scheduler runs as a separate thread and coordinates the execution of all queries. Apart from the scheduler, Quickstep has a pool of worker threads that carry out computations on the data.

Quickstep uses a query execution paradigm, which is built using previously proposed approaches [27, 74]. In this paradigm a query is executed as a sequence of *work orders*. A work order operates on a data block, which is treated as a self-contained mini-database [27]. (c.f. Section 3.4 for examples of work orders) The computation that is required for each operator in a query plan is decomposed into a sequence of work orders. For example, a select operator produces as many work orders as there are blocks in the input table. We first describe the storage management in Quickstep in Section 3.3. Quickstep's work order abstraction (described in Section 3.4) is tied with its storage management (described in Section 3.3).

## 3.3 Storage Management in Quickstep

Data organization in the Quickstep storage manager holds the key to intra-query parallelism [27]. Data in a relation are organized in the form of blocks. Each block holds a collection of tuples from a single table. A unique aspect of the storage organization in Quickstep is that blocks are considered to be independent and self-contained mini-databases. Thus, when creating an index, instead of creating a global index with "pointers" to the tuples in blocks, the index fragments are stored within the blocks. Each block is internally organized into *sub-blocks*. There is one *tuple storage sub-block*, which could be in a row store or a column store format. In addition, each block has one sub-block for each index created on the table. CSB+-tree [110] and BitWeaving [76] indices are currently supported. The blocks are free to self-organize themselves and thus a given table may

have blocks in different formats. For example, new blocks in a table may be in a row store format, while older blocks may be in a column store format.

This storage block design, as articulated earlier in [27] enables the query execution to be broken down into a set of independent tasks on each block. This is a crucial aspect that we leverage in the design of our scheduler.

The storage manager also contains a *buffer pool manager*. It organizes the memory as an array of slots, and overlays blocks on top of the slots (so block sizes are constrained to be a multiple of the underlying slot size). Memory allocations for data blocks for *both* permanent and temporary tables are always made from a centralized buffer pool. In addition, all allocations for run-time data structures, such as hash tables are also made by the buffer pool. The buffer pool manager employs an LRU-2 replacement policy. Thus, it is possible for a hash table to get evicted to disk, if it has become "cold"; e.g. if it belongs to a suspended query.

## 3.4 Work Orders

Work done for executing a query in Quickstep is split into multiple *work orders*. A work order contains all the information that is needed to process tuples in a given data block. A work order encapsulates the relational operator that is being applied, the relevant input relation(s), location of the input data block, any predicate(s) to be applied on the tuples in the input block, and descriptors to other run-time structures (such as hash tables). Consider the following full table scan query to illustrate the work order concept:

```
SELECT name FROM Employee WHERE city='San Diego'
```

The plan for this query has a simple *selection* operator. For the selection operator, the number of work orders is same as the number of input blocks in the `Employee` table. Each selection work order contains the following information:

- Relation: `Employee`, attribute: `name`
- Predicate: `city='San Diego'`

- The unique ID of an input block from the `Employee` table

The work orders for a join operation are slightly more complicated. For example, a *probe work order*, contains the unique ID of the probe block, a pointer to the hash table, the projected attributes, and the join predicate. Each operator algorithm (e.g. a scan or the build/probe phase of a hash-join) in the system has a C++ class that is derived from a root abstract base class that has a virtual method called `execute()`. Executing a work order simply involves calling the `execute()` method on the appropriate operator algorithm C++ class object.

## 3.5 Design of the Scheduler

In this section, we present an overview of the components in the proposed Quickstep scheduler.

### 3.5.1 Scheduler Architecture Overview

Figure 3.1 shows the architecture of the Quickstep scheduler. We begin by describing the *Query Manager*, that coordinates the progress of a single query in the system. It maintains the query plan DAG, and a data structure called the *Work Order Container* to track all the work orders that are ready for scheduling. Recall from Section 3.2, a description of the work carried out on a block of data is a *work order*. The Query Manager generates schedulable work orders for each active operator node in the DAG. It also runs a rudimentary DAG traversal algorithm to determine when to activate nodes in the DAG. The algorithm is decribed in Appendix A.1.

An important component of the system is the *Policy Enforcer*. It selects a query among all the concurrent queries, and schedules its work order for execution. This in essence is *a scheduling decision*, and is taken based on a high-level policy provided to the system. The policy is described in *Policy Specifications*, which is an abstraction that governs how resources are shared among concurrent queries.

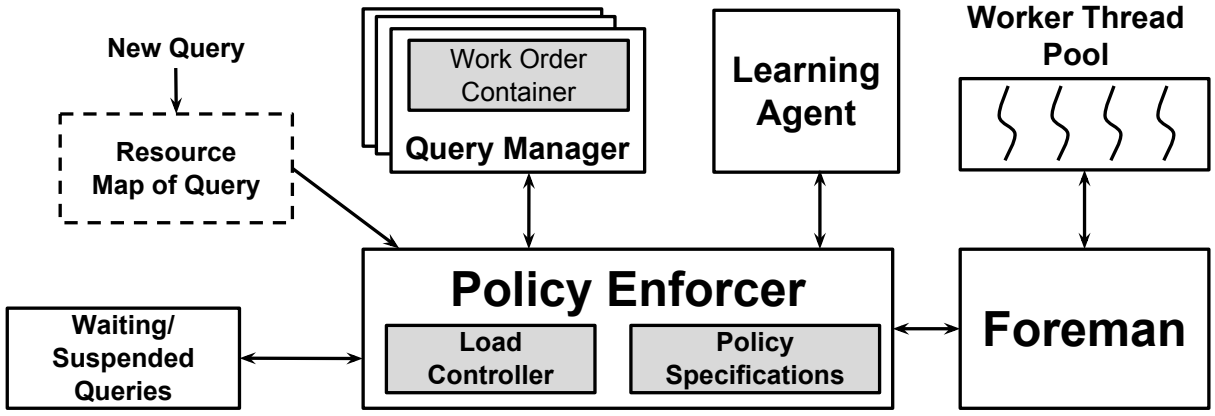



Figure 3.1: **Overview of the scheduler**

Policy Enforcer (PE) and various Query Managers (QM) communicate with each other as follows: **QM→PE**: Provides work orders that belong to the managed query for dispatching (to get executed). **PE→QM**: Upon completion of a work order, send a signal so that the QM can then decide if new nodes in the DAG can be activated, and if existing nodes can be marked as completed. A detailed description of the Policy Enforcer is present in Section 3.6.1.

The Policy Enforcer contains a *Load Controller* module, which is responsible for ensuring that the system has enough resources to meet the demands. A new query in the system presents its resource requirements for its lifetime in the form of a *Resource Map* to the Load Controller. A sample resource map is presented in Appendix A.2.

The Load Controller determines the fate of a new query. If enough resources are available, it admits the query. If the system risks thrashing due to the admission of the new query, it can take a number of decisions, including wait-listing the query or suspending older active queries to free up resources for the new query. We describe the Load Controller in Section 3.7.4.

The Policy Enforcer works with another module called the *Learning Agent*. Execution statistics of completed work orders are passed from the Policy Enforcer to the Learning

Agent. This component uses a simple learning-based method to predict the time to complete future work orders using the execution times of finished work orders. Such predictions form the basis for the Policy Enforcer's decisions regarding scheduling the next set of work orders (cf. Section 3.6.2 for details on Learning Agent).

The *Foreman* module acts as a link between the Policy Enforcer and a pool of worker threads. It receives work orders that are ready for execution from the Policy Enforcer, and dispatches them to the worker threads. The Foreman can monitor the number of pending work orders for each worker, and use that information for load-balancing when dispatching work orders. Upon completion of the execution of a work order, a worker sends execution statistics to the Foreman, which are further relayed to the Policy Enforcer. New work orders due to pipelining are generated similarly (more details on pipelining can be found in Chapter 4). We describe Quickstep's thread model in the next section.

## 3.6 Thread Model

Quickstep currently runs as a single server process, with multiple user-space threads. There are two kinds of threads. There is one *Scheduler* thread, and a pool of *Worker* threads. All the components in the scheduler architecture except the worker thread pool run in the scheduler thread. In the current implementation, all threads are spun upfront when the database server process is instantiated, and stay alive until the server process terminates.

The threads use the same address space and use shared-memory semantics for data access. In fact the buffer pool is stored in shared memory, which is accessible by all the threads. Each worker thread is typically pinned to a CPU core. Such pinning avoids costs incurred when a thread migrates from one CPU core to another, which results in loss of data and instruction cache locality. We do not pin the scheduler thread, as its CPU utilization is low and it is not worth dedicating a CPU core for the scheduler thread.

Every worker thread receives a work order from the Foreman, executes it and then waits for the next work order. In order to minimize worker's idle time, typically each

worker is issued multiple work orders at any given time. Thread-safe queues are used to communicate between the threads. The communication happens through light-weight messages from the sender to the receiver thread, which is internally implemented as placing a message object on the receiver's queue. A receiver reads messages from its queue. A thread (and its queue) is uniquely identified by its thread ID.

The thread communication infrastructure also implements additional features like the ability to query the lengths of any queue in the system, and cancellation of an unread message.

### 3.6.1 Policy Enforcer

The Policy Enforcer assigns a probability value to each active query in the system. A scheduling decision is essentially *probabilistic*, based on these probability values. The probability value assigned to a query indicates the likelihood of a work order from that query being scheduled. These probability values play a crucial role in the policy enforcement. In Section 3.7, we formally derive these probability values for different policies and also establish the relationship between probability values and the policy specifications.

An important information to determine such a probability value is an estimate about the run times of future work orders of the query. This information provides the Policy Enforcer some idea about the future resource requirements of each query. As the Policy Enforcer continuously monitors the resource allocation to different queries in the system, using these estimates, it can control the resource allocation with the goal of *enforcing* the specified policy for resource sharing. In the next section, we describe an estimation technique for the execution time of future work orders of a query.

### 3.6.2 Learning Agent

The Learning Agent module is responsible for predicting the execution times of the future work orders for a given query. It gathers the history of executed work orders of a query and applies a prediction model on such a history to estimate the execution time

of a future work order. This predicted execution time is used to compute the probability assigned to each query (cf. Section 3.7 for probability derivations).

An alternative to the Learning Agent could be a static method that assigns fixed probability values to active queries in the system. We now justify the need for the Learning Agent and highlight the limitations of the alternative mentioned above. An illustrative example is presented in A.4.

The time per work order metric doesn't stay the same throughout a query's lifetime, for reasons such as variations in input data (e.g. skew), CPU characteristics of different relational operators (e.g. scan vs hash probe). In each phase of the query, the time per work order is different. As the query plan gets bigger, the number of phases in the plan increase. In addition, different queries may be in different phases at a given point in time. To make things more complicated, queries can enter or leave the system at any time.

Therefore, it is difficult to statically pick a proportion of CPU to allocate to the concurrent queries. Hence there is a need to "learn" the various phases in the query execution and dynamically change the proportion of resources that are allocated to each query, based on each query's phase. Next, we study the methodology used by the Learning Agent.

#### 3.6.2.1 Learning Agent Methodology

The Learning agent uses the execution times of previously executed work orders $(t_{w_1}, t_{w_2}, \ldots, t_{w_k})$ to predict the execution time of the next work order $(t_{w_{k+1}})$ for a given query.[1] Figure 3.2 shows the Learning Agent's interaction with other scheduler components.

The set of previously executed work orders can belong to multiple relational operators in the query operator DAG. The Learning Agent stores the execution times of the work orders grouped by their source relational operator, e.g. the execution statistics of select

[1] In the beginning of a query execution, when enough information about work order execution times is not available, we use the default probabilities in the Policy Enforcer, instead of using default predicted times in the Learning Agent.

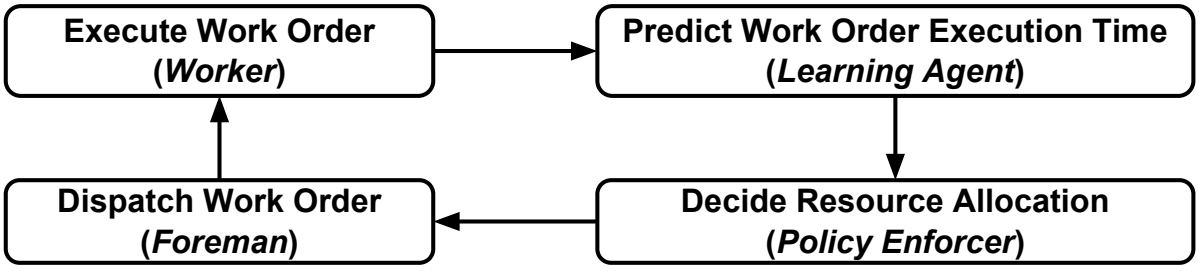



Figure 3.2: **Interactions among scheduler components**

work orders are maintained together and kept separate from those of aggregation work orders.

Quickstep's scheduler currently uses linear regression (specifically autoregression, which is a smaller class of linear regression) as the prediction model. We chose linear regression as it is fast, accurate, and efficient w.r.t. the computational and the storage requirements of the model. More details about our use of linear regression is described in A.5.

The problem of estimating the query execution time is well-studied, but requires complex methods [43, 130, 75, 28]. The Learning Agent does not require such methods. However, it can combine estimates from other methods with its own estimates.

## 3.7 Policy Derivations and Load Controller Implementation

Our work focuses on two critical system resources for in-memory database deployments: CPU and memory. The policies treat CPU as a *divisible resource*, and the policy specifications are defined in terms of relative CPU utilizations of queries or query classes. The load controller implementation treats memory as a *gating resource* and its goal is to avoid memory thrashing. We justify the choice of resources for policies and load controller implementations in Appendix A.6.

A policy specification consists of two parts: an inter-class specification (resource allocation policy across query classes), and an intra-class specification (resource allocations among queries within the same class). The default setting is uniform allocations for both intra and inter-class policies.

| Policy | Interpretation |
|---|---|
| Fair | In a given time interval, all active queries should get an equal proportion of the total CPU cycles across all the cores. |
| Highest Priority First (HPF) | Queries are executed in the order of their priority values; i.e. a higher priority query is preferred over a lower priority query for scheduling its work order. |
| Proportional Priority (PP) | The collective resources that are allocated to a query class (i.e. all queries with the same priority value) is proportional to the class' priority value based on a specified scale; e.g. (linear, exponential). |

Table 3.1: **Interpretations of the policies implemented in Quickstep**

In Section 3.7.4, we describe Quickstep's load control mechanisms. The load controller takes admission control and query suspension decisions based on the memory resource.

The scheduling policies, described below in Sections 3.7.1, 3.7.2, and 3.7.3 are subject to the decisions made by the load controller, i.e. the policies apply to the queries that are admitted by the load controller and have not been suspended.

The interpretations of various policies are presented in Table 3.1. Note that for the Fair policy, there is only one class. For the priority-based policies, we assume that queries are tagged with priority (integer) values. Next, we describe the probabilistic framework that we use to implement various policies (cf. Table 3.2 for notations).

### 3.7.1 Fair Policy Implementation

We assume $k$ concurrent active queries: $q_1, q_2, \ldots q_k$. The probability $pb_j$ is computed as: $pb_j = (\frac{1}{t_j})/(\sum_{i=1}^{k} \frac{1}{t_i})$

| Symbol | Interpretation |
|---|---|
| $q_i$ | Query $i$ |
| $pb_i$ | Probability assigned to $q_i$ |
| $PV_i$ | The priority value for $q_i$ |
| $t_i$ | Predicted work order execution time for $q_i$ |
| $t_{PV_i}$ | Proportion of time allocated for the class with priority value $PV_i$ |
| $prob_{PV_i}$ | Probability assigned to the class with priority value $PV_i$ |

Table 3.2: **Description of notations**

Observe that $pb_j \in (0, 1]$ and $\sum_{j=1}^{k} pb_j = 1$. Therefore, the $pb_j$ values can be interpreted as probability values. As all the probability values are non-zero, every query has a non-zero chance of getting its work orders scheduled.

Notice that $\forall i, j$ such that $1 \leq i, j \leq k$, $pb_i/pb_j = t_j/t_i$. If $t_i > t_j$, it means that the work orders for query $q_i$ take longer time to execute than the work orders for query $q_j$. Thus, in a given time interval, fewer work orders of $q_i$ must be scheduled as compared to the query $q_j$.

The probability associated with a query determines the likelihood of the scheduler dispatching a work order for that query. Thus, when $t_i > t_j$, $pb_j > pb_i$, i.e. the probability for $q_i$ should be proportionally smaller than probability for $q_j$.

### 3.7.2 Highest Priority First (HPF) Implementation

Let $\{PV_1, PV_2, \ldots, PV_k\}$ be the set of distinct priority values in the workload. A higher integer is assumed to imply higher importance/priority. The scheduler first finds the highest priority value among all the currently active queries which is $PV_{max}$. Next, a fair resource allocation strategy is used to allocate resources across all the active queries in that priority class.

In some situations, the queries from the highest priority value may not have enough work to keep all the workers busy. In such cases, to maximize the utilization of the available CPU resources, the scheduler may explore queries from the lower priority values to schedule work orders.

### 3.7.3 Proportional Priority (PP) Implementation

Let $P = \{PV_1, PV_2, \ldots PV_k\}$ be the set of the distinct priority values in the workload. We assume a linear scale for the priority values. A higher integer is presumed to imply higher priority.

In a unit time, a class with priority value $PV_i$ should get resources for a time that is proportional to its priority value i.e. $PV_i$. Therefore, the class with priority $PV_i$ should be allocated resources for $t_{PV_i} = PV_i / \sum_{j=1}^{k} PV_j$ amount of time.

We now estimate the number of work orders for priority class $PV_i$ that can be executed in its allotted time. For this task, we need an estimate for the execution time of a future work order from the class as a whole, referred to as $w_{PV_i}$ for the class with priority value $PV_i$. Therefore, assuming $m$ queries in a given class and the individual estimates of work order execution times for queries with priority $PV_i$ are $t_1, t_2, \ldots, t_m$, then the predicted work order execution time for the class is $w_{PV_i} = \sum_{j=1}^{m} t_j/m$. Therefore the estimated number of work orders executed for priority class $PV_i$ is $n_{PV_i} = t_{PV_i}/w_{PV_i}$.

After determining $n_{PV_1}, n_{PV_2}, \ldots, n_{PV_k}$, which are the estimated number of work orders executed by all the priority classes in their allotted time, computing probabilities for each class is straightforward. The probability of priority class $PV_i$ is $prob_{PV_i} = n_{PV_i} / \sum_{j=1}^{k} n_{PV_j}$. Next, we describe the load control mechanism.

