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: 07/09/2018

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

ACKNOWLEDGMENTS

Write acknowledgements here

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

Introduction

Scheduling is a fundamental problem in computer systems, and physical systems. Systems use resources and often times these reources are finite and scarce. Scheduling facilitates the systems to use limited resources and share them with the users of the system. Two examples of such systems are operating systems and database systems. Both systems use resources such as CPU, memory and disk.

Scheduling is a well studied problem both from theoretical and practical perspectives. Typicallt a scheduling problems is tasked with an objective and with a goal to find a schedule that can either meet the objective or get close to it. TODO: For example, job scheduling with deadline. Many scheduling problems are difficult to solve (NP hard). Prior work has proposed heuristics or greedy approaches to find solutions to scheduling problems.

In this dissertation, we present scheduling for executing queries in a database system.

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* 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 2.1 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. This 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 [65]. This figure 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. 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. Quickstep targets this need, and in this

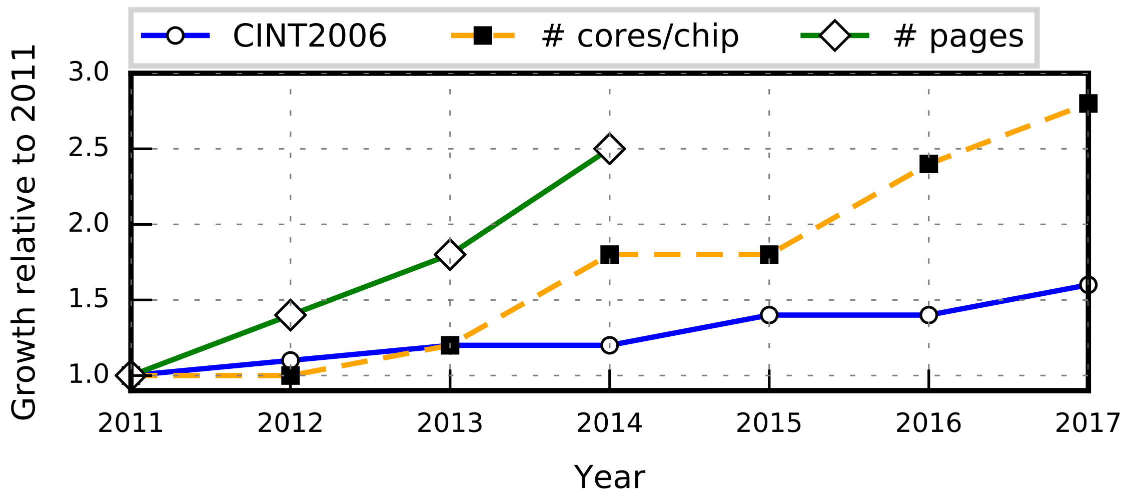


Figure 2.1: **Processor performance improvement as measured by the highest reported CINT2006 benchmark result for Intel Xeon chips from [81] compared to the number of pages indexed by Google (using estimates made by [82]). The figure does not show the increase in the number of queries (which is about 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 $\approx$4X from 2011 to 2017.**

paper we describe the initial version of Quickstep that targets single-node in-memory read-mostly analytic workloads.

To pay off the deficit, 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 2.1). 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 [21]. 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 equijoins, eliminating semi-joins entirely in some cases.

Overall, the key contributions of this paper 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

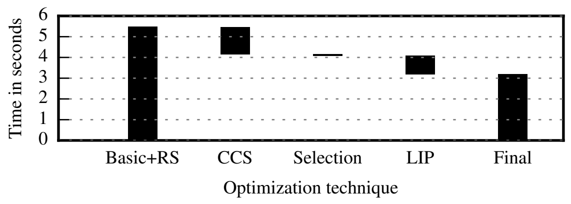


Figure 2.2: **A waterfall chart showing the impact of various techniques in Quickstep for query 10 from the TPC-H benchmark running on a 100 scale factor database. RS (Row Store), and CCS (Compressed Column Store) are both supported in Quickstep (see Section 2.3.1). Basic and Selection are template metaprogramming 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.**

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 2.2 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 equijoins using *exact filters*.

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 [92, 12], PostgreSQL [68], MonetDB [40], and VectorWise [96]. 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 knowkedge, 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 comparisions, e.g. [10], 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 reproducability goal that is being pursued in our community [19, 53, 54]. 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 paper 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.

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 this paper) and other classes of analytics/machine learning (using the approach outlined in [30, 93]). This paper 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. Currently, the system supports the following types: `INTEGER` (32-bit signed), `BIGINT/LONG` (64-bit signed), `REAL/FLOAT` (IEEE 754 *binary32*), `DOUBLE PRECISION` (IEEE 754 *binary64*), fixed-point `DECIMAL`, fixed-length `CHAR` strings, variable-length `VARCHAR` strings, `DATETIME/TIMESTA` (with microsecond resolution), date-time `INTERVAL`, and year-month `INTERVAL`.