### 3.7.4 Load Control Mechanism

Recall that the load control mechanism in Quickstep is designed to manage the availability of memory resource to the queries in the system. This task requires continuous monitoring of memory consumption in the system. The load controller component has

two functions: 1) Determining if new queries are allowed to run (a.k.a. admission control). 2) Suspending queries if the system is in danger of thrashing. We now explain how the load control mechanism realizes these two functions.

Recall from Section 3.5, that a new query entering the system presents to the Load Controller its Resource Map that describes the query's estimated range of resource requirements.

We denote the minimum and maximum memory requirements for a given query as $m_{min}$ and $m_{max}$, the threshold for maximum memory consumption for the database as $M$ and the current total memory consumption as $m_{current}$. The term $m_{current}$ includes total memory occupied by various tables, run time data structures such as hash tables for joins and aggregations for all the queries in the system.

In the simplest case, when there is enough memory to admit the query, we have $m_{max} + m_{current} < M$. In this case, the load controller can let the query enter the system.

When memory is scarce, i.e. $m_{min} + m_{current} > M$, the query can not be admitted right away. Its admission depends on the system's policy (i.e. one of the policies described earlier).

If the system is realizing the fair policy, all queries have the same priority. In this scenario, the load controller simply suspends the new query until enough memory becomes available, after which the query can be admitted.

For both priority-based policies, if the new query's priority is smaller than the minimum priority value in the system, then the load controller suspends the query. The suspended query can be admitted in the system when enough memory is available to admit it. In the other case, the load controller finds queries from the lower priority values that have high memory footprints. It continues to suspend such queries from the lower priority levels (in decreasing order of memory footprints) until enough memory becomes available to admit the given query.

## 3.8 Evaluation

In this section, we present an evaluation of our scheduler. The goals of the experimental evaluation are as follows:

1. To check if the policy enforcement meets the expected criterion defined in the policy behavior.

2. To illustrate the role of the learning component, we compare it against a policy implementation that doesn't use the learning-based feedback loop.

3. Examine the behavior of the learning-based scheduler in the presence of execution skew.

4. To observe the behavior of the load controller component of the scheduler in extreme/overloaded scenarios.

We use an instance from the Cloudlab [113] platform for our evaluation, which we described in Table 3.3.

| **Parameter** | **Description** |
|---|---|
| Processor | 2 Intel Xeon Intel E5-2660 2.60 GHz (Haswell EP) processors |
| Cores | 10 per socket, 20 per socket with hyper-threading |
| Memory | 80 GB per NUMA socket, 160 GB total |
| Caches | L3: 25 MB, L2: 256 KB, L1 (both instruction and data): 32 KB |
| OS | Ubuntu 14.04.1 LTS |

Table 3.3: **Evaluation Platform**

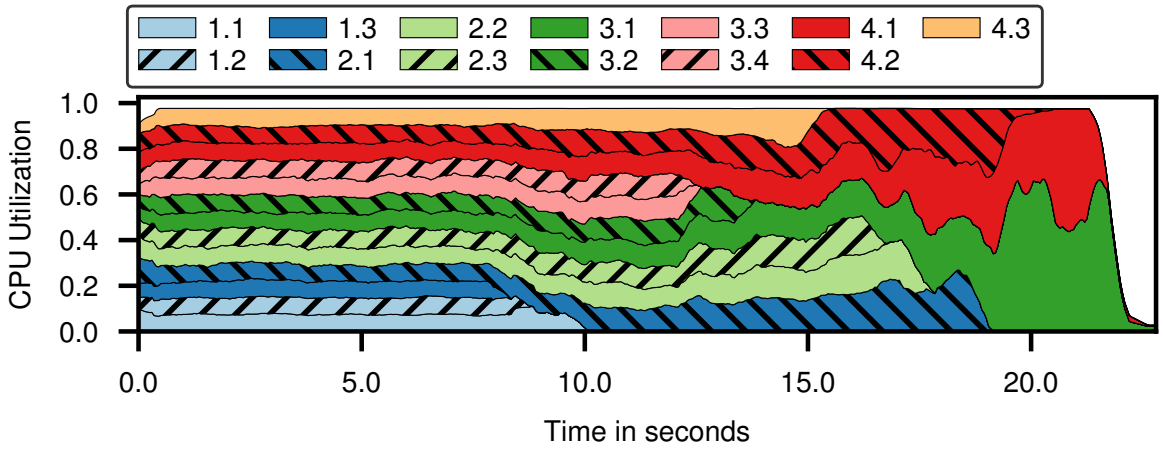



Figure 3.3: **CPU utilization of queries in fair policy**

### 3.8.1 Quickstep Specifications

We now describe Quickstep's configuration parameters that are used in the experiments. All 40 threads in the system are used as worker threads. The buffer pool is configured with 80% of the available system memory (126 GB). Memory for storage blocks, temporary tables, and hash tables is allocated from the buffer pool. The block size for all the stored relations is 4 MB. We preload the buffer pool before executing the queries, which means that the queries run on "hot" data.

### 3.8.2 Experimental Workload

For the evaluation, we use the Star Schema Benchmark (SSB) [95]. We justify the choice of SSB for our evaluation in Appendix A.7. We use two variants of the SSB SF 100 dataset, namely uniform and skewed. For the skewed dataset, the skew is introduced in the *lo_quantity* column of *lineorder* table, as described by Rabl et al. [106]. In the uniform dataset, each value in the domain [1, 50] is equally likely to appear in the *lo_quantity* column. In the skewed dataset, 90% values fall in the range [1, 10].

### 3.8.3 Evaluation of Policies

In this section, we evaluate the policies that are currently implemented in the system. Specifically, we verify if the actual CPU allocation among queries is in accordance with the policy specifications. To calculate CPU utilization, we use a log of start and end times for all work orders.

#### 3.8.3.1 Fair

In this experiment, we execute all 13 queries from the SSB concurrently using the fair policy. As described in Table 3.1, the policy specification implies a fair sharing of CPU resources among concurrent queries. The CPU utilization of the queries is depicted in Figure 3.3. As we can see, the CPU utilization of all the queries remains nearly equal to each other during the workload execution, despite queries belonging to different query classes with varying query complexities.

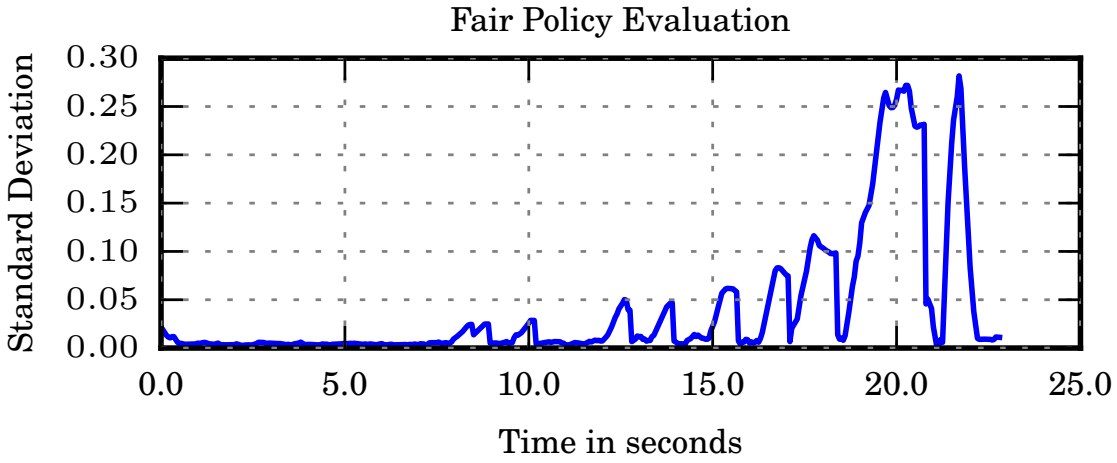



Figure 3.4: **Standard deviation of CPU utilization of active queries for fair policy**

Notice that the available CPU resources also get automatically distributed elastically among the active queries (e.g. at the 10 and 18 seconds marks) when a query finishes its

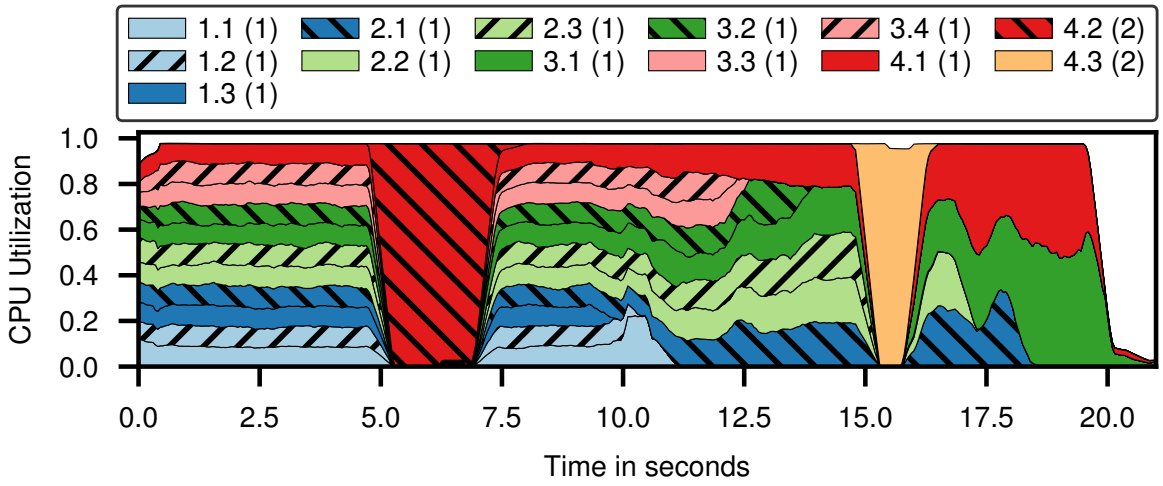



Figure 3.5: **CPU utilization in HPF policy. Note that $a.b(N)$ denotes a SSB query $a.b$ with priority $N$**

execution. This elasticity behavior allows Quickstep to fully utilize the CPU resources at *all* times.

In order to quantify the fairness metric, we compute the standard deviation of the relative CPU utilizations of queries. At a given instant, we only consider the active queries and compute the standard deviation of their relative CPU utilizations.

The result for this experiment can be found in Figure 3.4. Notice that for the first 7 seconds when all 13 SSB queries are under execution, the standard deviation of CPU utilizations is nearly zero, which means the fair policy gives all queries nearly equal share of CPU. As more queries finish their execution, there is a transient phase (notice the peaks) after which the standard deviation drops to low levels. This experiment also reinforces the previous results quantitatively, and shows that the implementation is able to meet the policy specifications.

#### 3.8.3.2 HPF

Now we validate if the implementation matches the specification of the HPF (cf. Table 3.1) policy.

All queries have the same priority value (1) except Q4.2 and Q4.3 which have a higher priority value (2). The execution begins with 11 queries having the same priority value. We inject Q4.2 in the system at around 5 seconds and Q4.3 at around 15 seconds. Figure 3.5 shows the CPU utilization of queries during the workload execution.

As the high priority queries arrive (at the 5 and 15 seconds marks), the existing queries pause their execution and the scheduler makes way for the higher priority query. As the higher priority queries finish their execution (i.e. at the 7 and 16 seconds marks), the paused queries resume their execution.

The result of this experiment also demonstrates that Quickstep's scheduler design naturally supports query suspension, which is an important concern in workload management.

### 3.8.4 Proportional Priority Policy

Now we examine the scheduler's behavior to the proportional priority policy in which the higher priority integer implies higher importance.

We pick two queries from each SSB class, and assign them a priority value. Our priority assignment reflects the complexity of the queries from the corresponding class. For instance, query class 1 has one join, class 2 has two joins and so on. Recall that in our implementation a higher priority integer implies higher importance.

Figure 3.6 shows the CPU allocation among concurrent queries in the proportional priority policy. We can see that a higher priority query gets proportionally higher share of CPU as compared to the lower priority queries. When all queries from the priority class 8 finish their execution (11 seconds), the lower priority classes elastically increase their CPU utilization, so as to use all the CPU resources. Also note that among the queries belonging to the same class, the CPU utilization is nearly the same, as described in the policy specifications.

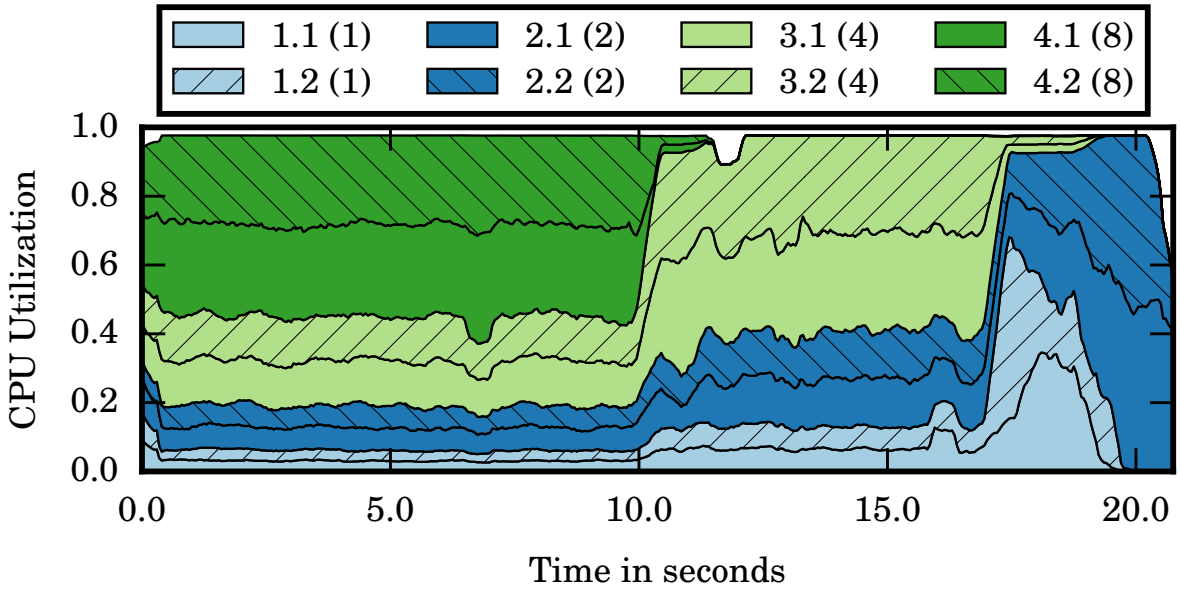



Figure 3.6: **CPU allocation for proportional priority policy. Note that $a.b(N)$ denotes a SSB query $a.b$ with priority $N$**

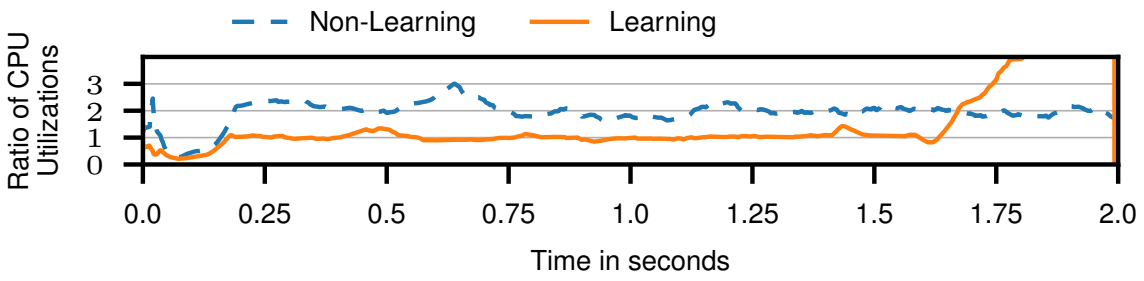



Figure 3.7: **Ratio of CPU utilizations $\frac{Q4.1}{Q1.1}$ with and without learning implementation**

### 3.8.5 Impact of Learning on the Relative CPU Utilization

In this experiment, we compare the learning-based scheduler with a static non-learning based implementation (baseline). We perform the comparison using fair policy, which should be the easiest policy for a static method to realize.

In the baseline, the probability assigned to each query remains fixed unless either a query is added or removed from the system. If there are $N$ active concurrent queries in the system, each query gets a fixed probability $1/N$.

We run $Q1.1$ and $Q4.1$ concurrently with the fair policy using both the learning and non-learning implementations. Our metric for this experiment is the ratio of CPU utilizations of $Q4.1$ and $Q1.1$. As per the policy specifications, the CPU utilization for both queries in the fair policy should be equal. Figure 3.7 shows the results of this experiment.

Observe in Figure 3.7, that the ratio in the non-learning implementation is closer to 2, meaning that the implementation is biased towards $Q4.1$. This behavior stems from the fact that the time per work order for $Q4.1$ is higher than $Q1.1$ (c.f. Figure A.2 in Appendix A.4). In contrast, the ratio of CPU utilizations in the learning implementation is nearly 1. The learning based implementation can identify various phases in query execution for both the queries and adaptively change the CPU allocation as per the changing demands of the queries. The non-learning implementation however fails to recognize the fluctuations in the CPU demands of queries and therefore does an unfair allocation of CPU resources.

### 3.8.6 Impact of Learning on Performance

Here we analyze the impact of the learning-based approach on the performance of queries. We use two query streams, one for $Q1.1$ and another for $Q4.1$. As one instance of $Q4.1$ finishes execution, another instance of $Q4.1$ enters the system (likewise for $Q1.1$). We compare the throughput for both $Q4.1$ and $Q1.1$ using the learning implementation of the fair policy against its non-learning implementation.