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 [33] 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 [15]). 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 [21] 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 [37] 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

(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 2.3 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 2.3. 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 2.3, 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

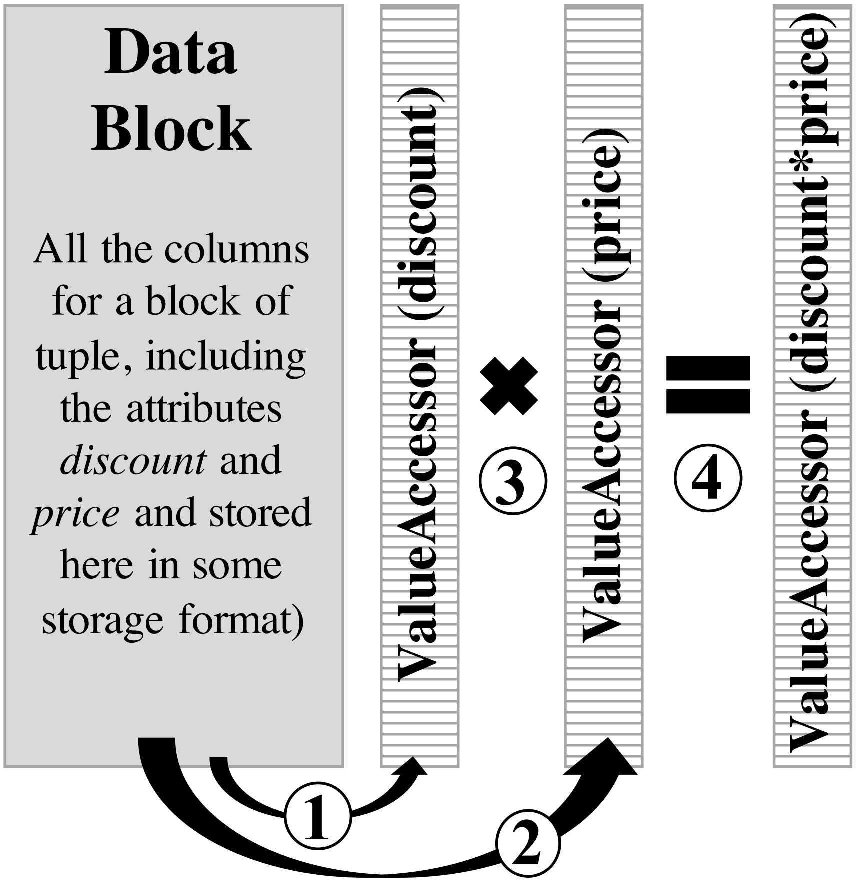

Data Block
All the columns for a block of tuple, including the attributes *discount* and *price* and stored here in some storage format)
ValueAccessor (discount)
ValueAccessor (price)
ValueAccessor (discount*price)
1
2
3
4

Figure 2.3: **Evaluation of the expression `discount*price`.**

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 [61].

Data from the storage manager can be persisted through a file manager abstraction that currently supports the Linux file system (default), and also HDFS [80].