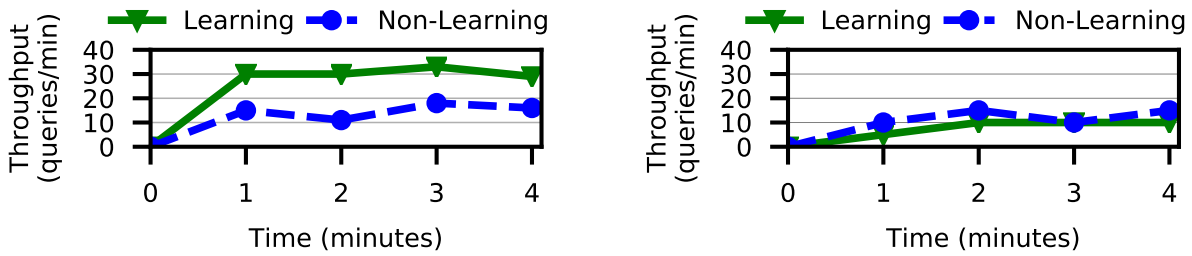
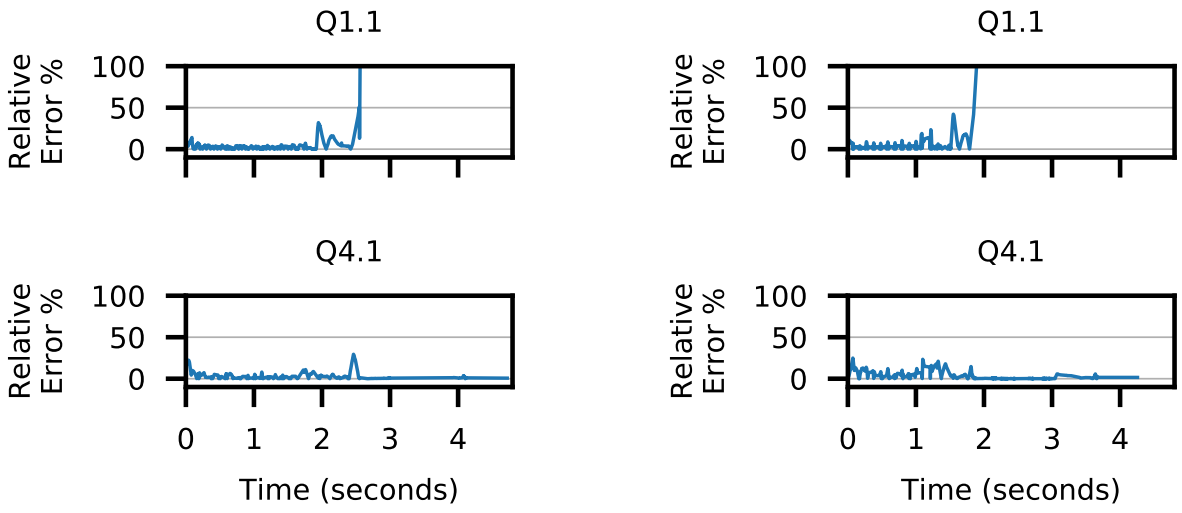



Figure 3.8: **Impact of learning on the throughput**

Figure 3.8 plots the result of this experiment and shows the throughput for each query stream. The throughput for the $Q4.1$ stream is not affected considerably by the choice of the implementation. However using the learning implementation, the throughput of the $Q1.1$ stream improves significantly (up to 3x better than the non-learning implementation). The reasons for the improvement are as follows: Following the result of the previous experiment (cf. Figure 3.7), in the non-learning implementation, $Q1.1$ which has shorter work orders, is starved of CPU resources due to $Q4.1$, which has longer duration work orders. In the learning-based implementation however, the $Q1.1$ stream gets its fair share of CPU resource (more than that in the non-learning implementation). Therefore, $Q1.1$'s performance is improved, resulting in its increased throughput.

This experiment highlights a two-fold impact of the learning module – first, it plays a crucial role in the fair policy enforcement. Second, it improves performance of queries with lower CPU requirements when they are competing with queries with higher CPU demands, thereby also increasing overall system throughput with such mixed and diverse workloads.




(a) Skewed dataset

(b) Uniform dataset

Figure 3.9: **Comparison of predicted and observed time per work order**

### 3.8.7 Experiment with Skewed and Uniform Data

In this experiment we test the learning capabilities of the Quickstep scheduler under the presence and absence of skew (skew description in Section 3.8.2). We execute $Q1.1$ and $Q4.1$ on the skewed and uniform data. We sort the skewed *lineorder* table on the *lo_quantity* column, to amplify the impact of skew. For the predicate $lo\_quantity \leq 25$ on the skewed table, some blocks have high selectivity and others have low selectivity.

We compare the predicted work order times for each query with its observed work order times. Figure 3.9 presents the results of this experiment, with relative error of the prediction on the Y-axis and time on X-axis. We can see that the relative error is very low in both the datasets for both queries. The execution of $Q1.1$ with skewed data takes longer than the uniform dataset. The intermediate peaks in the relative error correspond to phase change in the execution plan. Note that the scheduler learns the phase changes quickly, and adjusts its estimates after each phase change.

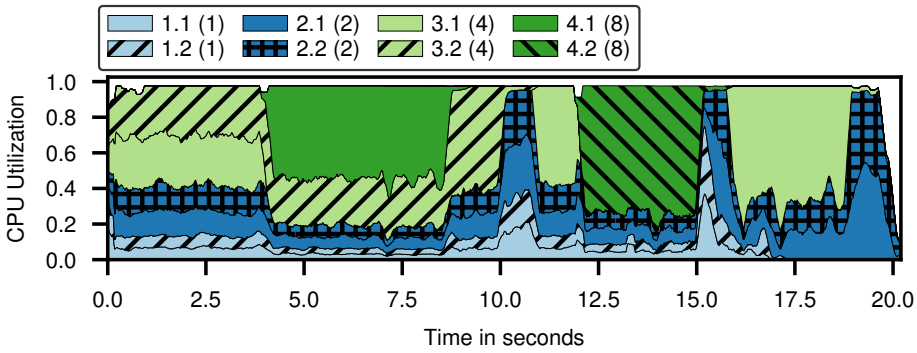



Figure 3.10: **Load control: An SSB query $a.b$ with priority $N$ is denoted as $a.b(N)$**

### 3.8.8 Load Controller

In this experiment, we evaluate the effectiveness of the load controller. We use two queries from each SSB query class. The priority value assigned to a query reflects its complexity e.g. proportional to number of joins in the query plan. We configure the load controller with the threshold for suspending queries as 56 GB. As the buffer pool size grows close to the threshold, the load control mechanism kicks in.

Quickstep's buffer pool stores the relational tables as well as hash tables used for joins and aggregations. If the requested memory cannot be allocated, the load controller can suspend a query with the highest memory footprint. In the current implementation, we check for reactivating the suspended query upon every query completion. Figure 3.10 shows the CPU utilization of the queries.

The execution begins with 6 queries. At around 4 seconds, a higher priority query $Q4.1$ enters the system. At this point $Q3.1$ has the highest memory footprint and the load controller picks it as the victim for suspension. We can observe in Figure 3.10, that from 4 to 11 seconds, the CPU utilization of $Q3.1$ is zero, reflecting its suspended state. The same pattern is repeated as another high priority $Q4.2$ enters the system at around 12 seconds. Once again $Q3.1$, that has the highest memory footprint, is suspended in order to allow $Q4.2$ to enter the system. Observe that in the 12 to 15 seconds time interval, $Q4.2$ gets executed and the suspended query $Q3.1$ doesn't utilize any CPU resource.

This experiment demonstrates the load control capabilities of the Quickstep scheduler. It stresses an important feature of our scheduler, which integrates load-controller functionality. Thus, admission control and query suspension is handled holistically by the scheduler.

## 3.9 Related Work

The *work orders* abstraction is similar to other abstractions like *morsels* in Hyper [74] and the *segment-based parallelism* [126]. Such abstractions provide a means to achieve high intra-query, intra-operator data parallelism. Hyper [74] uses a *pull-based* scheduling approach i.e. workers *pull* work (morsels) from a pool. We use a *push-based* model, where the scheduler controls the assignment of work to workers. The pull-based dispatch model suffices for executing one query at a time. However, a push-based model can be simpler to implement sophisticated functionalities such as priority-based query scheduling, incorporating a flow control across multiple pipelines, (as shown in [126]), cache-conscious task scheduling.

The elastic pipelining implementation [126] uses a *scalability vector* to vary the degree of parallelism of segments of the query plan. The scalability vector tracks the query performance when number of cores are varied, and it does not use any prediction technique. Their objective is to maximize the performance of a single query executed on a cluster. Our work focuses on resource sharing among concurrent queries, by enforcing policies using a learning-based approach. Additionally, we can accommodate estimates provided by other techniques.

There is related work [52] on ordering queries in a workload with different objectives such as fairness, effectiveness, efficiency and QoS. This work is complimentary to our scheduler design as it deals with ordering the queries *before* they enter the system, where as we focus on scheduling *admitted* queries.

Several enterprise databases [6, 7, 5, 9, 2, 3] offer workload management solutions which classify queries based on their estimated resource requirements, encode resource

allocation limits as resource pools and map workloads to such resource pools. While such estimation methods can be used to complement our approach, our scheduler can also work without such detailed estimation techniques. Prior research in this area [67, 68] has focused on identifying misbehaving queries, prioritizing/penalizing queries to meet the service level objectives. Our load controller can be complemented with such functionalities.

Predicting query performance is an active area of research. Earlier work [130, 131, 43] includes analytical models based on the optimizer's cost models for both single query and multiple concurrent queries. By design, our Learning Agent can incorporate such techniques, but can also function without them. More accurate work order execution time estimates can further improve adherence to the policy specifications.

Scheduling problem has also been studied in the OS and the networks community. Our scheduler's probabilistic framework is inspired by the seminal lottery scheduling [125] in which different processes are assigned certain number of lottery tickets, A lottery is conducted after every fixed time intervals and the winner process gets to execute in the next quantum.

A key difference in lottery scheduling and our work is that the OS scheduling is usually preemptive. The OS maintains a process context that captures the state of the preempted process. Quickstep's scheduling is non-preemptive, which means once a work order begins its execution on a CPU core, it continues to do so until completion. Non-preemptive scheduling provides us an exemption from maintaining work order context (similar to process context), thereby simplifying the relational operator execution algorithms.

Quickstep's design philosophy is to make scheduling transparent and fine-grained, and to decouple mechanism from policies [71] - a common theme found in the OS literature.

Deficit Round Robin (DRR) [115] is a technique for network packet scheduling. Our usage of work order execution time as a metric is similar DRR's usage of packet sizes. However DRR scheduling is inherently round robin based (with additional maintenance of quantum information), where as our scheduling is based on dynamic probabilities.

# Chapter 4

# Revisiting Pipelining for In-memory Database Systems

## 4.1 Introduction

Pipelined execution of operators in a query plan is ubiquitous in database management systems [60, 63, 100, 126, 137]. A key aspect of pipelining that is exploited by many systems is the prevention of materialization of intermediate results of an operator. In a disk-based system, read and write operations involving disk are expensive. Thus pipelining significantly improves query performance in disk-based systems. In the in-memory setting, query processing involves little or no disk involvement. Thus data movement across memory hierarchies (e.g. caches to memory) is no longer a bottleneck in query execution. Pipelining has been studied extensively in the disk setting and shared-nothing environments [29, 39, 40, 47, 48, 114] and the interest in pipelined query processing has continued in the in-memory settings [23, 80, 126, 129].

In this work, we revisit the notion of pipelining in the in-memory settings, and present an in-depth study on the effect of pipelined and non-pipelined execution on the performance of a query. Our focus is on systems that use block-based query processing technique. Block-based query processing is used in many systems including Quickstep [100], MonetDB [60], Hyrise [4], Hive [58], Impala [66], and RocksDB [42].

We first describe the basics of pipelining in query processing. In a query execution, data are often streamed between two or more physical relational operators. One of the operators in the pipeline acts as a producer operator, that processes the data and streams it to the consumer operator for further processing. When the consumer can start processing

its input data, even before the producer has processed all of its input, we say that the in the query plan, these relational operators form a pipeline. A commonly found pipeline found in query plans is made up of selection operator and probe (hash join) operator.

We compare the importance of pipelining in query processing in the disk setting and in-memory setting through an example below.

**Example 1.** *Let us consider a query which has 1 million intermediate tuples. We assume that in a disk-based system that uses pipelining, the query runs for 10 minutes. Without pipelining, all tuples need to be stored to the disk and brought back. We assume the combined disk read and write latency to be 1 millisecond. The time to write the tuples to disk and reading them back to memory is 1 million $\times$ 1 millisecond = 1000 seconds.*

Note that in Example 1, the overhead of materializing the intermediate tuples to disk and reading them back is almost twice the query execution time itself. Thus there is a big incentive to pipeline the intermediate data, in order to improve query performance. Let us consider the same query for in-memory systems.

**Example 2.** *Consider the earlier query in the in-memory settings by assuming the run time of the query to be 10 seconds. We assume the combined read and write latency from cache to memory to be 10 nanoseconds. Without pipelining, the overhead of writing and reading intermediate data is 1 million $\times$ 10 nanoseconds = 10 milliseconds.*

The materialization overhead which was nearly 100% in the disk setting, is merely 0.1% in the in-memory setting.

Admittedly our example is simple and glosses over many important details. However it raises an important question: Is pipelining for in-memory setting as important as the disk setting? Answering this question turns out to be challenging, primarily because a combination of factors jointly impacts the performance of a query when there is no I/O bottleneck. We first identify these factors, which are parallelism, block size, storage format, sequence of pipelines in a query plan, and hardware characteristics. The collective space of combinations of these dimensions is large. Through our study, we explore this space by

identifying the impact of these dimensions on the relative performance of pipelining and non-pipelining strategies.

Prior work has looked at individual dimensions and studied their impact on overall query execution. (Related work is in Section 4.2). However to the best of our knowledge, this is the first work aimed at finding the difference between pipelining and non-pipelining strategies, while also studying the impact of various dimensions on this comparison.

There are multiple dimensions associated with pipelining in the in-memory settings. We give a brief overview of these dimensions next, a more elaborate discussion is present in Section 4.4.

**Block size** Block size refers to the unit of storage for the data movement in query execution. A small block size results in a more frequent pipelining and vice-versa.

**Storage Format** The storage format of stored tables play an important role in query performance. We look at two storage formats: row store and column store and study their impact on pipelining.

**Pipeline Sequence** A query plan often has multiple pipelines connected to each other, such that there can be several permutations of these pipelines for execution; each with potentially different performance characteristics.

**Parallelism** Intra-operator parallelism is an important technique to leverage multi-core hardware and improve performance. We look at aspects such as scalability of operators with varying parallelism and their impact on the two pipelining strategies.

**Hardware Characteristics** Modern database systems try to leverage features provided by the hardware. We look at one such feature which is prefetching and study its impact on pipelining.

Studying these factors in a single systems requires implementation of pipelining and non-pipelining styles of query processing in the same system. Systems often have a fixed way of coordinating work execution in a pipeline, which makes the comparison a challenging task.

Our study is done in a system called Quickstep [100] (system background presented in Section 4.3). A key aspect that enables us to perform this comparison is Quickstep's novel scheduler design (described in detail in earlier works [100, 37]). The scheduler relies on a work order abstraction for representing work to be done in a query. The work order abstraction gives the Quickstep scheduler a powerful control over query execution. We leverage a key insight that pipelining and non-pipelining strategies result in different schedules of work. Thus by changing the scheduling strategy, query execution in Quickstep can be done in pipelined or non-pipelined way.

We compare the performance of pipelining execution strategy to a non-pipelining execution strategy through various combination of knobs. The pipelining and non-pipelining strategies differ in terms of the *eagerness* with which the output of producer is processed by the consumer. We also present an analytical model which takes into account many aspects such as cache miss penalties, block size, number of threads. We perform micro-benchmarking experiments as well as execute TPC-H queries on Quickstep. Although our analysis is based on TPC-H query plans generated by Quickstep, we believe that our methodology should be broadly applicable to other systems.

**Summary of results and contributions** : We summarize our contributions as follows:

- We show that the performance of pipelining in the in-memory environments for systems using block-based query processing depends on many dimensions like parallelism, block size, storage format and hardware characteristics like prefetching.

- The benefits of pipelining on query performance cannot be considered in isolation, rather they depend a lot on the query structure.

- As block sizes increase, pipelining and non-pipelining strategies start to behave similarly. This is a surprising result given the amount of attention pipelined query processing has received in the past.

- We propose an analytical model to compare the differences between pipelining and non-pipelining strategies. We perform micro-benchmarking experiments and their results ustify the conclusions from the analytical model.

This chapter is organized as follows: In Section 4.2 we provide a background about pipelining in database systems and also discuss related work. We give a brief overview of Quickstep in Section 4.3. We discuss the dimensions associated with this study in Section 4.4. We present an analytical model to understand the behavior of pipelining and non-pipelining strategies along with micro-benchmarking results in Section 4.5. In Section 4.6, we present the experimental evaluations. Section 4.7 summarizes our experimental findings.

## 4.2 Pipelining Background and Related Work

In this section we give a background about pipelining in database systems and discuss the related work.

A simplest pipeline of relational operator consists of two operators: A producer operator and a consumer operator. The output of the producer operator can be passed (or *streamed*) to the consume operator. An example of such a simple pipeline can be a hash based join. If there's a selection on the probe side, the selection operation and the subsequent probe operation can form a pipeline (assuming the hash table is already built).

Deeper pipelines may consist more than two operators, such that any two adjacent operators can form a producer and consumer pair. Data from the original producer operator can be passed all the way to the last operator in the pipeline. An example of such deep pipeline could be a left-deep plan for a multi-way join query.

There are two aspects about pipelining: *Materialization* (or the lack of) and *eager* execution of consumer operator on the output of the producer operator. Note that various

systems may have different representations for the temporary data, which is the output of a producer operator. Systems such as MonetDB [60] and Quickstep [100] that operate with the block-style processing model fully materialize the output. Vectorwise [137] has a compact representation of the intermediate output and does not fully materialize the ouptut. Systems such as Hyper [63], LegoBase [65] generate compiled code for the full pipeline, therefore they do not need to have a representation for the temporary data.

Pipelining in database systems has been studied extensively. Most recently Wang et al. [126] proposed an iterator model for pipelining in in-memory database clusters. Their key idea is to provide flexibility in the traditional iterator through operations such as expand and shrink. Neumann [93] proposed compilation techniques for query plans, which is used by Hyper [63, 74]. As we mentioned earlier, query compilation is one of the techniques for realizing pipelining in a query plan. Vectorwise [137] pioneered the vectorized query processing model through the hyper-pipelining query execution [23]. Breaking up from the tradtitional tuple-at-a-time processing model, Vectorwise used batches (or vectors) of tuples. These batches could be amenable to parallel SIMD instructions and thus could provide greater speedups over Vectorwise's predecessor MonetDB [60].

Menon et al. propose Relaxed Operator Fusion model [84] that puts together techniques like compilation, vectorization and software prefetching in a single query processing engine Peloton [8].

There is large body of prior work on the effect of storage format and page layouts on query performance [27, 14, 53, 51, 12]. In our work, we only focus on using row store and column store format for the comparison between pipelining and non-pipelining.

Incorporating parallelism for query execution within single node database deployment has been an active area of study since the prevalence of multi-core computing which is exemplified by many modern systems [100, 74, 137, 4].

Liu and Rundensteiner [79] study pipelined parallelism in bushy plans and propose alternatives to maximal pipeline processing. Our work differs from them in multiple aspects: We focus on single node in-memory query execution with large intra-operator parallelism.

We focus on the query scheduler phase, which comes after the optimal query plan has been generated by the optimizer. Their work focuses on optimizing query plans in the distributed execution environment with limited memory per node.

Zhu et al. propose look ahead techniques to increase robustness of query plans [136] in the Quickstep system [100]. Their key technique is to minimize the data that passes from the producer operator to the consumer operator in a pipeline through the help of bloom filters.

Work sharing across queries has been studied extensively. [55, 138]. Scan sharing has shown to have significant improvements in query performance, especially in the disk-setting. Such sharing typically happens for the selection operation on large tables (e.g. lineitem in TPC-H schema), which in many query plans is the starting point of a pipeline.

## 4.3 Quickstep Background

In this section we provide a brief background of Quickstep and its implementation of different pipelining strategies. Quickstep is designed with a goal to get high performance for in-memory analytic workloads. One of the techniques used by Quickstep to get high performance is through large intra-operator parallelism.

Quickstep uses a cost-based optimizer to generate query plans. Joins in Quickstep use non-partitioned hash based implementation. The operators in Quickstep process a batch of input tuples, rather than one tuple at a time. Prior work [23] has shown that the vectorized style processing outperforms tuple-at-a-time processing technique.

Quickstep uses an abstraction called *work orders*, which represents the relational operator logic that needs to be executed on a specified input. The work done for a query is broken up in a series of work orders. These work orders can be executed independently and in parallel.

Quickstep has two kinds of threads - a scheduler thread and several worker threads. The worker threads execute these work orders. The scheduler thread coordinates the execution of work orders which includes dispatching the work orders to worker threads

and monitoring the progress of query execution. Once assigned a work order, the worker thread executes it until its completion.

### 4.3.1 Managing Storage in Quickstep

Quickstep supports a variety of storage formats such as row store, column store with the optional support of compression. The data in a table is horizontally partitioned in small independent storage blocks, as proposed in some earlier designs [27, 100]. The size of each storage block is fixed, yet configurable. The intermediate output of relational operators (e.g. filter) is stored in temporary output blocks, which follow a similar design as the storage blocks of the base tables.

Each relational operator work order has a unique set of input, described based on the semantics of the operator. For instance, a select work order's input consists of a storage block and a filter predicate. A probe join hash table work order's input is made up of a pointer to the hash table and a probe input block. A work order execution involves reading the input(s), applying the relational operator logic on the input(s) and writing the output to a temporary block.[1]

Quickstep maintains a thread-safe global pool of partially filled temporary storage blocks. During a work order execution, a block is *checked out* from the pool, output gets written to the block, and it is returned to the pool at the end of the execution. Therefore a block is used by atmost one operator work order simultaneously. This approach has two benefits: first, we maintain locality of output block when output gets written to it and second, by reusing output blocks memory framgmentation is minimized.

### 4.3.2 Pipelining Implementation in Quickstep

Quickstep's implementation of pipelining is at a block level. As described earlier, the output of a relational operator work order is stored in temporary blocks. As soon as a

[1]Output of majority of the operators is represented in the form of storage block, except when the output itself is a data structure like hash table; in the case of a build hash operator, or hash-based aggregation operators.

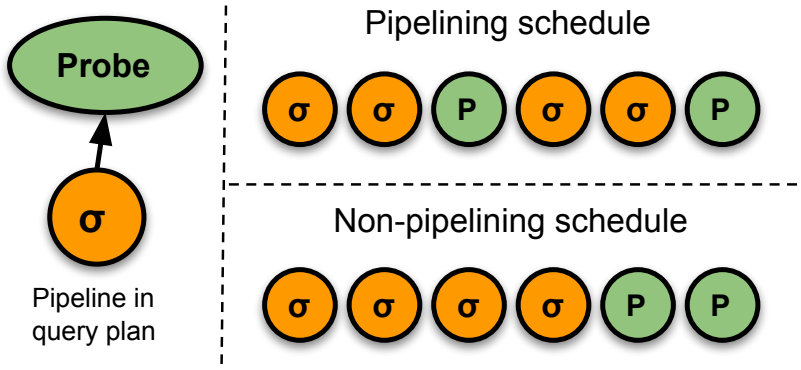



Figure 4.1: **Interplay between scheduling strategies and pipelining behavior. A sample pipeline of a filter operator ($\sigma$) and a probe operator (P) for a hash join is shown on the left. On the right are two possible interleaving of the work orders of these two operators, resulting in pipelining and non-pipelining schedules**

block is full, it is deemed ready for pipelining. The scheduler receives a signal as soon as an intermediate output block gets filled, after which it dispatches a work order for the consumer operator for execution.[2]

### 4.3.3 Pipelining and Scheduling

The scheduler for Quickstep can implement different scheduling strategies that can have an impact on the sequence in which different work orders for different operators are executed. We view pipelining and non-pipelining strategies as manifestation of two different scheduling strategies.

For pipelining behavior, a consumer operator work order is scheduled as soon as it is available. For the non-pipelining behavior, a consumer operator work order is not scheduled until all the corresponding producer work orders have finished execution.

The implementation of Quickstep scheduler allows to write more sophisticated scheduling policies, such as pipelining implementation with an upper or lower limit on the number of concurrent consumer work orders under execution, or pipelining under a specified

[2]All partially filled blocks are pipelined at the end of the operator's execution.

memory budget. To keep our analysis focused, we only discuss the two extremes which are pipelining (eager) and non-pipelining (lazy) strategies.

## 4.4 Discussion on Dimensions

The performance of pipelining and non-pipelining strategies depends upon multiple factors associated with query processing. We broadly classify these dimensions in three categories: physical organization of data (storage format and block size), execution environment (parallelism and hardware characteristics), and structural aspects of query (pipeline sequence). In this section we discuss these dimensions.

### 4.4.1 Storage Format

Data processing time is impacted by the way data are organized. We look at two common storage formats: row store and column store. In column store format, values of a given column are stored in contiguous memory region. Accessing a single column which has a sequential access pattern, results in a good cache behavior. In row store format, all columns of a tuple are stored in a contiguous region. Thus accessing a particular column involves bringing unnecessary data (non-referenced columns) in the caches.

Column stores have shown to have higher performance for analytical workloads [12]. The difference between the performance of column store and row store format is prominent for scan operators. Typically scans are faster with column store than in row store.

Recent studies [100] have shown that the performance gap between column stores and row stores is not as high as shown in the prior work. Therefore we experiment with both storage formats. For our comparison, all the base tables in the TPC-H schema are stored in the same storage format. We use row store format for temporary tables irrespective of the storage format of the base tables.

### 4.4.2 Block Size

We start by explaning the concept of block size. As the producer operator processes the input, it materializes the output to a temporary block. We let the temporary block to reach a threshold size (known as block size) and then pass the block to the consumer operator.

We would like to examine the impact of block size on the performance of the pipelining strategies. Consider an example pipeline: select operator $\rightarrow$ probe hash table operator. A smaller block size in the pipelining strategy would mean that the block can potentially get filled quickly. Therefore, compared to a larger block size, the probe tasks may be scheduled more frequently and each task itself may be shorter.

### 4.4.3 Parallelism

Intra-operator parallelism is prevalent in many modern database systems [60, 74, 100, 137]. Pipelining encapsulates inter-operator parallelism, as it exemplifies simultaneous execution of producer and consumer operators in a pipeline. We would like to study the impact of parallelism on the relative performance of pipelining and non-pipelining.

We define some terminologies that are relevant for our study in the context of concurrency. *Degree of parallelism* (DOP) of an operator refers to the number of concurrent threads involved in executing work orders of that operator. The *scalability* of an operator (using $T$ threads) is its performance with DOP as $T$ relative to its performance when DOP is 1.

We now discuss the DOP behavior of operators in Quickstep. Recall from Section 4.3.3 that Quickstep has a scheduler that dispatches work orders of relational operators to worker threads. In Quickstep, the pipelining strategy can result in a fluid DOP of operators. Consider the example from Figure 4.1. Until there is enough input available to schedule a probe work order, all worker threads operate on filter operator, thus the DOP of the filter operator is same as number of threads in the system. As soon as a probe work order is generated it gets scheduled, at which time the DOP of the probe operator is 1. If

multiple probe work order are generated simultaneously, the DOP of the probe operator could be higher.

In the non-pipelining strategy, the DOP of operators remains roughly the same. As shown in Figure 4.1 for the non-pipelining schedule, the DOP of select and probe operator remain maximum (equal to number of threads).

#### 4.4.3.1 Scalability

In an ideal environment, adding more threads for an operator execution should offer linear speedup. The assumption being that each parallel work order operates at the same speed and thus by executing more tasks concurrently, the overall execution time reduces proportionally.

Linear speedups for operators (or for queries as a whole) are not always possible. DeWitt and Gray [41] propose reasons for less than ideal speedup for parallel databases such as startup costs, interference from concurrent execution and skew. We can extend some of their ideas to in-memory systems. For example interference can come from various sources such as contention due to latches, and shared use of a common bandwidth like memory bandwidth or bandwidth for data movement across NUMA sockets.

For an operator that exhibits poor scalability, increasing its DOP beyond a limit degrades its performance. Specifically the execution time for each work order of the operator increases. We observe that poor speedup can have contrasting impact on pipelining (positive) and non-pipelining (negative) strategies. The reason for such contrasting behavior lies in the difference in the DOP values of a consumer operator in query processing with and without pipelining. Recall from Figure 4.1 that pipelining yields in a lower DOP for the consumer operator, where as non-pipelining results in a higher DOP for the consumer operator.

### 4.4.4 Hardware Prefetching

We start by explaining what hardware prefetching is. It is a technique used by modern hardware to proactively fetch data in caches by speculating its access in the future. The prefetcher observes patterns of data accesses from memory to caches and speculates the access of a data element in advance. Prefetching hides the latency due to a cache miss and thus potentially improves performance. There are two kinds of prefetching: spatial and temporal, among which we focus on spatial prefetching.

Now we describe why prefetching is important to our study. Pipelining involves a context switch on the kinds of work orders that get executed (c.f. Figure 4.1). Thus pipelining may affect prefetcher's understanding of data access pattern. Therefore we are specifically interested in the impact of hardware prefetching on pipelined query processing.

In addition to the hardware-based prefetching implementation, there are software-based techniques for prefetching. There is prior work on using software-based prefetching to improve the performance of database operators [30, 84]. We restrict ourselves to hardware-based prefetching, as we can observe the impact of hardware prefetcher without modifying the implementation of the relational operators.

For our study, we run the queries with pipelining in two scenarios: a) hardware prefetching is enabled (default behavior of the hardware) b) hardware prefetching is disabled (by setting bit 0 and 1 in Model-specific Register (MSR) at address 0x1A4) as disclosed by Intel [124].

### 4.4.5 Pipeline Sequences

A query plan is composed of several pipelines that are connected to each other. To execute a query, there are many possible permutations of such pipelines each with potentially different performance characteristic.

To compare query performance with and without pipelining, we should ensure that both strategies use the same sequence. Therefore to come up with a sequence given a query plan, we present an algorithm. This algorithm generates a pipelining sequence that

maximizes the opportunity for pipelining. We call this algorithm MPS (Maximal Pipeline Sequence) Algorithm.

We make the following assumptions for the use of our algorithm. Only one pipeline gets executed at a time. Simultaneous execution of multiple pipelines could pollute the cache, thus making it harder to reason about performance. We assume a hash-based implementation of join algorithms. A logical join operation has corresponding two physical operators: a build operator that builds the hash table and a probe operator that lets the input probe the hash table. The probe operator does not begin its execution until the build operator's execution is over. We assume no disk spilling during a query execution.

**Algorithm 4.1** Pipeline Sequencing Algorithm

```
1:  function GETSEQUENCE(Operator op)
2:      if op.name() == select then
3:          return op.ID();
4:      end if
5:      if op.name() == build then
6:          return GetSequence(op.child()) + op.ID();
7:      end if
8:      if op.name() == probe then
9:          return GetSequence(op.buildOp)) + GetSequence(op.child()) + op.ID();
10:     end if
11: end function
```

Algorithm 4.1 describes the MPS algorithm which is a variation of depth first traversal of the query plan DAG. The key heuristics behind the algorithm is to construct the hash tables for the join early and facilitate the pipelining for the probing, by delaying the execution of the probe operator. There are two kinds of inputs for a probe operator - the probe relation and the hash table. By nature, the accesses within the hash table are random, where as the access pattern for the probe relation is sequential. Therefore during the

probe operation we prefer probe relation being hot in caches, as opposed to the hash table being hot.

The function GETSEQUENCE produces a sequence of pipelines in a recursive manner. We describe rules for each kind of operator such as `select`, `build`, and `probe`. Each rule determines the position of the given operator w.r.t to the sequence generated by the subtree rooted at this operator. For example, a `select` operator terminates the pipeline sequence, or a `build` operator appends itself before its subtree.

## 4.5 Analytical Model

In this section we model the performance of queries with and without pipelining, using parameters such as cache misses, number of threads used for execution, and block size. Our model is primarily meant for in-memory environments, but it can be easily extended to disk settings, as we show in Section 4.5.2. We use two key ideas in formulating our model, which we describe below.

First idea is to focus on the difference between the two strategies while ignoring the similarities between them. Many operations are common to query processing with and without pipelining: e.g. the total cost of reading from L1 cache is the same irrespective of the schedule. As we are interested in a relative comparison of the two strategies, it is safe to ignore the costs of operations which are common to both strategies and focus on the additional work for each strategy that is not present in the other strategy.

Our second idea is that when dealing with multi-megabyte blocks in a sequential access pattern, the cost of L3 cache misses of reading the block is less than missing the entire block. This idea is based on the usefulness of hardware prefetching. As the block is read into memory, initial few tuples would incur an L3 cache miss, but we assume that the prefetcher can quickly detect the access pattern and thus beyond a point, the miss penalty would decrease.

We analyze a basic pipeline of length two, in which the producer is a select operator and the consumer is a probe operator for a hash-based join. This pipeline can be found in

| **Notation** | **Description** |
|---|---|
| $R_h$ | Cost of reading a block to memory hierarchy $h$ from lower level hierarchy |
| $AR_h$ | Amortized cost of reading a block sequentially to memory hierarchy $h$ from a lower level hierarchy |
| $W_h$ | Cost of writing a block to memory hierarchy $h$ from a higher level hierarchy |
| $IC$ | Cost of an instruction cache miss |
| $M_h$ | Cost of a data cache miss from memory hierarchy h |
| $N_{op}^{in}$ | Number of input blocks for operator $op$ |
| $N_{op}^{out}$ | Number of output blocks for operator $op$ |
| $T$ | Number of threads in the system |
| $B$ | Block size |

Table 4.1: **Notations used for the analytical model**

many TPC-H queries like Q07 and Q19 in which the selection is performed on the *lineitem* table and the output is subsequently used to probe a join hash table. Pipelines of higher lengths and/or different operators can be dealt with similarly. Table 4.1 contains various parameters that we use to determine the costs for different scheduling strategies.

In the pipelining based execution, the input for `probe` (which is the output of `select`) is presumed to be hot in caches while it is read to perform the probe operation. In the non-pipelining case, the output of `select` is not immediately consumed by the `probe`, thus an input probe block is likely to be cold in the caches when it is read for the `probe` operation.

Thus, in the no-pipelining case, the extra work done can be quantified as:

$$AR_{L3} * N^{in}_{probe} + p_1 * N^{in}_{probe} * M_{L3} + W_{mem} * N^{out}_{select}$$

Here's the explanation of terms: $AR_{L3} * N^{in}_{probe}$ indicates the total cost of reading `probe` blocks sequentially from the memory, expressed as the amortized cost of reading a block sequentially times the number of blocks.

Note that a `probe` task has two input components: `probe` input block and a hash table. As the reads in a hash table are random, it disrupts the sequential access pattern used for reading the `probe` input blocks. Therefore we account for the penalty caused in reading the `probe` input blocks as $p_1 * N^{in}_{probe} * M_{L3}$ where $p_1$ is the probability that there is a L3 cache miss for reading `probe` input after the context switch back from reading the hash table.

Finally, $W_{mem} * N^{out}_{select}$ is the penalty incurred in writing the output of the `select` operator from cache to memory in the non-pipelining case. Next we quantify the additional work done In the pipelining case:

$$N^{out}_{select} * IC + N^{in}_{probe} * IC +$$
$$p_1 * N^{out}_{select} * M_{L3} + p_2 * (R_{mem} + W_{mem}) * N^{out}_{select}$$

Notice that in a pipelined execution, every `probe` task execution involves two context switches: First from `select` to `probe` and another from `probe` to `select`. Thus we

account for two instruction cache misses; one for each of such context switch, which is represented by the terms $N^{out}_{select} * ic$ and $N^{in}_{probe} * IC$.

Now we explain the term $p_1 * N^{out}_{select} * M_{L3}$, which represents the cache misses due to disruption in sequential access pattern of a `select`, due to the intermittent `probe` operations. The term $p_1$ is the probability of an L3 cache miss for `select` after the context switch back from `probe` to `select`.

Finally the term $p_2 * (R_{mem} + W_{mem}) * N^{out}_{select}$ reflects the impact of block size on the reading and writing of `probe` input blocks. As multiple threads share the L3 cache, each write for creating `probe` input, and subsequent `probe` input read may not be guaranteed to be served from the L3 cache especially when the block sizes are higher. The term $p_2$ represents the likelihood that the reads and writes incur L3 cache misses, and is expressed as $min(1, 2b * T/size(L3))$. The term $p_2$ is smaller for smaller block sizes, and it is 1 for large block sizes and when $T$ is high.

### 4.5.1 Difference between strategies for large blocks

We now look at the difference between two strategies while focusing on their behavior for large block sizes. We make few observations below that can help simplify the difference of two strategies. As the block sizes are typically of few megabytes, the instruction cache miss penalty get amortized, thus we can ignore the penalty associated with instruction cache misses. Second, we observe that $N^{in}_{probe} = N^{out}_{select}$. Thus the ratio of costs of non-pipelining and pipelining strategies looks as follows:

$$\frac{AR_{L3} * N^{in}_{probe} + W_{mem} * N^{out}_{select}}{p_2 * (R_{mem} + W_{mem}) * N^{out}_{select}}$$

This ratio can be simplified to

$$\frac{AR_{L3} + W_{mem}}{p_2 * (R_{mem} + W_{mem})}$$

We consider few representative cases below to estimate the difference between the two strategies. For high block sizes (size $> \frac{|L_3|}{2*T}$), $p_2$ is close to 1. For such large blocks, the

amortized cost of sequentially reading a block to L3 ($AR_{L3}$) is similar to reading a block on its own from memory ($R_{mem}$), as prefetching is not going to provide much help. Thus we expect for larger block sizes, the difference between two strategies to be negligible.