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 2.4 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.

```sql
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

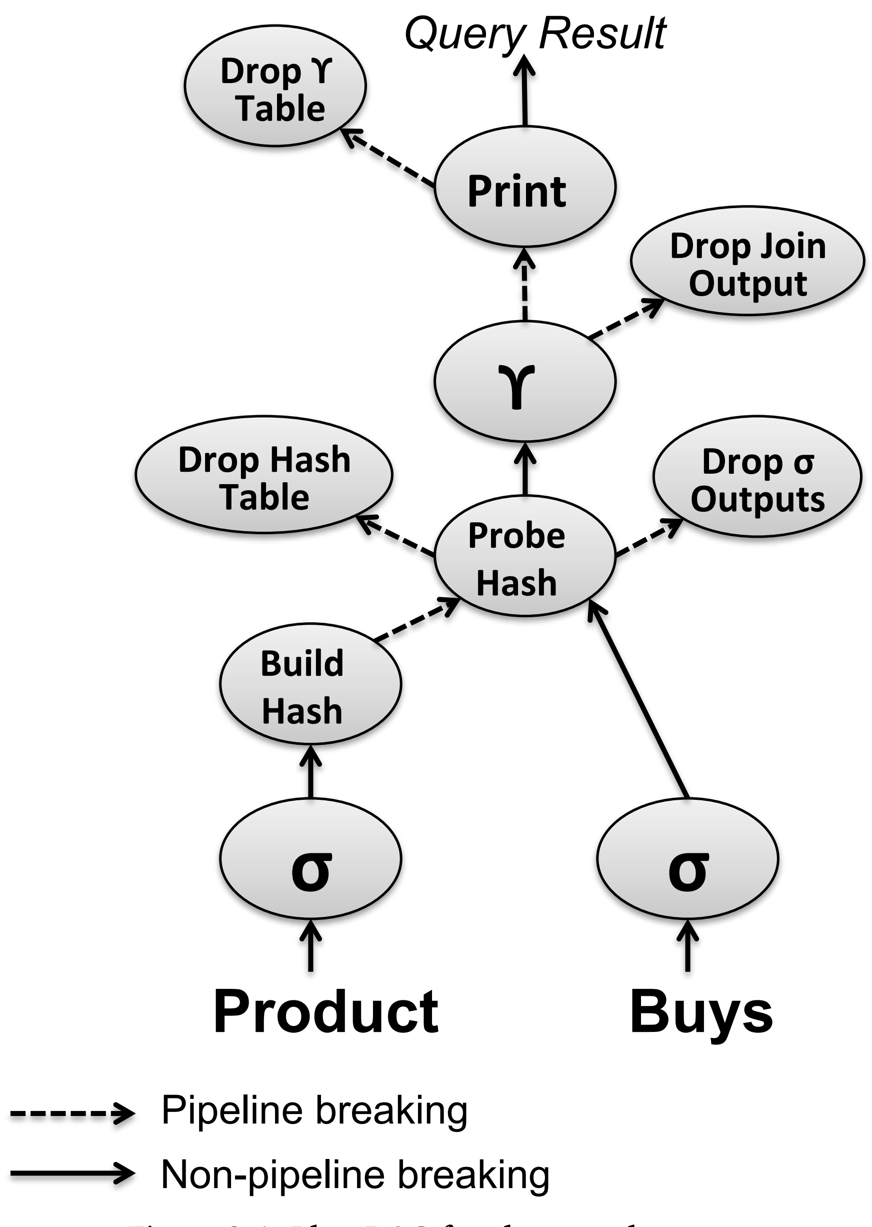


Figure 2.4: **Plan DAG for the sample query**

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 2.4 using the symbol σ) 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 2.4), 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 γ 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 σ 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 [46].

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 [27].

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 $i = 1, 2, \dots, n$. After relaxation, P then simplifies to $P'(R) = p_{1,1} \vee p_{1,2} \vee \dots \vee p_{1,n}$, 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 [18], 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 [55] and Oracle [63]. 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 in this paper (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 [62] query Q4.1 (see Figure 2.5a). 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 2.5b. 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.

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 [95] for more details.

Consider SSB Query 4.1 from Figure 2.5a 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 [16].

The LIP technique is based on semi-join processing and sideways information passing [14, 41, 13], 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 2.5c. 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 [95].

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 this paper.

The three open-source systems that we use are MonetDB [40], PostgreSQL [68] and Spark [92, 12] and the commercial system is VectorWise [96]. 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 this paper). VectorWise and Hyper [43] 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 [24], GreenPlum [35], and Redshift [75]. We use PostgreSQL v. 9.6.2, which includes about a decade's worth of work by the community to add intra-query parallelism [69]. 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 [76].

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 [77]. 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 2.6 shows the results for all systems when using the TPC-H dataset at SF 100 (∼100GB dataset).

As can be seen in Figure 2.6, 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 [79], 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 [18], 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 [66].

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. [10]; although this previous work has used indirect comparisons using the SSB benchmark and across two different systems.

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. [56, 39, 84]).

For this experiment, we use a specific schema-based denormalization technique that has been previously proposed [52]. 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 2.7. 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.

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. [10], 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) [83]. 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 [10]).

In Figure 2.8, 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 [10] 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.

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. (In the interest of space we do not present the detailed results.)

The results for TPC-H are similar, and omitted in the interest of space.

2.6.7 Template Metaprogramming Impact

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 2.9.

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 2.2 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).

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 2.10 shows the impact of these techniques over a baseline in which neither of these techniques are used. As shown in Figure 2.10, 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.

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 2.11. 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 2.11, 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 [25]), 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 2.12 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 the presentation of this paper, 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 [47, 17, 73, 43, 91, 11, 12, 96, 31, 64], 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).

Our vectorized execution on blocks has similarity to the work on columnar execution methods, including recent proposals such as [94, 74, 42, 88, 87, 51, 9, 70, 32]. 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 [42, 88, 87, 9, 70, 32, 38, 60, 12, 67, 57, 94, 74]). 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 [20]. 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. [21] and Leis et al. [48]. 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 [26]. 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 [14]. The LIP technique presented in Section 2.5.3 also draws on similar ideas, and is described in greater detail in [95].

Achieving *robustness* in query processing is a goal for many database systems [34]. 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 [72] for robustness, but their focus is largely on simpler scan operations.

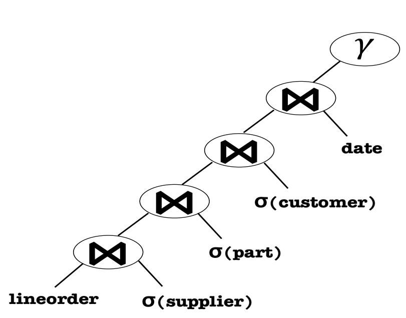
Overall, we articulate the growing need for the scaling-up approach, and present the design of Quickstep that is designed for a very high-level of intra-operator parallelism to address this need. We also present a set of related query processing and optimization methods. Collectively our methods achieve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

2.8 Conclusions & Future Work

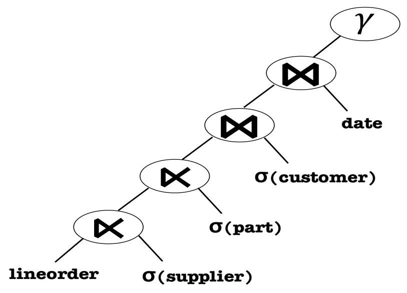
Compute and memory densities inside individual servers continues to grow at an astonishing pace. Thus, there is a clear need to complement the emphasis on “scaling-out” with an approach to “scaling-up” to exploit the full potential of parallelism that is packed inside individual servers.

This paper has 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.

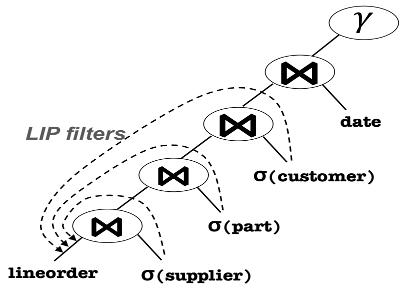
Aiming for higher performance is a never-ending goal, and there are a number of additional opportunities to achieve even higher performance in Quickstep. Some of these opportunities include operator sharing, fusing operators in a pipeline, improvements in individual operator algorithms, dynamic code 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.



(a) Original query plan



(b) Plan using join to semi-join transformation



(c) Query plan using LIP (only)

Figure 2.5: **Query plan variations for SSB Query 4.1**

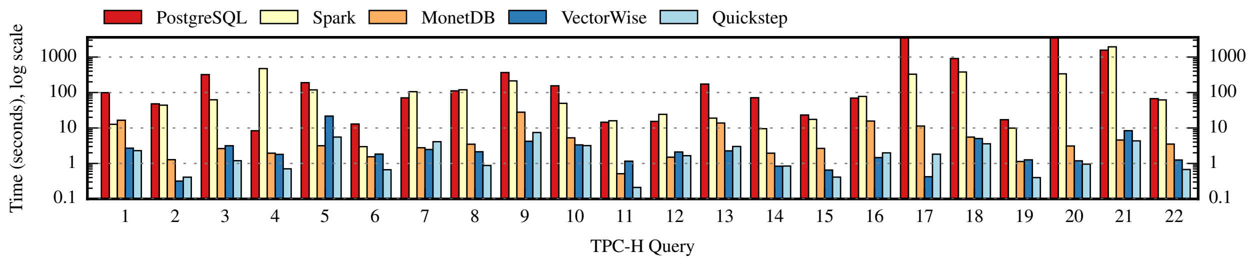


Figure 2.6: **Comparison with TPC-H, scale factor 100. Q17 and Q20 did not finish on PostgreSQL after an hour.**

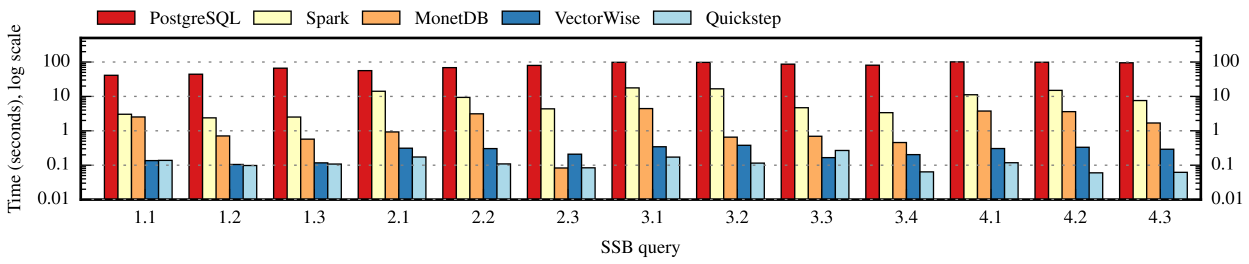
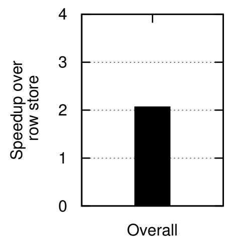
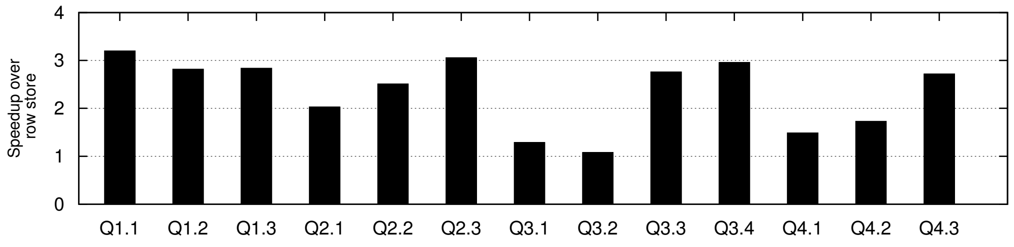


Figure 2.7: **Comparison with denormalized SSB, scale factor 50.**

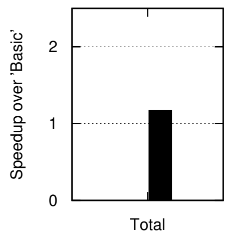


(a) Overall speedup

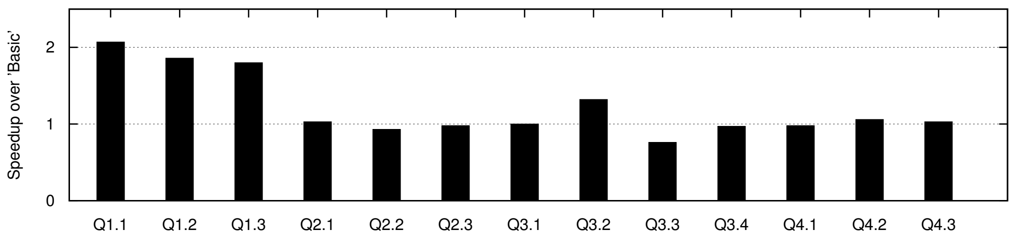


(b) Detailed query speedups

Figure 2.8: **Impact of storage format on performance for SSB scale factor 100**

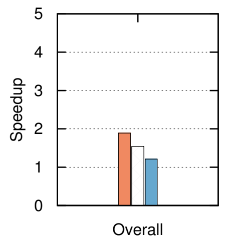


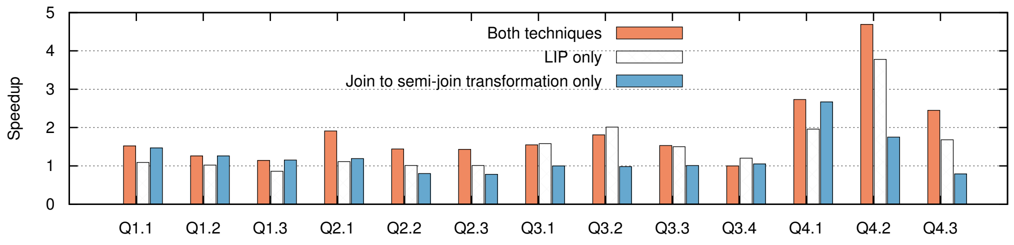
(a) Change in total time.



(b) Change in the detailed query execution times.

Figure 2.9: **Impact of template metaprogramming.**





(a) Overall speedup

(b) Breakdown of speedup due to techniques

Figure 2.10: **Impact of Exact Filter and LIP using SSB at scale factor 100.**