We acknolwedge that our model does not adequately capture the behavior for smaller block sizes. Smaller block sizes result in a large number of blocks, which incurs a large overhead in storage management. Some examples for such overhead include creation cost of several blocks, maintaining references for blocks present in-memory, synchronization costs in the data structures for storage management etc. Moreover linux hugepages facilitate using pages with large sizes (such as 2 MB), so that the TLB misses are reduced. A detailed analysis of overheads for small block sizes is beyond the scope of our study.

### 4.5.2 Applying the Model to Disk Setting

Our model can be easily applied to the disk setting. We change the parameters from Table 4.1 appropriately to fit the disk setting. The terms $p_1$ and $p_2$ can be nearly 0, assuming that the hash table is always kept in the buffer pool. Thus, we can see that the additional work done in the non-pipelining setting is

$$R_{disk} * N^{in}_{probe} + w_{disk} * N^{out}_{select}$$

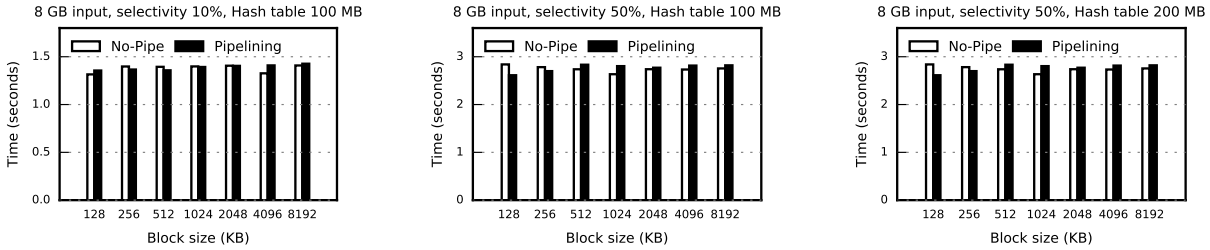
which could be in the order of seconds for thousands of blocks. The additional work done in the pipelining setting:

$$N^{out}_{select} * IC + N^{in}_{probe} * IC$$

is substantially less (order of nanoseconds or microseconds for thousands of blocks) than that in the non-pipelining case. Thus the analytical model is consistent with the expected behavior for disk-based systems.

### 4.5.3 Micro-benchmarking

To test the correctness of the analytical model, we use a micro-benchmark, which is written in C++ and it mimics a two operator pipeline of selection and probe operators.




(a) Selectivity 10%, 100 MB hash table (b) Selectivity 50%, 100 MB hash table (c) Selectivity 50%, 200 MB hash table

Figure 4.2: **Results of the micro-benchmarking experiments. The size of the base table is 8 GB.**

The micro-benchmark uses a tuple of single attribute (8 byte integer) and a C++ vector to abstract the storage. We can vary parameters used in the micro-benchmark such as the selectivity of the select operator, number of threads, and the size of the hash table.

The results for our micro-benchmarking study are presented in Figure 4.2. We use the same setup as described in Section 4.6.2 and use 10 threads on a single NUMA socket.

We can observe that the difference between pipelining and non-pipelining is small across all the variations. Second, for larger block sizes, the two strategies behave quite similarly which is in accordance to our explanation in Section 4.5.1.

## 4.6 Experimental Evaluation

We now describe the experiments to be conducted for this study. The goals for our experiments is to understand the performance characteristics of queries with and without pipelining and while doing so, observe the impact of the dimensions presented in Section 4.4.

### 4.6.1 Workload

We use queries and dataset from the TPC-H benchmark [10] for scale factor 50. Note that the query plans generated by Quickstep may result in different query plans, pipelines of varying lengths if compared with other systems. Therefore the results for pipelining analysis for Quickstep may not be directly applicable to other systems. However we believe that our methodology is broadly extensible.

### 4.6.2 Hardware Description

We now describe the hardware configuration and Quickstep specifications used for our evaluation. The hardware configuration of the machine that we used are described in Table 4.2.

| **Parameter** | **Description** |
|---|---|
| Processor | 2 Intel Xeon Intel E5-2660 v3 2.60 GHz (Haswell EP) processors |
| Cores | 10 physical, 20 with hyper-threading per socket |
| Memory | 80 GB per NUMA socket, 160 GB total |
| Caches | L3: 25 MB, L2: 256 KB, L1 (both instruction and data): 32 KB |
| OS | Ubuntu 14.04.1 LTS |

Table 4.2: **Evaluation platform**

Our analysis focuses on performance of single socket, and thus we use only one of the two NUMA sockets on the machine. Unless specified, we use all 20 threads as Quickstep workers. The buffer pool size of Quickstep is configured with 80% of the system's memory viz. 126 GB. We run each query 11 times, discard the first two runs and report the median of the remaining nine runs.

### 4.6.3 Results

In this section we present the results of our experimental evaluation. We focus on the following metrics: Overall query response time, execution time of a pipeline, and execution time of individual operator work orders.

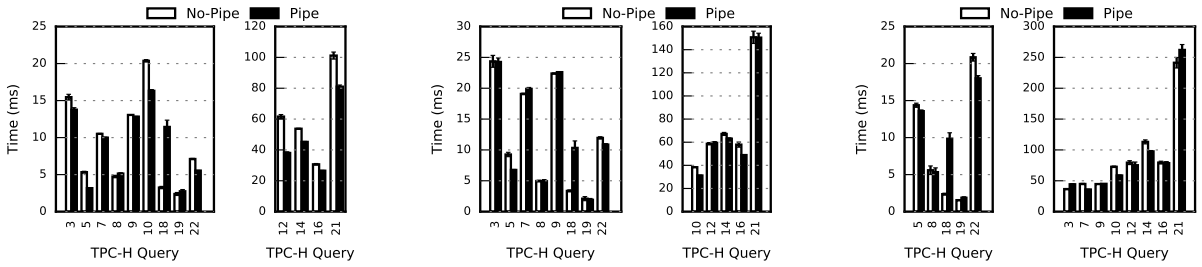
#### 4.6.3.1 Impact of pipelining on query performance

We are interested in measuring the impact of pipelining on a query's performance. The only observable metric for a user of the system is the response time of a query. Complex queries like the ones in TPC-H are made up of several pipelining consisting of multiple operators. Thus it is not immediately clear of the impact of pipelining on the entire query's response time. We claim that not all queries benefit from pipelining equally.

To analyze the impact of pipelining on a query's response time, we first look at where does time go in a TPC-H query execution. The intuition is that if there is only one operator in the query where majority of the query execution time is spent, pipelining may not play a big role in the overall execution time of the query. To make the analysis simpler, we analyze the most dominant operator (where the highest time is spent) and the second most dominant operator. Note that for this analysis, we run the queries in the non-pipelining mode, so that the percent time spent metric is accurate.

Figure 4.6b shows the results of this experiment, for tables stored in column store format. For some queries (Q1, Q6, Q13, Q15, Q17, Q19, Q22) the dominant operator takes up the majority of the query execution time (more than 50%). We also note that the dominant operator for many of these queries is a leaf operator (e.g. Selection on a base table, building a hash table on a base table, Aggregation on a base table). Thus, these queries may not really get benefits from the pipelining by the virtue of their query plan structures.

In many queries large data are pruned out at the beginning of the pipeline. Therefore not enough data is passed on to the consumer operators and hence the impact of pipelining is not significant.




(a) Block size 1 MB (b) Block size 2 MB (c) Block size 4 MB

Figure 4.3: **Performance comparison of probe hash operator when it is the first consumer operator in a pipeline**

#### 4.6.3.2 Performance of Consumer Operator

Pipelining should benefit the consumer operator as its input should be hot in caches. To verify if pipelining improves the performance of the consumer operator, we perform the following experiment.

We compare the work order execution time for the deepest pipelines in which the first consumer operator is a probe operator for a join. If there are multiple deep pipelines in a query with equal length, we pick the pipeline that has more number of tasks at the starting operator. The reason we pick probe hash table operator is that selection $\rightarrow$ probe pipeline is commonly found in query plans. To reduce noise, we consider only such probe operators for which there are at least 5 work orders.

Figure 4.3 plots the mean work order execution times for the first consumer operator. We can observe in Figure 4.3 that in general pipelining benefits the performance of the probe operator. The extent of improvement diminishes as we increase the block size from 1 MB to 4 MB. This behavior is consistent with the findings from the analytical model (c.f. Section 4.5).

#### 4.6.3.3 Performance of Deep Pipelines

Having looked at the performance of the consumer opertors, we zoom out to examine the performance of pipelines in each query. To do that, we measure the execution time of the deepest pipelines from each query. The execution time of a pipeline is defined as the time difference between two events: end of execution of the last work order of the last operator in the pipeline and beginning of the execution of the first work order of the first operator in the pipeline. Figure 4.4 shows the results of this experiment.

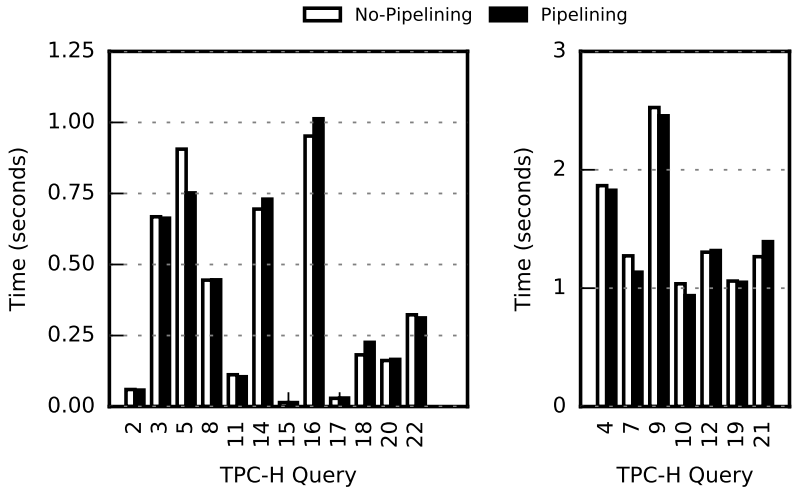



Figure 4.4: **Execution times of operator pipelines in TPC-H query plans using pipelining and non-pipelining strategies with block size 2 MB**

Recall that the pipelines in consideration are ones which are deep and the first operator in these pipelines has large amount of work. Hence these pipelines form a significant fraction of the total query execution time. We can observe from Figure 4.4 that the pipeline execution times in the pipelining and non-pipelining implementation are fairly similar, despite some individual consumer operator getting performance improvement due to pipelining (c.f. Figure 4.3).

#### 4.6.3.4 Overall Execution Time:

After analyzing the performance of deep pipelines, we further zoom out to look at the overall execution times of the queries. We compare the overall execution time of queries using pipelining and non-pipelining strategies.

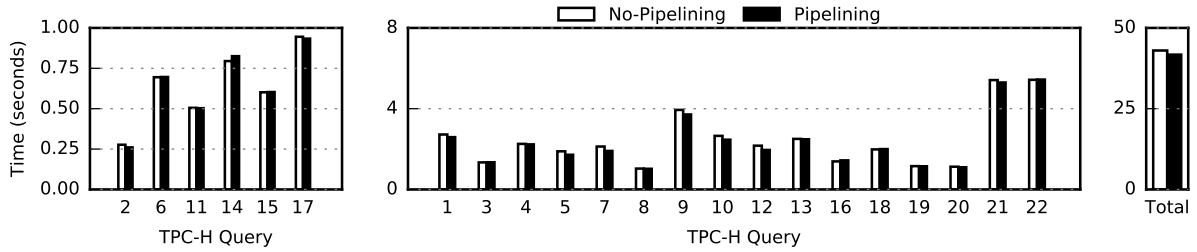


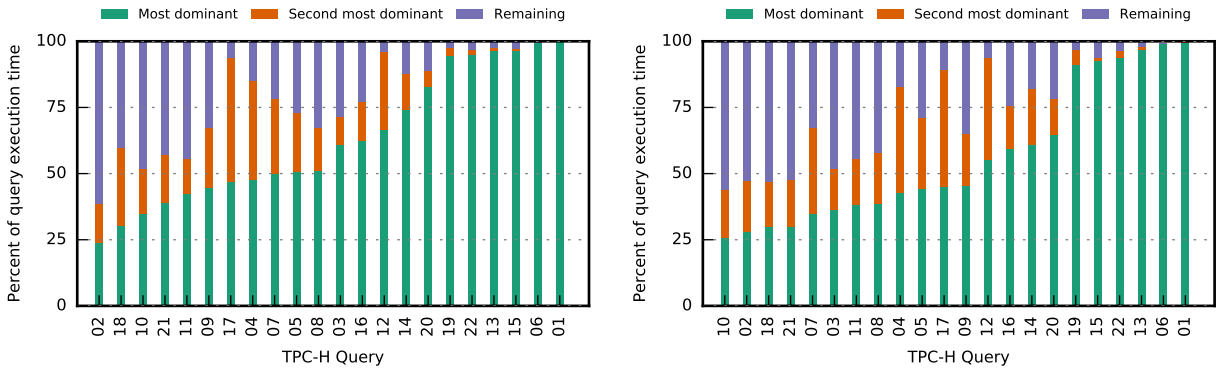

Figure 4.5: **Execution times of TPC-H queries with base tables in column store format and block size 2 MB**

The absolute times for all TPC-H queries for block size 2 MB are shown in Figure 4.5. We can observe that there is little to no difference between query execution times for pipelining and non-pipelining strategies.

From these experiments we can conclude that pipelining though benefits individual operators in a query, the overall impact of the choice of with and without pipelining is fairly minimal.

#### 4.6.3.5 Effect of Storage Format

Next, we study the effect of storage format of base tables on the pipelining performance. We use two configurations: a) All TPC-H tables stored in row store format b) All TPC-H tables stored in column store format. Then we compare the improvement of pipelining over non-pipelining for each configuration. Note that in both configurations, temporary tables are stored in row store format. We run the queries using 20 threads and block size of 2 MB.




(a) **Row store** (b) **Column store**

Figure 4.6: **Distribution of time spent in each TPC-H query among its operators**

Let us start by understanding the time distribution across operators when we use the row store format. We present the time distribution for the row store format in Figure 4.6a. Contrast with column store (similar distribution for column store is presented in Figure 4.6b). Typically selection operation is faster for tables stored in column store than in row store. Thus we can observe that the dominance of the selection operation on the overall query execution time is higher in case of row store, e.g. Q03, Q07, Q10, Q11, Q12, Q14, Q16, Q18, Q19, Q20. As a result, we expect that the impact of pipelining on overall query execution time should further diminish when using the row store format.

We plot the absolute execution times Figure 4.7. First, we can observe that column store queries are slightly faster than their row store counterparts. Second, the total query execution time is not significantly different between using pipelining and non-pipelining strategy.

Note that query plans start with accessing base tables in the leaf nodes. Base tables usually have more number of columns as compared to the intermediate tables, due to projections. As we progress upwards (from leaf to root nodes), the number of referenced columns keep decreasing. Thus the performance gap for operations on intermediate tables is narrow when using row store vs column store formats.

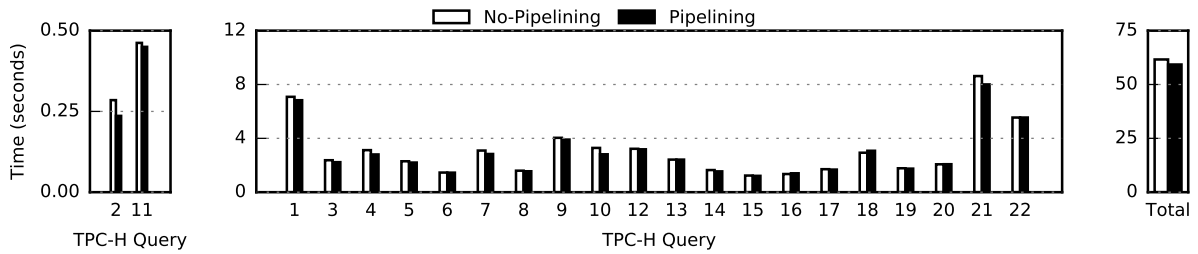


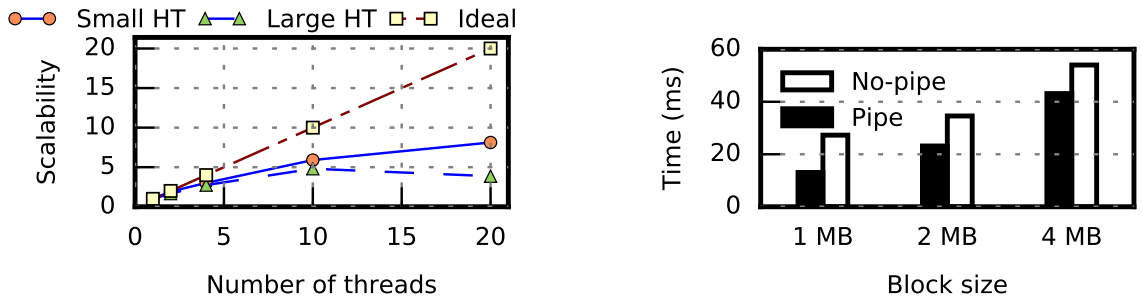

Figure 4.7: **Execution times of TPC-H queries with base tables using row store format and block size of 2 MB**

#### 4.6.3.6 Effect of Parallelism

Next, we study the effect of parallelism on the performance of pipelining and non-pipelining strategies. We run the TPC-H queries with 10 threads (instead of the 20 threads used for other experiments) and observe that the relative performance of pipelining and non-pipelining remains similar. Hence we don't report the results for experiments with 10 threads.

We present a result from TPC-H Q07 that shows the impact of scalability of operators on the performance of pipelining and non-pipelining strategies. In TPC-H Q07, the deepest pipeline has three probe operators, let us call them $P_1$, $P_2$ and $P_3$. One of these probe operators ($P_2$) involves probing a large hash table, which is constructed on entire *orders* table without any filter. The other probe operators ($P_1$ and $P_3$) have relatively smaller hash tables as input. Recall from Section 4.4.3.1, the degree of parallelism (DOP) of an operator in a pipeline is usually lower in the pipelining strategy.

We find the scalability of these operators and present them in Figure 4.8a. Note that though $P_1$ (small hash table, marker ●) has less than ideal scalability, it still gives some performance improvement as the number of threads increase. For $P_2$ (large hash table, marker ▲), the scalability curve dips as the number of threads increase. Therefore for $P_2$, increasing parallelism beyond a limit hurts its performance. The reason for such a poor scalability is the large size of its probe hash table, because of which results in a large random read traffic during the tuple reconstruction phase of the probe operator.