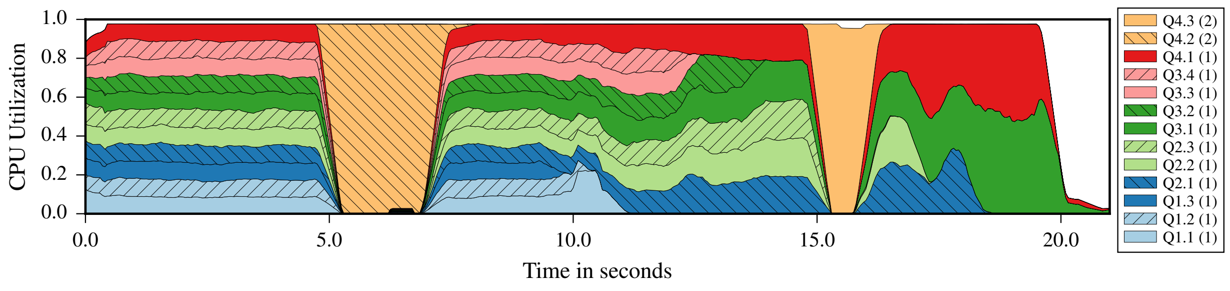


Figure 2.11: **Prioritized query execution. QX.Y(1) indicates that Query X.Y has a priority 1. Q4.2 and Q4.3 have higher priority (2) than the other queries (1).**

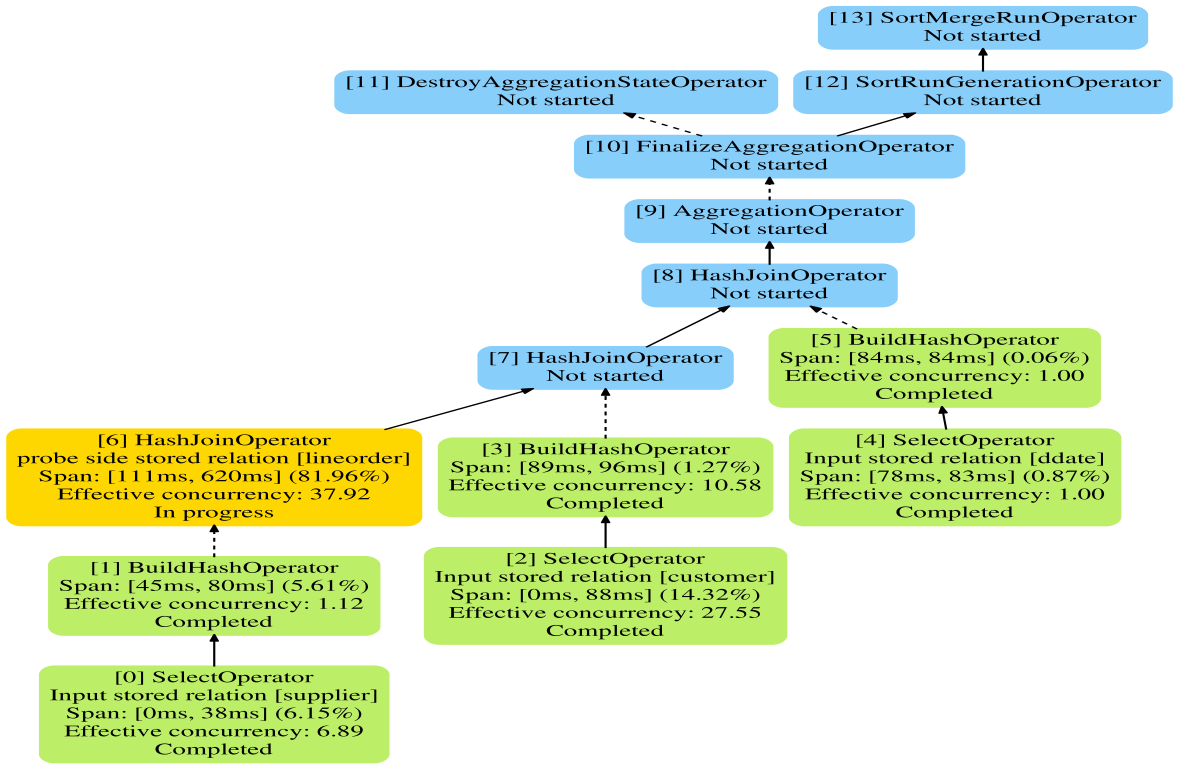

[13] SortMergeRunOperator
Not started
[11] DestroyAggregationStateOperator
Not started
[12] SortRunGenerationOperator
Not started
[10] FinalizeAggregationOperator
Not started
[9] AggregationOperator
Not started
[8] HashJoinOperator
Not started
[7] HashJoinOperator
Not started
[5] BuildHashOperator
Span: [84ms, 84ms] (0.06%)
Effective concurrency: 1.00
Completed
[6] HashJoinOperator
probe side stored relation [lineorder]
Span: [111ms, 620ms] (81.96%)
Effective concurrency: 37.92
In progress
[3] BuildHashOperator
Span: [89ms, 96ms] (1.27%)
Effective concurrency: 10.58
Completed
[4] SelectOperator
Input stored relation [ddate]
Span: [78ms, 83ms] (0.87%)
Effective concurrency: 1.00
Completed
[1] BuildHashOperator
Span: [45ms, 80ms] (5.61%)
Effective concurrency: 1.12
Completed
[2] SelectOperator
Input stored relation [customer]
Span: [0ms, 88ms] (14.32%)
Effective concurrency: 27.55
Completed
[0] SelectOperator
Input stored relation [supplier]
Span: [0ms, 38ms] (6.15%)
Effective concurrency: 6.89
Completed

Figure 2.12: **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.**

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 [59, 58]. 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 [22, 49, 86]. 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 [46]. 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 (the focused setting for this paper), 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 [1]. 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 3.5) 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 3.3. Section 3.4 describes the formulation of the policies and the load control mechanisms. Section 3.5 contains the experimental results. Related work is discussed in Section 3.6, and Section 3.7 contains our concluding remarks.

3.2 Background

In this section we establish some prerequisites for the proposed Quickstep scheduler framework. Quickstep [1] 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 paper. 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 (built using previously proposed approaches [22, 49]), in which 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 [22]. (c.f. [28] 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. Quickstep's work order abstraction is tied with its storage management, which we describe in [28].

3.3 Design of the Scheduler

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

3.3.1 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 (described in [28]) to determine when to activate nodes in the DAG.

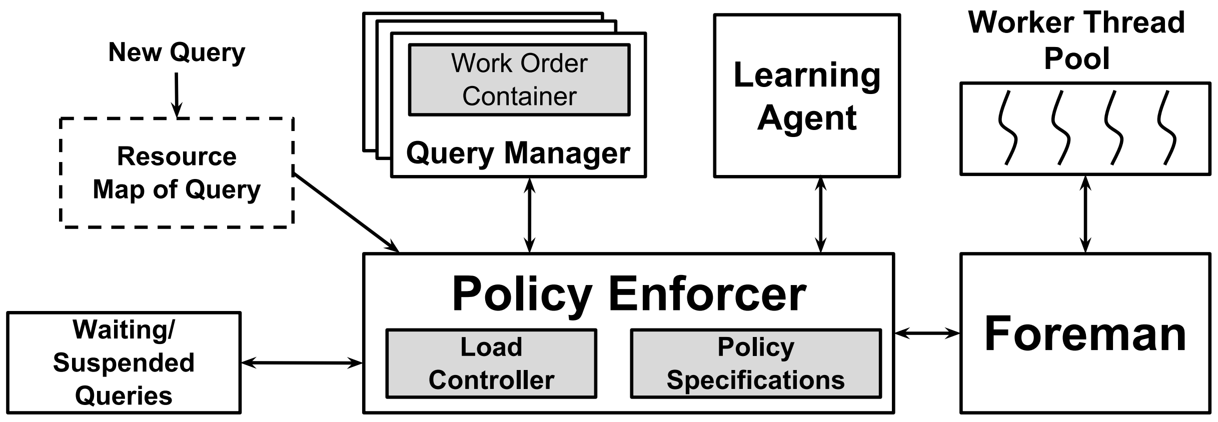


Figure 3.1: Overview of the scheduler

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.

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 3.3.2.

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 [28].

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 3.4.4.

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 3.3.3 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 (these details are presented in [28]).

Quickstep has two kinds of threads – a *scheduler* thread and many *worker* threads that perform the actual relational operations as defined by *work orders*. More details about the thread model can be found in [28].

3.3.2 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 3.4, 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.

3.3.3 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 3.4 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 [28].

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.

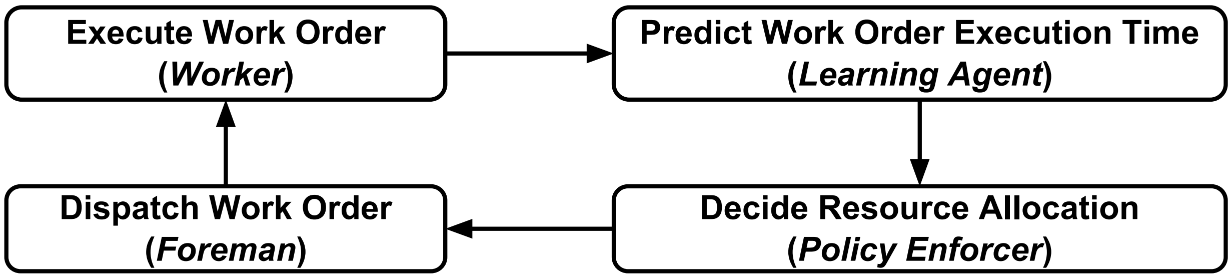