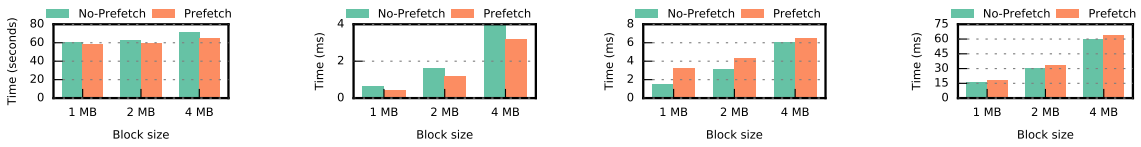



(a) Scalability (b) Execution time

Figure 4.8: **Scalability of two different probe operators in TPC-H Q07, compared with the ideal scalability. Also shown are the execution times for the probe operator with poor scalability in various settings**

To verify if our hypothesis regarding the impact of poor scalability on pipelining and non-pipelining (c.f. Section 4.4.3.1) is correct, we compare the mean work order execution time for the probe operator $P_2$ with poor scalability in pipelined and non-pipelined implementations. Figure 4.8b presents the result of this comparison. We can see that $P_2$ is faster with pipelining for all three block sizes. The difference between the two strategies is higher for lower block size and it reduces as the block size increases. For lower block sizes, the number of probe work orders are higher resulting in a larger contention which causes higher performance degradation.

This behavior leads us to the next question: What is the impact of poor scalability on TPC-H Q07's performance? The percent improvements of pipelining over non-pipelining in TPC-H Q07's execution times for different block sizes are 20% (1 MB), 12% (2 MB), and 9% (4 MB). The improvement depends on the position of the poorly scalable operator in a deep pipeline. If the operator is at the end of the pipeline with less work to be done (compared to the operators at the beginning of the pipeline), the extent of overall query time improvement is low.




(a) TPC-H execution (b) Selection (c) Build hash table (d) Probe hash table

Figure 4.9: **Performance of row store format with and without hardware prefetching. Shown above are total TPC-H execution times and median work order execution times for selection, probe and build hash operators in Q07 with various block sizes**

This experiment highlights an aspect of parallelism which causes performance difference between pipelining and non-pipelining strategies. Systems can have scalability issues due to various external (hardware interference, slow network) and internal factors (skew, poor implementation of operators). We would like to stress that the reason for poor scalability of the probe operator in Quickstep when the build side is large, is not our focus[3], rather it is the impact of poor scalability on the performance of pipelining strategies.

#### 4.6.3.7 Effect of Hardware Prefetching

We examine the effect of prefetching on the relative performance of pipelining strategy. We run this experiment using row store format and use three values for block sizes namely 1 MB, 2 MB and 4 MB. Figure 4.9 presents the improvements in query execution times due to hardware prefetching.

In the row store format, even if a single attribute is scanned, a lot of unnecessary data from the tuple is read. As row store tuples are fixed width[4], the hardware prefetcher can detect the access pattern of scanning a single attribute. We observe that hardware prefetching has a small impact on the total execution time, as shown in Figure 4.9a. Notice that the impact of prefetching on the total execution time increases as the block size increases.

[3]The specific probe operator in TPC-H Q07 can be made more scalable by partitioning the join or by putting the payload in the hash table bucket.

[4]variable length attributes are stored in a separate region, with a pointer to the region stored in the tuple

For a higher block size, the prefetcher can observe more data traffic per block and thus it can speculate and prefetch more efficiently.

Beyond the overall execution time, we are interested in understanding the impact of hardware prefetching on individual operators. We pick three operations from Q07 and compare their execution times with and without prefetching: selection (Figure 4.9b), building a join hash table (Figure 4.9c) and probing a join hash table (Figure 4.9d).

Our observations are as follows: Prefetching generally benefits selection operators. The extent of improvement increases as the block size increases. This behavior is understandable as selection has a sequential access pattern and the prefetcher can understand the strides in the memory accesses across tuples in the row store format.

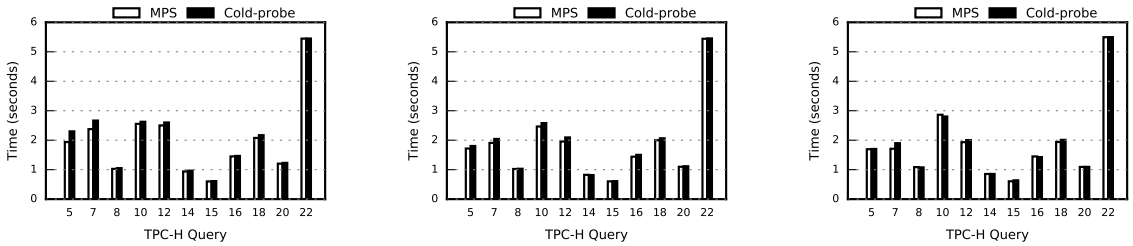
Prefetching seems to have no impact on probe hash table operator. A probe operation involves reading two data streams - a sequential read access pattern for the probe input and a random read access pattern for the hash table. Perhaps due to a mix of these two access patterns, probe operator does not seem to be benefited by hardware prefetching.

For the build hash table operation, prefetching seems to have an adverse effect as its work orders get slower with prefetching. A possible explanation for this slowdown could be the mixed access patterns of sequential read and random write behavior in the build hash operation.

We ran the same experiment for column store format and found little to no performance difference due to prefetching. We speculate that the prefetcher does not make any significant contribution to an already optimized access pattern of column stores.

### 4.6.4 Effect of Pipeline Sequences

We now evaluate the effect of using different sequences of pipelines in a query plan. Note that some TPC-H queries have a large plan (e.g. Q07, Q21), in which several sequences are possible. On the flipside, plans for some queries are so small such that we cannot produce multiple pipeline sequences (e.g. Q01, Q06). Thus to keep our analysis manageable, we produce pipeline sequences which can be given a semantic information.




(a) Block size 1 MB (b) Block size 2 MB (c) Block size 4 MB

Figure 4.10: **Comparison of query performances that use two different pipeline sequences. MPS refers to the pipeline sequence output by the MPS algorithm proposed in Section 4.4.5 and another sequence in which probe input is cold, but the hash table is hot at the time of first probe**

One such sequence is the output of the MPS algorithm proposed in Section 4.4.5. Recall that the heuristics for the MPS algorithm was to produce a maximal probe pipeline, in which the probe input would be hot in caches, as much as possible. We use another sequence in which the probe input is cold, however the hash table is hot.

We use the following criterion for selecting a query for this experiment: We should come up with multiple pipeline sequences from the given query plan. In the deepest pipeline, there should be a probe operator whose input is not a stored table, and the input itself should generate substantial work (at least 10 work orders).

In all the previous experiments, we use the pipeline sequence which is an output of the MPS algorithm. Thus we only show the results for the cold probe sequence in Figure 4.10. In majority of the cases, overall query performance remains similar in both sequences.

To find out if the cold probe sequence affects probe operator's performance, we plot the median execution times of the first probe operator in the deepest pipeline of the query plan. Figure 4.11 shows the results of this experiment. In some queries (e.g. Q05, Q07), the performance of the probe operator remains similar in both sequences. However in some other queries cold-probe sequence results in a slower probe operator (e.g. Q12, Q14) than its MPS counterpart. The small performance improvement in the probe operator's

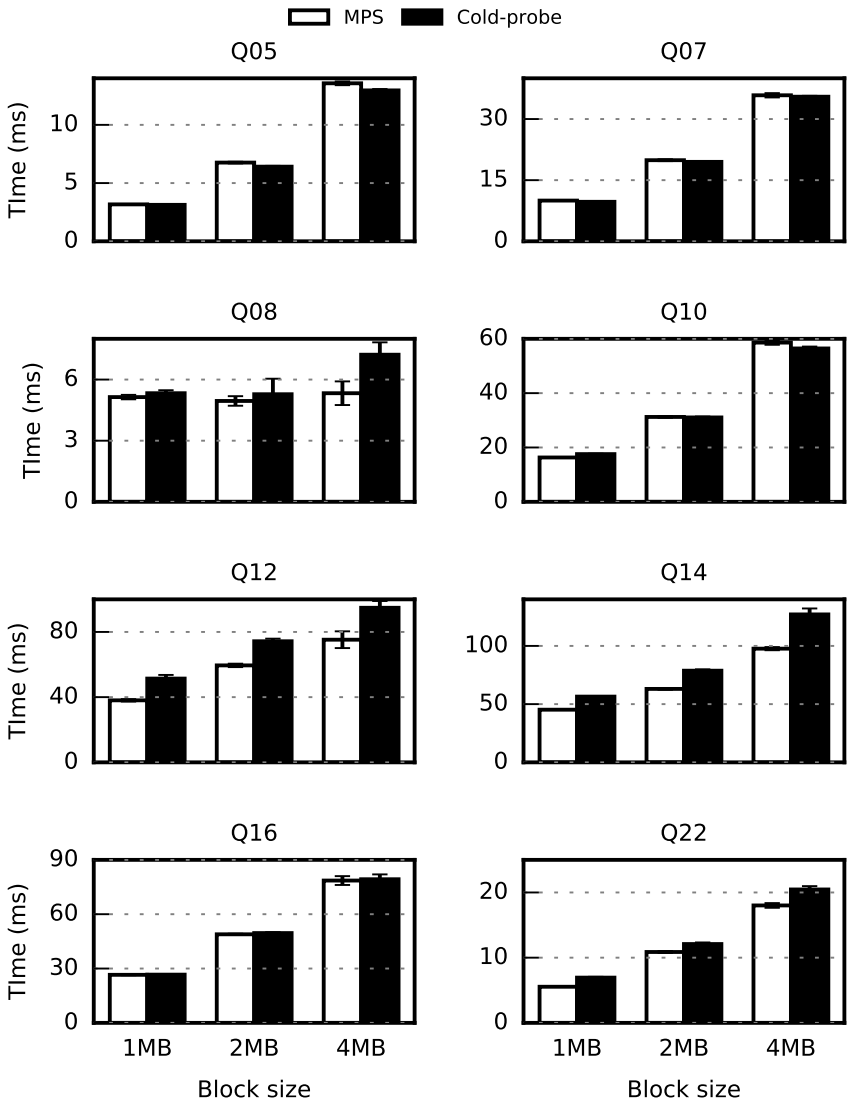



Figure 4.11: **Probe performance in cold probe sequence vs MPS algorithm output**

performance translates in a smaller performance improvement in overall query execution time.

## 4.7 Summary of Experiments

We have described a large number of experiments in Section 4.6 on queries from the TPC-H benchmark. In this section, we summarize our findings and connect our results back to the original dimensions (c.f. Section 4.4) we used for our analysis. Recall that our focus is to understand the relative performance of pipelining strategy compared to a non-pipelining strategy.

Our high level conclusion is that in the in-memory setting, for systems using block-based architecture, the performance of pipelining and non-pipelining strategy is similar. We now discuss the impact of individual dimensions.

**Block size** We find that a higher block size bridges the gap between pipelining and non-pipelining. A larger block size results in a lower degree of parallelism for operators in a pipeline and thus also aides those operators that suffer from poor scalability. A very large block size however can cause memory fragmentation. It may also result in a very low DOP such that CPU cores are left underutilized.

**Parallelism** Parallelism can affect the performance of pipelining and non-pipelining strategy in different ways. We saw in Section 4.6.3.6, how a poorly scalable operator can benefit the performance of pipelining strategy, where as it can adversely impact the performance of pipelining strategy.

**Storage Format** The gap between pipelining and non-pipelining performance is largely unaffected by the choice of storage format for the base tables. We note that individually some queries execute faster when run on base tables stored in column store format. The benefit of using column store format over row store format is the highest for base tables

(typically leaf operators in a query plan). As the number of attributes in tables reduce from the leaf levels of the query plans to the root, the advantage of using a column store starts to diminish.

**Hardware Prefetching** Hardware prefetching improves pipelining query performance. The effect is more prominent in row store than in column store format. We saw that prefetching improves scan performance in a representative query. Prefetching had no effect on probe operations and it affected the performance of build hash operations adversely.

As L3 cache size increase becomes a less viable option due to power constraints, hardware prefetching techniques are being looked at with greater interest [87]. Combined with the software-based prefetching efforts [30, 84], hardware prefetching could provide greater benefits in the future.

# Chapter 5

# Conclusions and Future Work

In this chapter, we discuss the lessons learnt, present our conclusions, and describe the future work. Through this dissertation, we present the design and implementation of a query scheduler. The goal of this scheduler is to efficiently execute queries in an in-memory environment while effectively exploiting the features provided by the modern hardware. A key contribution of this dissertation is to demonstrate that the scheduler is capable of playing an important role in the database system architecture.

## 5.1 Lessons Learnt

We now discuss the lessons learnt from this dissertation.

### 5.1.1 Importance of Abstraction and Contracts

A good abstraction is vital for developing a good system. The work order abstraction in Quickstep proved to be a boon for the query scheduler. It allowed us to construct different layers on top of simple operator execution, for which the abstraction was created.

There was a clarity of the contract between query optimizer and query scheduler: the query optimizer creates an immutable query plan (DAG of relational operators), which is then passed on to the query scheduler. Once the query scheduler starts to operate on the query plan, there is no going back to the optimizer.

We designed the operators to be stateful and allowed the scheduler to access their state thereby getting more insights about the query execution progress. The book-keeping

related data structures are handled only by the scheduler thread, thus they need not be thread-safe, which helped simplify their design.

### 5.1.2 Importance of Query Scheduler for High Performance

Achieving high performance is a never unending pursuit for database systems. Though the scheduler has a lot of power in terms of controlling resources and co-ordinating the query execution, it still needs support from other modules in the database system for high performance. Query optimizer is a vital component of the system. Scheduling work for an optimal query plan is far more effective than that for a poor plan. Moreover, efficient implementation of relational operators is crucial to leverage the most from the hardware. Therefore, even though the scheduler is an important piece in the overall puzzle, it must work in tandem with other components in the system.

### 5.1.3 Importance of Measurement, Estimation, and Analysis

Measuring and estimating performance in systems is highly important. Back-of-the-envelope calculations is a handy technique that can be useful to anticipate the effect of a technique or a change. By doing such calculations, we may be able to save a lot of development effort.

Our work on revisiting pipelining relied heavily on measuring performance and analyzing the impact of various techniques on performance. Putting the performance improvement of pipelining into perspective was only possible due to the emphasis on measurement. We made lot of mistakes in attributing performance changes to wrong reasons and learnt some valuable lessons the hard way.

While dealing with many parameters, it is important to adhere to basic scientific principles, e.g, not changing two parameters simultaneously, while studying the effect of one.

## 5.2 Conclusions

We now discuss the conclusions from individual chapters of this dissertation.

In Chapter 2, we present the design and implementation of the Quickstep database system. Quickstep has a unique block based storage management system, that supports variety of storage formats such as row store, column store along with support for compression. The query execution in Quickstep happens with the help of a task abstraction called work orders. The work orders abstraction connects storage management in Quickstep with the execution engine. Quickstep's query scheduler, which is the focus of this dissertation controls the execution of these work orders and thus manages query execution in the system. Through our experimental evaluation, we show that the intra-operator and inter-operator parallelism through the work orders abstraction speeds up the query execution in Quickstep. The work order abstraction is instrumental for realizing intra-operator parallellism. Comparison with other systems also show that Quickstep is faster than other systems, often by orders of magnitude.

In Chapters 3 and Chapter 4, we show that the scheduler can be used to meet multiple objectives. In Chapter 3 we use the scheduler for the problem of resource governance. We specifically target the problem of allocating resources such as CPU to concurrent queries according to well defined policies. A key challenge in this problem is the fluctuation in the resource requirements of a query. We target real time queries setting, in which queries could arrive at any time, thus it further complicates the problem. Our solution uses a probabilistic framework for taking the scheduling decisions and thereby allocating resources to queries. The probabilistic framework is backed by a learning agent, that monitors the resource consumption in the system and using learning techniques, predicts the resource requirements of queries in the near future. We form a close loop between scheduling decisions and the learning, where the learning agent monitors the system, predicts the future resource requirements and tweaks the probabilities. The adjusted probabilities result in

change in the resource allocation, leading the system closer to meeting the high level policy goals. Our experimental evaluation shows that Quickstep scheduler can meet the goals for a variety of policies including fair and priority-based.

Chapter 4 uses the scheduler for the performance objective. Here we analyze the role of pipelining in the in-memory settings. We cast pipelining as a schedule in which consumer tasks (or work orders) are interleaved with the producer tasks. We compare pipelining with a non-pipelining strategy in which the said interleaving is not present. The two strategies differ primarily in the eagerness of data processing by the consumer operator. We underscore the impact of various parameters such as number of threads, the size of blocks, and hardware prefetching on the relative performance of the two pipelining strategies. Our analysis (based on evaluation on TPC-H queries as well as an analytical model and microbenchmarking) shows that the relative gap between the two strategies is not as high as in the disk based settings. We highlight that the impact of pipelining on query processing should be considered holistically with all the parameters listed earlier.

### 5.2.1 Future Work

Our scheduling work leads to many interesting future work directions. There is an ongoing effort to make Quickstep work efficiently on a distributed setup. The distributed setting offers some challenges which are not as relevant in the in-memory single node setting. Locality of a work order execution is quite important in the distributed setting. The penalty of a non-local execution involves high data movement cost over network, which can degrade query performance.

Load balancing is another important aspect that a distributed query scheduler has to worry about. Overloading a node with work can throttle the query processing and may cause bottlenecks. A subtle aspect in distributed query scheduling is that the control traffic (e.g. dispatching a work order, reeving work order completion feedback) may be expensive. Therefore the scheduler may have to batch or aggregate control messages.

In the single node setting, the scheduler can play a big role in query processing on heterogeneous hardware. An example of such system can be a server configured with an FPGA or a GPU. If the scheduler can understand the strengths and weaknesses of each heterogeneous hardware component, it can route work orders to different components appropriately. A key challenge in this problem is to understand the memory access costs for different hardware components and thus estimating the cost of executing operations.

# Bibliography


[1] Apache (incubating) quickstep data processing platform. http://www.quickstep.io.

[2] Greenplum workload management. http://gpdb.docs.pivotal.io/4370/admin_guide/workload_mgmt.html.

[3] HP Global Workload Manager. https://h20565.www2.hpe.com/hpsc/doc/public/display?sp4ts.oid=3725908&docId=emr_na-c04159995.

[4] Hyrise. https://hpi.de/plattner/projects/hyrise.html. Accessed on 07/20/2018.

[5] IBM DB2 Workload Manager. https://www-01.ibm.com/support/knowledgecenter/SSEPGG_10.1.0/com.ibm.db2.luw.admin.wlm.doc/com.ibm.db2.luw.admin.wlm.doc-gentopic1.html.

[6] MS SQL SERVER resource governor. https://msdn.microsoft.com/en-us/library/bb933866.aspx.