Figure 3.2: Interactions among scheduler components

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

Quickstep's scheduler currently uses 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 [28].

The problem of estimating the query execution time is well-studied, but requires complex methods [29, 89, 50, 23]. The Learning Agent does not require such methods. However, it can combine estimates from other methods with its own estimates.

[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.

3.4 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 [28].

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.

In Section 3.4.4, 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 3.4.1, 3.4.2, and 3.4.3 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).

3.4.1 Fair Policy Implementation

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

Observe that $pb_j \in (0, 1]$ and $\sum\limits_{j=1}^{k} pb_j = 1$. 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.

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

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.

3.4.2 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.

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

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.

3.4.3 Proportional Priority (PP) Implementation

Let $P = \{PV_1, PV_2, \dots PV_k\}$ 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 $t_{PV_i} = PV_i / \sum\limits_{j=1}^{k} PV_j$ 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 $t_1, t_2, \dots, t_m$, then the predicted work order execution time for the class is $w_{PV_i} = \sum\limits_{j=1}^{m} t_j / m$. 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.

3.4.4 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 3.3, 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.

3.5 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 [77] platform for our evaluation, which we described in Table 3.3.

3.5.1 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,

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

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.

3.5.2 Experimental Workload

For the evaluation, we use the Star Schema Benchmark (SSB) [62]. 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. [71]. 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].

3.5.3 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.

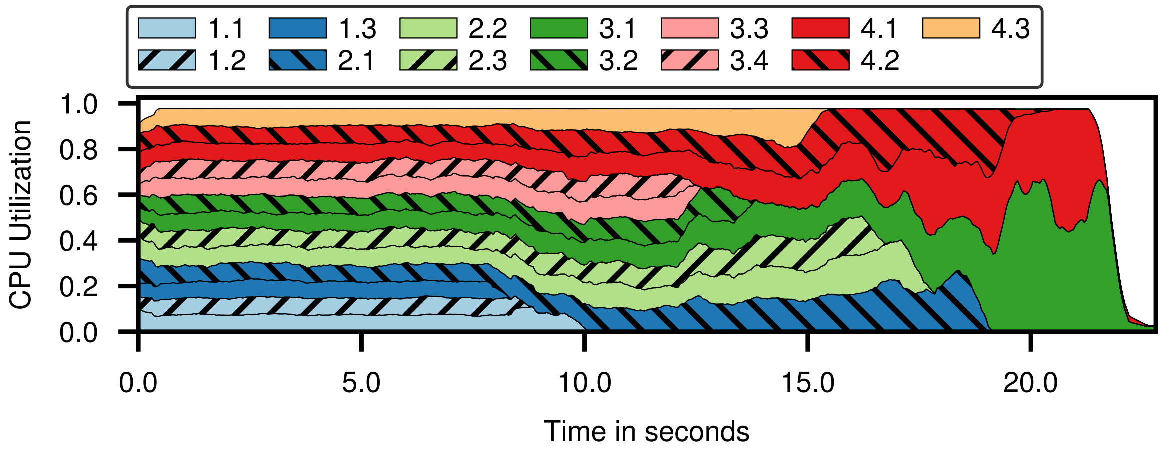


Figure 3.3: CPU utilization of queries in fair policy

3.5.3.1 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.

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

3.5.3.2 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

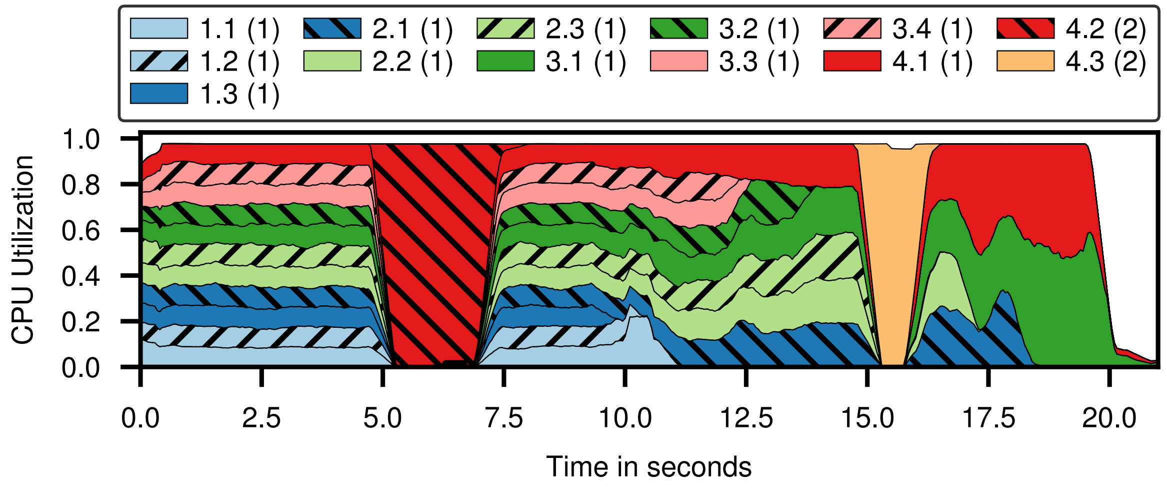


Figure 3.4: CPU utilization in HPF policy. Note that $a.b(N)$ denotes a SSB query $a.b$ with priority N

inject Q4.2 in the system at around 5 seconds and Q4.3 at around 15 seconds. Figure 3.4 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.

Due to space restriction, we omit the evaluation of Proportional Priority policy, and refer the interested reader to [28].

3.5.4 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 3.5 shows the results of this experiment.

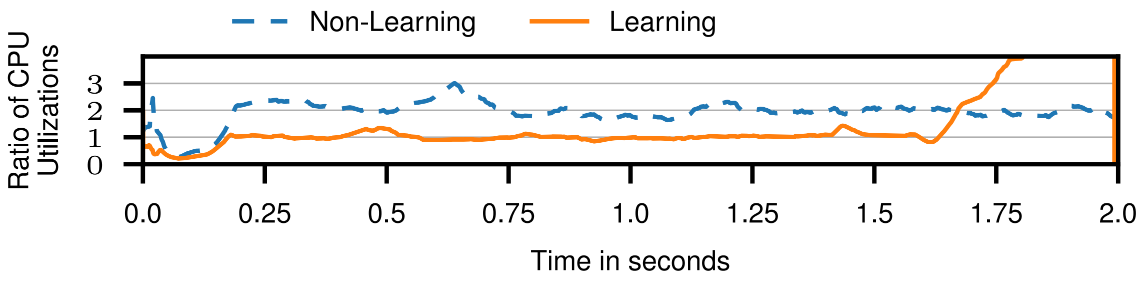


Figure 3.5: Ratio of CPU utilizations $\frac{Q4.1}{Q1.1}$ for learning and non-learning implementations

Observe in Figure 3.5, 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 2 in [28]). 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.

3.5.5 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 3.6 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 3.5), 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.

3.5.6 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 3.5.2). 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 3.7 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

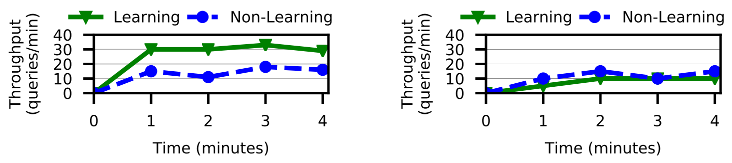