[7] Oracle resource manager. https://docs.oracle.com/database/121/ADMIN/dbrm.htm#ADMIN027.

[8] Peloton - the self driving database management system. htts://pelotondb.io. (Accessed on 07/05/2018.

[9] Teradata workload management. http://www.teradata.com/Teradata_Workload_Management.

[10] Tpc-h - homepage. http://www.tpc.org/tpch/.

[11] D. J. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682, 2006.

[12] D. J. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: how different are they really? In *SIGMOD Conference*, pages 967–980. ACM, 2008.

[13] L. Abraham, J. Allen, O. Barykin, V. R. Borkar, B. Chopra, C. Gerea, D. Merl, J. Metzler, D. Reiss, S. Subramanian, J. L. Wiener, and O. Zed. Scuba: Diving into data at facebook. *PVLDB*, 6(11):1057–1067, 2013.

[14] A. Ailamaki, D. J. DeWitt, and M. D. Hill. Data page layouts for relational databases on deep memory hierarchies. *VLDB J.*, 11(3):198–215, 2002.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.

[16] R. Barber, G. M. Lohman, I. Pandis, V. Raman, R. Sidle, G. K. Attaluri, N. Chainani, S. Lightstone, and D. Sharpe. Memory-efficient hash joins. *PVLDB*, 8(4):353–364, 2014.

[17] C. Beeri and R. Ramakrishnan. On the power of magic. In *PODS*, pages 269–284, 1987.

[18] P. A. Bernstein and D.-M. W. Chiu. Using semi-joins to solve relational queries. *J. ACM*, 28(1):25–40, Jan. 1981.

[19] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *SIGMOD*, pages 37–48, 2011.

[20] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *CACM*, 13:422–426, 1970.

[21] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, 2008.

[22] P. A. Boncz, T. Neumann, and O. Erling. TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark. In *5th TPC Technology Conference, TPCTC*, pages 61–76, 2013.

[23] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. In *CIDR*, pages 225–237. www.cidrdb.org, 2005.

[24] P. Bonnet, S. Manegold, M. Bjørling, W. Cao, J. Gonzalez, J. A. Granados, N. Hall, S. Idreos, M. Ivanova, R. Johnson, D. Koop, T. Kraska, R. Müller, D. Olteanu, P. Papotti, C. Reilly, D. Tsirogiannis, C. Yu, J. Freire, and D. E. Shasha. Repeatability and workability evaluation of SIGMOD 2011. *SIGMOD Record*, 40(2):45–48, 2011.

[25] L. Bouganim, O. Kapitskaia, and P. Valduriez. Memory-adaptive scheduling for large query execution. In *Proceedings of the Seventh International Conference on Information and Knowledge Management*, CIKM '98, pages 105–115, New York, NY, USA, 1998. ACM.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. OSDI, pages 205–218, 2006.

[27] C. Chasseur and J. M. Patel. Design and evaluation of storage organizations for read-optimized main memory databases. *PVLDB*, 6(13):1474–1485, 2013.

[28] S. Chaudhuri, V. R. Narasayya, and R. Ramamurthy. Estimating progress of long running SQL queries. In *SIGMOD*, 2004.

[29] M. Chen, M. Lo, P. S. Yu, and H. C. Young. Applying segmented right-deep trees to pipelining multiple hash joins. *IEEE Trans. Knowl. Data Eng.*, 7(4):656–668, 1995.

[30] S. Chen, A. Ailamaki, P. B. Gibbons, and T. C. Mowry. Improving hash join performance through prefetching. In *ICDE*, pages 116–127. IEEE Computer Society, 2004.

[31] Y. Chi, H. Hacigümüs, W. Hsiung, and J. F. Naughton. Distribution-based query scheduling. *PVLDB*, 2013.

[32] Y. Chi, H. J. Moon, and H. Hacigümüs. icbs: Incremental costbased scheduling under piecewise linear slas. *PVLDB*, 2011.

[33] Citus Data. `https://www.citusdata.com`, 2016.

[34] D. L. Davison and G. Graefe. Memory-contention responsive hash joins. In *VLDB*, 1994.

[35] D. L. Davison and G. Graefe. Dynamic resource brokering for multi-user query execution. In *SIGMOD*, 1995.

[36] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. OSDI, pages 10–10, 2004.

[37] H. Deshmukh, H. Memisoglu, and J. M. Patel. Adaptive concurrent query execution framework for an analytical in-memory database system. In *BigData Congress*, pages 23–30. IEEE Computer Society, 2017.

[38] H. Deshmukh, H. Memisoglu, and J. M. Patel. Adaptive concurrent query execution framework for an analytical in-memory database system - supplement. `http://www.cs.wisc.edu/~harshad/includes/scheduler-supplement.pdf`, May 2017.

[39] A. Deshpande and L. Hellerstein. Flow algorithms for parallel query optimization. In *ICDE*, pages 754–763. IEEE Computer Society, 2008.

[40] D. J. DeWitt, S. Ghandeharizadeh, and D. A. Schneider. A performance analysis of the gamma database machine. In *SIGMOD Conference*, pages 350–360. ACM Press, 1988.

[41] D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Commun. ACM*, 35(6):85–98, 1992.

[42] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum. Optimizing space amplification in rocksdb. In *CIDR*. www.cidrdb.org, 2017.

[43] J. Duggan, U. Çetintemel, O. Papaemmanouil, and E. Upfal. Performance prediction for concurrent database workloads. In *SIGMOD*, 2011.

[44] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[45] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees. The SAP HANA database – an architecture overview. *IEEE Data Eng. Bull.*, 35(1):28–33, 2012.

[46] Z. Feng, E. Lo, B. Kao, and W. Xu. Byteslice: Pushing the envelop of main memory data processing with a new storage layout. In *SIGMOD*, pages 31–46, 2015.

[47] R. H. Gerber. *Dataflow Query Processing using Multiprocessor Hash-partitioned Algorithms*. PhD thesis, Madison, WI, USA, 1986. TR672.

[48] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, pages 102–111, 1990.

[49] G. Graefe, A. C. König, H. A. Kuno, V. Markl, and K.-U. Sattler. 10381 Summary and Abstracts Collection – Robust Query Processing. Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2011.

[50] Greenplum database. `http://greenplum.org`, 2016.

[51] M. Grund, J. Krüger, H. Plattner, A. Zeier, P. Cudré-Mauroux, and S. Madden. HYRISE - A main memory hybrid storage engine. *PVLDB*, 4(2):105–116, 2010.

[52] C. Gupta, A. Mehta, S. Wang, and U. Dayal. Fair, effective, efficient and differentiated scheduling in an enterprise data warehouse. In *EDBT*, 2009.

[53] R. A. Hankins and J. M. Patel. Data morphing: An adaptive, cache-conscious storage technique. In *VLDB*, pages 417–428. Morgan Kaufmann, 2003.

[54] J. R. Haritsa, M. Livny, and M. J. Carey. Earliest deadline scheduling for real-time database systems. In *RTSS*, pages 232–242. IEEE Computer Society, 1991.

[55] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *SIGMOD Conference*, pages 383–394. ACM, 2005.

[56] Harshad Deshmukh. Storage Formats in Quickstep. http://quickstep.incubator.apache.org/guides/2017/03/30/storage-formats-quickstep.html, 2017.

[57] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*, 1(2):141–259, 2007.

[58] Y. Huai, A. Chauhan, A. Gates, G. Hagleitner, E. N. Hanson, O. O'Malley, J. Pandey, Y. Yuan, R. Lee, and X. Zhang. Major technical advancements in apache hive. In *SIGMOD Conference*, pages 1235–1246. ACM, 2014.

[59] IBM Corp. Database design with denormalization. http://ibm.co/2eKWmW1.

[60] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. MonetDB: Two decades of research in column-oriented database architectures. *IEEE Data Eng. Bull.*, 35(1):40–45, 2012.

[61] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE*, pages 774–783, 2008.

[62] R. Johnson, V. Raman, R. Sidle, and G. Swart. Row-wise parallel predicate evaluation. *PVLDB*, 1(1):622–634, 2008.

[63] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[64] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. 2nd edition, 2002.

[65] Y. Klonatos, C. Koch, T. Rompf, and H. Chafi. Building efficient query engines in a high-level language. *PVLDB*, 7(10):853–864, 2014.

[66] M. Kornacker, A. Behm, V. Bittorf, T. Bobrovytsky, C. Ching, A. Choi, J. Erickson, M. Grund, D. Hecht, M. Jacobs, I. Joshi, L. Kuff, D. Kumar, A. Leblang, N. Li, I. Pandis, H. Robinson, D. Rorke, S. Rus, J. Russell, D. Tsirogiannis, S. Wanderman-Milne, and M. Yoder. Impala: A modern, open-source SQL engine for hadoop. In *CIDR*, 2015.

[67] S. Krompass, U. Dayal, H. A. Kuno, and A. Kemper. Dynamic workload management for very large data warehouses: Juggling feathers and bowling balls. In *VLDB*, 2007.

[68] S. Krompass, D. Gmach, A. Scholz, S. Seltzsam, and A. Kemper. Quality of service enabled database applications. In *ICSOC*, 2006.

[69] A. Kumar, M. Boehm, and J. Yang. Data management in machine learning: Challenges, techniques, and systems. In *SIGMOD Conference*, pages 1717–1722. ACM, 2017.

[70] Y. Kwok and I. Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Comput. Surv.*, 31(4):406–471, 1999.

[71] B. W. Lampson and H. E. Sturgis. Reflections on an operating system design. *CACM*, 1976.

[72] P. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik. Enhancements to SQL server column stores. In *SIGMOD*, pages 1159–1168, 2013.

[73] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD*, pages 743–754, 2014.

[74] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *SIGMOD*, 2014.

[75] J. Li, R. V. Nehme, and J. F. Naughton. GSLPI: A cost-based query progress indicator. In *ICDE*, 2012.

[76] Y. Li and J. M. Patel. Bitweaving: fast scans for main memory data processing. In *SIGMOD*, 2013.

[77] Y. Li and J. M. Patel. WideTable: An accelerator for analytical data processing. *PVLDB*, 7(10):907–918, 2014.

[78] R. Linn and W. Zhang. Hybrid flow shop scheduling: a survey. *Computers & industrial engineering*, 37(1-2):57–61, 1999.

[79] B. Liu and E. A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, pages 829–840. ACM, 2005.

[80] F. Liu and S. Blanas. Forecasting the cost of processing multi-join queries via hashing for main-memory databases. In *SoCC*, pages 153–166. ACM, 2015.

[81] S. Manegold, I. Manolescu, L. Afanasiev, J. Feng, G. Gou, M. Hadjieleftheriou, S. Harizopoulos, P. Kalnis, K. Karanasos, D. Laurent, M. Lupu, N. Onose, C. Ré, V. Sans, P. Senellart, T. Wu, and D. E. Shasha. Repeatability & workability evaluation of SIGMOD 2009. *SIGMOD Record*, 38(3):40–43, 2009.

[82] I. Manolescu, L. Afanasiev, A. Arion, J. Dittrich, S. Manegold, N. Polyzotis, K. Schnaitter, P. Senellart, S. Zoupanos, and D. E. Shasha. The repeatability experiment of SIGMOD 2008. *SIGMOD Record*, 37(1):39–45, 2008.

[83] M. Mehta and D. J. DeWitt. Dynamic memory allocation for multiple-query workloads. In *VLDB*, 1993.

[84] P. Menon, A. Pavlo, and T. C. Mowry. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *PVLDB*, 11(1):1–13, 2017.

[85] Microsoft. Implied predicates and query hints. https://blogs.msdn.microsoft.com/craigfr/2009/04/28/implied-predicates-and-query-hints/, 2009.

[86] Microsoft Corp. Optimizing the Database Design by Denormalizing. https://msdn.microsoft.com/en-us/library/cc505841.aspx.

[87] S. Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2):35:1–35:35, 2016.

[88] B. Nag and D. J. DeWitt. Memory allocation strategies for complex decision support queries. In *CIKM*, pages 116–123. ACM, 1998.

[89] F. Nagel, G. M. Bierman, and S. D. Viglas. Code generation for efficient query processing in managed runtimes. *PVLDB*, 7(12):1095–1106, 2014.

[90] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri. SQLVM: performance isolation in multi-tenant relational database-as-a-service. In *CIDR*, 2013.

[91] V. R. Narasayya, I. Menache, M. Singh, F. Li, M. Syamala, and S. Chaudhuri. Sharing buffer pool memory in multi-tenant relational database-as-a-service. *PVLDB*, 2015.

[92] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[93] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *PVLDB*, 4(9):539–550, 2011.

[94] E. J. O'Neil, P. E. O'Neil, and G. Weikum. An optimality proof of the lru-*K* page replacement algorithm. *J. ACM*, 46(1):92–112, 1999.

[95] P. O'Neil, E. O'Neil, and X. Chen. The star schema benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.pdf, Jan 2007.

[96] Oracle. Push-down part 2. https://blogs.oracle.com/in-memory/push-down:-part-2, 2015.

[97] Oracle. White paper. http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.pdf, 2017.

[98] Pamela Vagata and Kevin Wilfong. Scaling the Facebook data warehouse to 300 PB. https://code.facebook.com/posts/229861827208629, 2014.

[99] J. M. Patel, H. Deshmukh, J. Zhu, H. Memisoglu, N. Potti, S. Saurabh, M. Spehlmann, and Z. Zhang. Quickstep: A data platform based on the scaling-in approach. Technical Report 1847, University of Wisconsin-Madison, 2017.

[100] J. M. Patel, H. Deshmukh, J. Zhu, N. Potti, Z. Zhang, M. Spehlmann, H. Memisoglu, and S. Saurabh. Quickstep: A data platform based on the scaling-up approach. *PVLDB*, 11(6):663–676, 2018.

[101] M. L. Pinedo. *Scheduling: theory, algorithms, and systems*. Springer, 2016.

[102] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - A vector algebra for portable database performance on modern hardware. *PVLDB*, 9(14):1707–1718, 2016.

[103] PostgreSQL. `http://www.postgresql.org`, 2016.

[104] PostgreSQL. Parallel Query. `https://wiki.postgresql.org/wiki/Parallel_Query`.

[105] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.

[106] T. Rabl, M. Poess, H. Jacobsen, P. E. O'Neil, and E. J. O'Neil. Variations of the star schema benchmark to test the effects of data skew on query performance. In *ICPE*, pages 361–372. ACM, 2013.

[107] B. Raducanu, P. A. Boncz, and M. Zukowski. Micro adaptivity in vectorwise. In *SIGMOD*, pages 1231–1242, 2013.

[108] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang. DB2 with BLU acceleration: So much more than just a column store. *PVLDB*, 6(11):1080–1091, 2013.

[109] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *ICDE*, pages 60–69, 2008.

[110] J. Rao and K. A. Ross. Making $B^+$-trees cache conscious in main memory. In *SIGMOD*, 2000.

[111] Amazon Redshift. `https://aws.amazon.com/redshift/`, 2016.

[112] Reynold Xin. Technical Preview of Apache Spark 2.0. `https://databricks.com/blog/2016/05/11`.

[113] R. Ricci, E. Eide, and The CloudLab Team. Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications. *USENIX ;login:*, 39(6), Dec. 2014.

[114] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In *VLDB*, pages 469–480. Morgan Kaufmann, 1990.

[115] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round-robin. *IEEE/ACM Trans. Netw.*, 1996.

[116] L. Shrinivas, S. Bodagala, R. Varadarajan, A. Cary, V. Bharathan, and C. Bear. Materialization strategies in the vertica analytic database: Lessons learned. In *ICDE*, pages 1196–1207. IEEE, 2013.

[117] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[118] Standard Performance Evaluation Corporation. INT2006 (Integer Component of SPEC CPU2006). `https://www.spec.org/cpu2006/CINT2006`, 2016.

[119] Statistic Brain Research Institute. Google Annual Search Statistics. `http://www.statisticbrain.com/google-searches`, 2016.

[120] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[121] Sybase Inc. Denormalizing Tables and Columns. `http://infocenter.sybase.com`.

[122] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, N. Zhang, S. Antony, H. Liu, and R. Murthy. Hive-a petabyte scale data warehouse using hadoop. In *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*, pages 996–1005. IEEE, 2010.

[123] J. Ullman. Np-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384 – 393, 1975.

[124] V. Viswanathan. Disclosure of h/w prefetcher control on some intel processors. `https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors`, September 2014. (Accessed on 06/12/2018).

[125] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *OSDI*, 1994.

[126] L. Wang, M. Zhou, Z. Zhang, Y. Yang, A. Zhou, and D. Bitton. Elastic pipelining in an in-memory database cluster. In *SIGMOD*, 2016.

[127] T. Willhalm, I. Oukid, I. Müller, and F. Faerber. Vectorizing database column scans with complex predicates. In *ADMS*, pages 1–12, 2013.

[128] T. Willhalm, N. Popovici, Y. Boshmaf, H. Plattner, A. Zeier, and J. Schaffner. SIMD-scan: Ultra fast in-memory table scan using on-chip vector processing units. *PVLDB*, 2(1):385–394, 2009.

[129] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

[130] W. Wu, Y. Chi, H. Hacigümüs, and J. F. Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *PVLDB*, 2013.

[131] W. Wu, X. Wu, H. Hacigümüs, and J. F. Naughton. Uncertainty aware query execution time prediction. *PVLDB*, 2014.

[132] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD*, pages 13–24, 2013.

[133] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *USENIX*, pages 15–28, 2012.

[134] Q. Zeng, J. M. Patel, and D. Page. Quickfoil: Scalable inductive logic programming. *PVLDB*, 8(3):197–208, 2014.

[135] J. Zhou and K. A. Ross. Implementing database operations using SIMD instructions. In *SIGMOD*, pages 145–156, 2002.

[136] J. Zhu, N. Potti, S. Saurabh, and J. M. Patel. Looking ahead makes query plans robust. *PVLDB*, 10(8):889–900, 2017.

[137] M. Zukowski and P. A. Boncz. Vectorwise: Beyond column stores. *IEEE Data Eng. Bull.*, 35(1):21–27, 2012.

[138] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *VLDB*, pages 723–734. ACM, 2007.

# APPENDIX

We describe the various implementation details, design choices and algorithms used in the Appendix.

## A.1 DAG Traversal Algorithm

The Query Manager is presented with a DAG for each query, where each node in the DAG represents a relational operator primitive. The edges in the DAG are annotated with whether the *consumer* operator is blocked on the output produced by the *producer* operator, or whether data pipelining is allowed between two adjacent operators.

Consider a sample join query and its DAG showed in Figure A.1. The solid arrows in the DAG correspond to "blocking" dependencies, and the dashed arrows indicate pipeline-able/non-blocking dependencies. To execute this query we need to select tuples from the `ddate` table, stream them to a hash table, which can then be probed by tuples that are created by the selection operator on the `lineorder` table. The output of the probe hash operation can be sent to the print operator, which displays the result. Note that the "drop hash" operator is used to drop the hash table, but only after the "probe hash" operation is complete. Similarly, the other drop operators indicate when intermediate data can be deleted.

The Query Manager uses a DAG Traversal algorithm (cf. Algorithm A.1) to process the DAG, which essentially is an iterative graph traversal method. The algorithm simply finds nodes in the DAG that have all their dependencies met, and marks such nodes as "active" (line 15). Work orders are requested and scheduled for all active nodes (line 18), and the completion of work orders is monitored. Operators are stateful and they produce work

**Algorithm A.1** DAG Traversal

```
1: G = {V, E}
2: activeEdges = {e ∈ E | e.isNotPipelineBreaking()}
3: inactiveEdges = {e ∈ E | e.isPipelineBreaking()}
4: completedNodes = {}
5: for v ∈ V do:
6:     if v.allIncomingEdgesActive() then
7:         v.active = True
8:     else
9:         v.active = False
10:    end if
11: end for
12: while completedNodes.size() < V.size() do
13:    for v ∈ V – completedNodes do
14:        if v.allIncomingEdgesActive() then
15:            v.active = True
16:        end if
17:        if v.active then
18:            v.getAllNewWorkOrders()
19:            if v.finishedGeneratingWorkOrders() then
20:                completedNodes.add(v)
21:                for outEdge ∈ v.outgoingEdges() do
22:                    activeEdges.add(outEdge)
23:                end for
24:            end if
25:        end if
26:    end for
27: end while
```

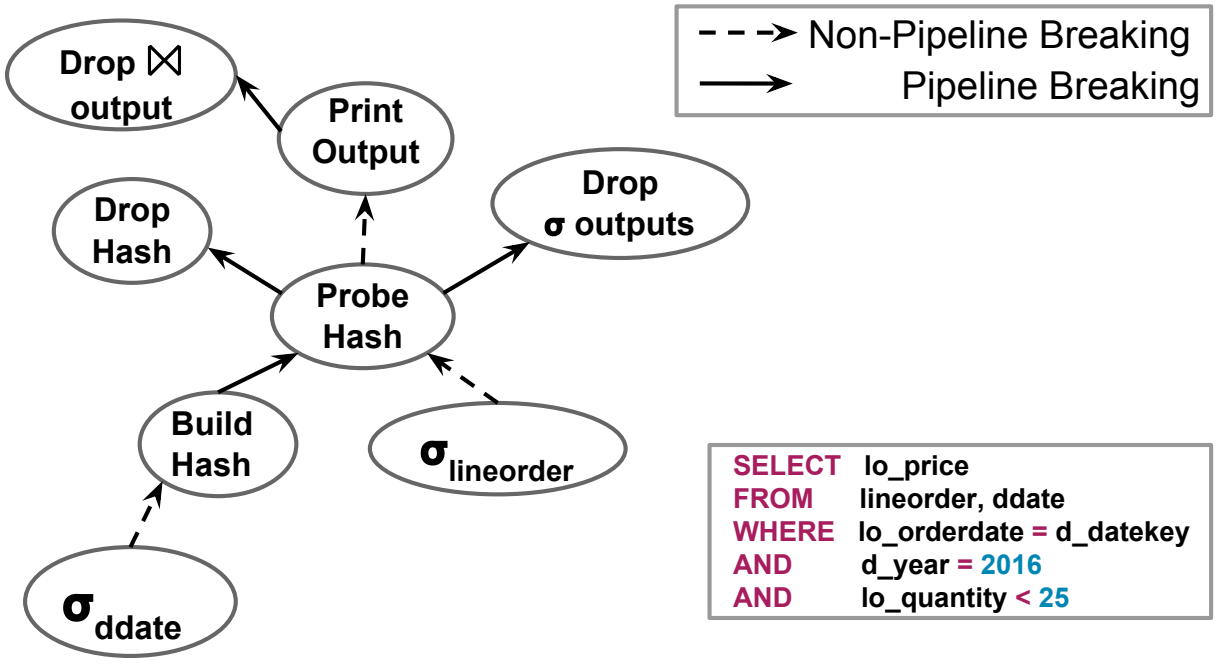



Figure A.1: **A join query and its DAG**

orders when they have the necessary data. The work order generation stops (line 19) when the operators no longer have any input to produce additional work orders. When no more work orders can be generated for a node, that node is marked as "completed" (line 20). When a node is marked as completed, all outgoing blocking edges (the solid lines in Figure A.1) are "activated" (line 22). Pipelining is achieved as all non-blocking edges (dotted lines in Figure A.1) are marked as active upfront (line 2). The query is deemed as completed when all nodes are marked as "completed."

## A.2 Resource Map Discussion

An example Resource Map of an incoming query to the system is shown below:

```
CPU:    {min: 1 Core, max: 20 Cores}
Memory: {min: 20 MB,  max: 100 MB}
```

This Resource Map states that the query can use 1 to 20 cores (i.e. specifies the range of intra-operator parallelism) and is estimated to require a minimum of 20 MB of memory to run, and an estimated 100 MB of memory in the worst case.

In Quickstep, the query optimizer provides the estimated memory requirements for a given query. Other methods can also be used, such as inferring the estimated resources from the previous runs of the query or other statistical analyses. The scheduler is agnostic to how these estimates are calculated.

## A.3 Pipelining in Quickstep

During a work order (presumably belonging to a producer relational operator in a pipeline) execution, output data may be created (in blocks in the buffer pool). When an output data block is filled, the worker thread sends a "block filled" message to the corresponding query's manager via the following channel: Worker $\rightarrow$ Foreman $\rightarrow$ Policy Enforcer $\rightarrow$ Query Manager. The Query Manager may then create a new work order (for a

consumer relational operator in the pipeline) based on this information; e.g. if this block should be pipelined to another operator in the query plan.

Note that pipelining in Quickstep works on a block-basis, instead of the traditional tuple-basis.

## A.4 Motivation for the Learning Agent Module

One might question the need of the Learning Agent and instead consider assigning a fixed probability value to each query (say $1/N$, with $N$ queries in the fair policy). In the following section, we address this issue. A motivational example for the learning agent is described in Appendix A.4.

We perform an experiment, where the goal is to analyze the patterns in work order execution times of two queries. The dataset used for the experiment comes from the Star Schema Benchmark (SSB) [95] at a scale factor of 100. We pick two SSB queries $Q1.1$ and $Q4.1$, and execute them on a machine with 40 CPU cores. $Q1.1$ has a single join operation and $Q4.1$ has four join operations. Figure A.2 shows the observed average time per work order for both queries. We now describe the trends in time per work order for the queries.

We can observe $Q1.1$'s execution pattern denoted by the dashed line in Figure A.2. The time per work order remains fairly stable (barring some intermittent fluctuations) from the beginning until 1.8 s. This phase corresponds to the selection operation in $Q1.1$ which evaluates predicates on the *lineorder* (fact) table. A small bump in time per work order can be observed at the 1.8 s mark, when the probe phase of $Q1.1$ begins and continues until 2 s. Towards the end of the execution of $Q1.1$, (2.2 s) there is a spike in time per work order when the query enters the aggregation phase. The output of the hash join is fed to the aggregation operation. The results of aggregation are stored in per-thread private hash tables, which are later merged to produce the final output.

Now we analyze the execution pattern for $Q4.1$ which has 4 join operations. This query is more complex than $Q1.1$.Therefore, the execution pattern of $Q4.1$ exhibits more phases, with different times per work order as compared to $Q1.1$. Various small phases before the

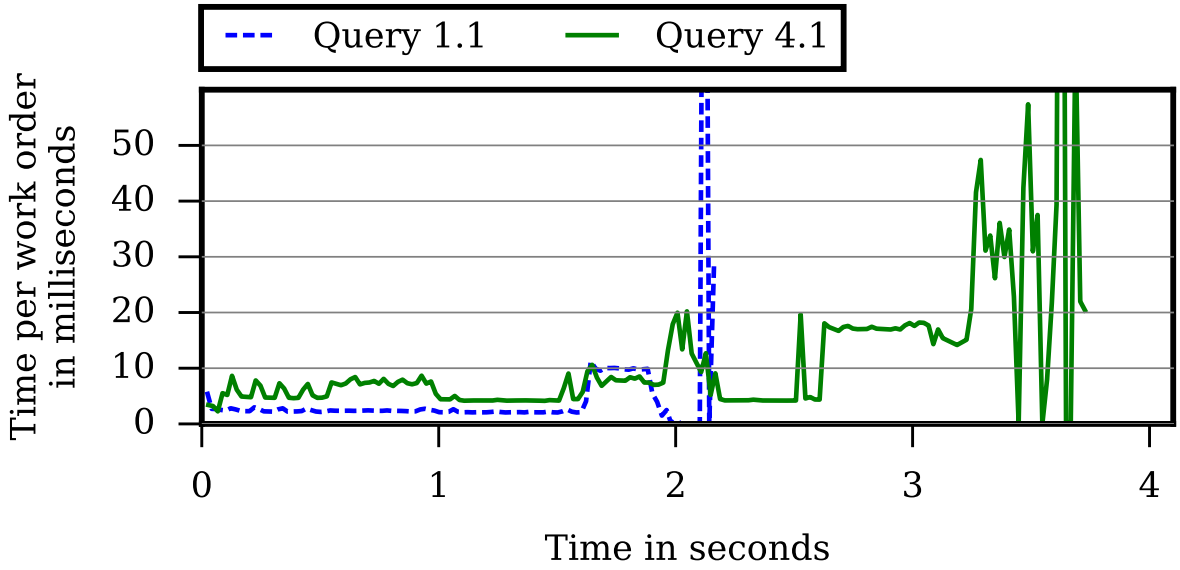



Figure A.2: **Time per work order for SSB Q1.1 and Q4.1**

0.5 s mark correspond to the selection predicates that are applied to the dimension tables. (Note that in $Q4.1$ there is no selection filter on the *lineorder* table). The selections on dimension tables get executed quickly. The longer phases denote the different probe hash table operations in the query. Towards the end, similar to $Q1.1$, there is a spike in the execution time per work order which correspond to the aggregation phase.

It is clear that the work order execution times for both queries are different, and the difference between them changes over time. If the scheduler assigns the same probability to both queries (i.e. 0.5), it is equally likely to schedule a work order from either of them. As a result, the queries will have different CPU utilization times in a given epoch, thus resulting in an unfair CPU allocation. In order to be consistently fair in allocating CPU resources to the queries, we should continuously observe the work order execution times of queries and adjust the CPU allocation accordingly.

## A.5 Usage of Linear Regression in Learning Agent

The Learning Agent uses linear regression for predicting the execution time of the future work orders. To lower the CPU and memory overhead of the model, we limit the amount of execution statistics stored in the Learning Agent. We discard records beyond a certain time window. When all the work orders of an operator finish execution, we remove its records completely. In a query, if multiple relational operators are active, linear regression combines the statistics of all active operators and predicts a single value for the next work order execution time.

## A.6 Resource Choices for Policy Implementations and Load Controller Implementations

In the current implementation of Quickstep, we have focused on two key types of resource in the in-memory deployment scenarios – CPU and memory. Both these resources have different resource characteristics, which we outline below.

First, consider the CPU resource. On modern commodity servers there are often tens of CPU cores per socket, and the aggregate number of cycles available per unit time (e.g. a millisecond) across all the cores is very large. Further, an implication of Quickstep's fine-grained task allocation and execution paradigm is that the CPU resource can be easily shared at a fine time-granularity. Several work orders, each from different query can be executed concurrently on different CPU cores, and each query may execute thousands or millions or even more number of work orders. Thus, in practical terms, the CPU resource can be viewed as a nearly infinitely divisible resource across concurrent queries. In addition, overall system utilization is often measured in terms of the CPU utilization. Combining all these factors, specifying a policy in terms of the CPU utilization is natural, and intuitive for a user to understand the policy. For example, saying that a fair policy equally distributes the CPU resource across all (admitted) concurrent queries is simple to understand and reason.

Memory, on the other hand, is a resource that is allocated by queries in larger granular chunks. Active queries can have varying memory footprints (and the footprint for a query can change over the course of its execution). Thus, memory as a resource is more naturally viewed as a "gating" resource. Therefore, it is natural to use it in the load controller to determine if a query can be admitted based on its requested memory size. Actual memory consumption for queries can also be easily monitored, and when needed queries can be suspended if memory resource needs to be freed up (for some other query, perhaps with a higher priority).

## A.7 Applicability of SSB for our evaluation

The SSB is based on the TPC-H benchmark, and is designed to measure the query performance when the data warehouse uses the popular Kimball [64] approach. At a scale factor of X, the benchmark corresponds to about X GB of data in the corresponding TPC-H warehouse. The SSB benchmark has 13 queries, divided in four categories. Each query is identified as **qX.Y**, where $X$ is the class and $Y$ is the query number within the class. There are four query classes, i.e. $1 \leq X \leq 4$. The first and second classes have three queries each, the third class has four queries, and the fourth class has three queries. The queries in each category are similar with respect to aspects such as the number of joins in the query, the relations being joined, the filter and aggregation attributes. The grouping of queries in various classes makes this benchmark suitable for our experiments, as it provides a way of assigning priorities to the queries based on their class.

## A.8 Policy Enforcer

The Policy Enforcer applies a high level policy for resource allocation among concurrent queries. It uses a probabilistic-framework to select work orders from a pool of work orders belonging to different concurrent queries for scheduling. The Policy Enforcer assigns a probability to each active query, which indicates the likelihood of a work order from that

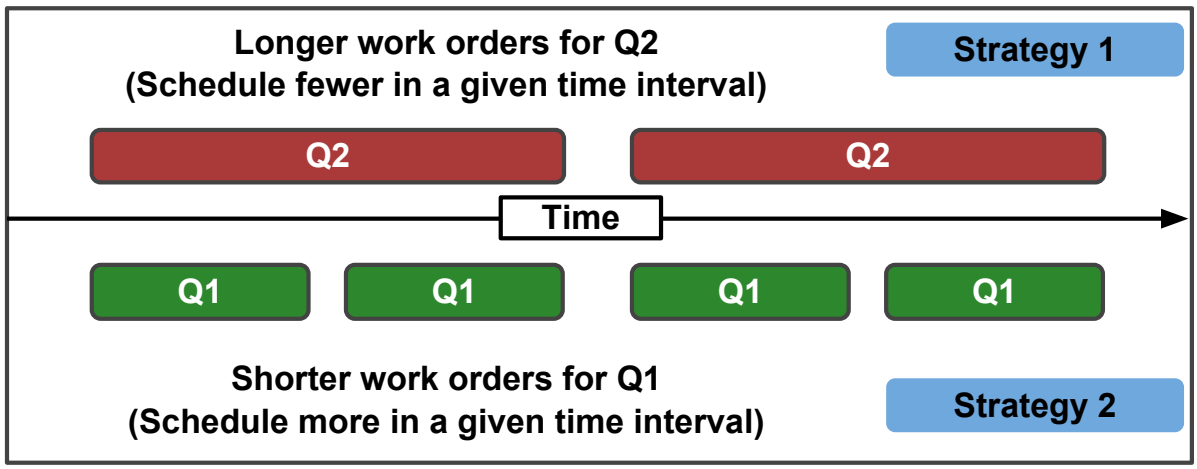



Figure A.3: **Scheduling queries having different work order execution times for the fair policy. The solid boxes with $Q_i$ depicts the lifetime of a work order from the Query $Q_i$**

query getting scheduled for execution in the near future. The probability-based work order selection strategy brings powerful control to the scheduler through a single parameter – i.e. by controlling the probability setting, the scheduler can control the resource sharing among concurrent queries.

The challenge in designing the policy enforcer lies in transforming the policy specifications to a set of probabilities. A critical piece that we use in such transformations is the prediction of work order execution times for the concurrent queries, which is done by the Learning Agent. In the remainder of this section, we provide an intuition for deriving probability values from the work order execution times.

We now motivate the probabilistic approach used by the Policy Enforcer with an example. Consider a single CPU core and two concurrent queries $q_1$ and $q_2$. (The idea can be extended to multi-cores and more than two queries.) Initially, we assume perfect knowledge of the execution times of work orders of the queries. Later, we will relax this assumption.

Let us assume that as per the policy specifications, in a given time interval, the CPU resources should be shared equally. Suppose that work orders for $q_1$ take less time to

execute than work orders for $q_2$, as shown in Figure A.3. As the Policy Enforcer aims to allocate equal share of the CPU to $q_1$ and $q_2$, a simple strategy can be to schedule proportionally more work orders of $q_1$ than those of $q_2$, in a given time window. The number of scheduled work orders is inversely related to the work order execution time. This proportion can be determined by the probabilities $pb_1$ and $pb_2$ for queries $q_1$ and $q_2$, respectively. The probability $pb_i$ is the likelihood of the scheduler scheduling next work order from query $i$. The probability is assigned by the Policy Enforcer to each active query in the system. Note that, $pb_1 > pb_2$ and $pb_1 + pb_2 = 1$.

Notice that the Policy Enforcer is not concerned with the complexities of the operators in the query DAGs. It simply maintains the probability associated with each active query which is determined by the query's predicted work order execution times.

The Policy Enforcer can also function with workloads that consist of queries categorized in multiple classes, where each class has a different level of "importance" or "priority". The policy specifies that the resource allocation to a query class must simply be in accordance to its importance, i.e. queries in a more important class should collectively get a higher share of the resources, and vice versa. In such scenarios, the Policy Enforcer splits its work order selection strategy in two steps - selection of a query class and subsequent selection of a query within the chosen query class. Intuitively, the Policy Enforcer should assign higher probability to the more important class and lower probability to the less important class.

Once a query class is chosen, the Policy Enforcer must pick a query from the chosen class. Each query class can specify an optional intra-class resource allocation sub-policy. By default, all queries within a class are treated equally. Thus, the probability-based paradigm can be used to control both inter and intra-class resource allocations.

There could be many reasons for categorizing queries in classes, including the need to associate some form of urgency (e.g. interactive vs batch queries), or marking the importance of the query source (e.g. the position of the query submitter in an organizational hierarchy). In addition, the resource allocations across different classes can also be chosen based on various scales, such as linear or exponential scale allocations based on the class

number. An attractive feature of the Policy Enforcer is that it can be easily configured for use in a variety of ways. Under the covers, the Policy Enforcer simply maps each class to a collective class probability value, and then maps each query in each class to another probability. Once these probabilities are calculated, the remaining mechanisms simply use them to appropriately allocate resources to achieve the desired policy goal.