Figure 3.6: Impact of learning on the throughput

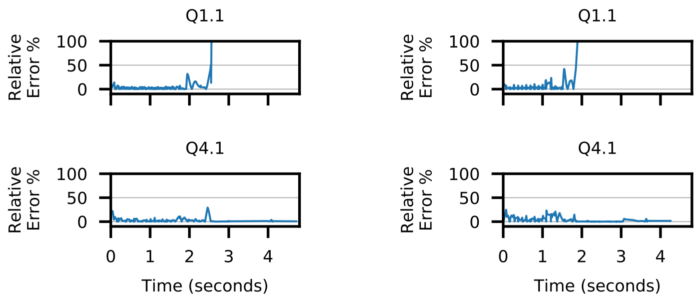
change in the execution plan. Note that the scheduler learns the phase changes quickly, and adjusts its estimates after each phase change.

3.5.7 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 3.8 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 3.8, 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.



(a) Skewed dataset (b) Uniform dataset

Figure 3.7: Comparison of predicted and observed time per work order

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.

3.6 Related Work

The *work orders* abstraction is similar to other abstractions like *morsels* in Hyper [49] and the *segment-based parallelism* [86]. Such abstractions provide a means to achieve high intra-query, intra-operator data parallelism. Hyper [49] 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 [86]), cache-conscious task scheduling.

The elastic pipelining implementation [86] 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.

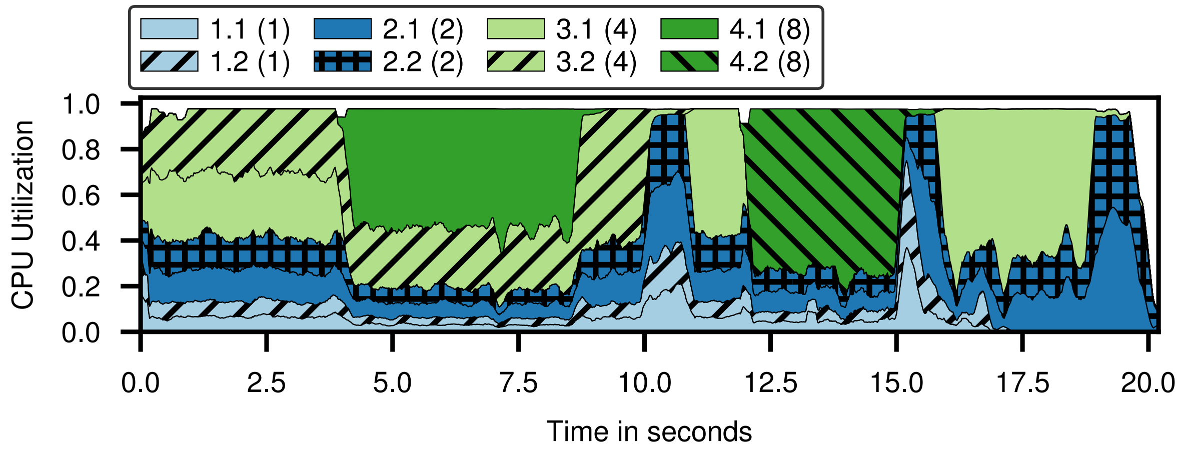


Figure 3.8: Load control: An SSB query $a.b$ with priority N is denoted as $a.b(N)$

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 [36] 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 [5, 6, 4, 8, 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 [44, 45] 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 [89, 90, 29] 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 [85] 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 [46] - a common theme found in the OS literature.

Deficit Round Robin (DRR) [78] 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.

Our scheduler is designed for an in-memory analytical database system. There are several such systems such as MonetDB [40], Spark SQL [7].

3.7 Conclusions and Future Work

We present a scheduler framework built on the principle of separation of policy and mechanism. It supports a wide-range of policies, even in dynamic workload settings and without requiring complex and accurate estimates from a query optimizer. The proposed framework is holistic as it also incorporates a load control mechanism. We have implemented our methods in an open-source in-memory database Quickstep, and also demonstrated the effectiveness of our approach. There are many directions for future work, including extending the scheduler framework to the distributed version of Quickstep.

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] 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`.

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

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

[7] Spark SQL & Dataframes | Apache Spark. `http://spark.apache.org/sql/`.

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

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

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

[11] 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.

[12] 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.

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

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

[15] 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.

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] C. Chasseur and J. M. Patel. Design and evaluation of storage organizations for read-optimized main memory databases. *PVLDB*, 2013.

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

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

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

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

[27] H. Deshmukh, H. Memisoglu, and J. M. Patel. Adaptive concurrent query execution framework for an analytical in-memory database system. *IEEE BigData Congress*, 2017.

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

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

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

[31] 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.

[32] 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.

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

[34] 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.

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

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

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

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

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

[40] 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.

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

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

[43] 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.

[44] 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.

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

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

[47] 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.

[48] 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.

[49] 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.

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

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

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

[53] 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.

[54] 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.

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

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

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

[58] 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.

[59] 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.

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

[61] 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.

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

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

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

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

[66] 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.

[67] 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.

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

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

[70] 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.

[71] 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.

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

[73] 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.

[74] 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.

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

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

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

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

[79] 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.

[80] 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.

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

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

[83] 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.

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

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

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

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

[88] 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.

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

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

[91] 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.

[92] 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.

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

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

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

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