Quickstep: A Data Platform Based on the Scaling-In Approach

Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Hakan Memisoglu,
Navneet Potti, Saket Saurabh, Marc Spehlmann, Zuyu Zhang
University of Wisconsin – Madison

Submission Type: Research

Abstract

Modern servers pack enough storage and computing power that just a decade ago was spread across a modest-sized cluster. This paper presents a prototype system, called Quickstep, to exploit the large amount of parallelism that is packed inside such modern servers. Quickstep builds on a vast body of previous work on methods for organizing data, optimizing, scheduling and executing queries, and brings them together in a single system. Quickstep also includes new query processing methods that go beyond previous approaches. To keep the project focused, the project's initial target is read-mostly in-memory data warehousing workloads in single-node settings. In this paper, we describe the design and implementation of Quickstep for this target application space. In this paper, we also present experimental results comparing the performance of Quickstep to a number of other systems. These experiments show that Quickstep is often faster than many other contemporary systems, and in some cases faster by an order-of-magnitude. Quickstep is an Apache (incubating) project and lives at: `https://github.com/apache/incubator-quickstep`.

1 Introduction

Query processing systems today face a host of challenges that were not as prominent just a few years ago. First, the hardware landscape has changed dramatically in recent years, driven by the need to consider energy as a first-class (hardware) design parameter. Consequently, across the entire processor-IO hierarchy, the hardware paradigm today looks very different than it did just a few years ago. Because of this shift, 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 1 illustrates this deficit issue by comparing improvements in processor performance (blue line) with just the growth rate of data (green line) that is indexed by Google. 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 [41]. This figure also shows (using a dotted orange line) the growth in the number of cores/processor over time. As one can observe, the number of cores/processor is rising rapidly as multi-core techniques are critical to realizing overall higher processor performance. In addition, as noted in the caption of the figure, 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-in* to exploit the full (parallel) processing power that is locked in commodity servers today. Quickstep targets this need, and in this paper we describe the initial version of Quickstep that targets single-node settings, and for the important case of in-memory read-mostly analytic workloads.

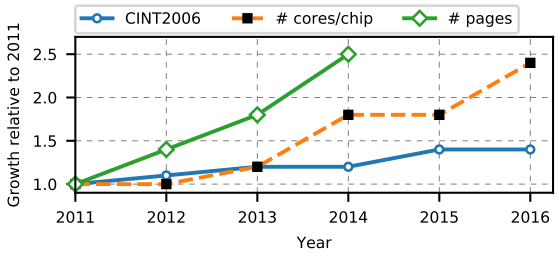


Figure 1: Processor performance improvement as measured by the highest reported CINT2006 benchmark result for Intel Xeon chips from [50] compared to the number of pages indexed by Google (data till 2014, which is the last year for which estimates made by [51] are currently available). The figure does not show the increase in the number of queries (which is about 2.5X for Google search queries over the same period), and the increase in the complexity of queries as applications request richer analytics. These aspects make the deficit problem worse. Also shown in the figure is the maximum number of cores 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 have grown by about 4X between 2011 and 2016.

Next, we describe key aspects of the Quickstep system. First, to pay off the deficit (discussed above), we have taken the approach of thinking bottom-up from the hardware to the software. A crucial design is to use mechanisms that allow for *high intra-operator paral-*

lelism. Such mechanisms are important to exploit the full potential of the high level of hardware compute parallelism that is present in modern servers (the dotted orange line in Figure 1). Unlike most database management systems (DBMSs), Quickstep has a storage manager with a block layout, where each block behaves like a mini self-contained database [13]. 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 is designed to cleanly separate (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.

Quickstep is designed to be easy to use in a variety of settings, including as a standalone server and as an embedded database engine in virtualized containers. Quickstep employs a unified buffer manager to store both the database tables and any intermediate results (including hash tables that are built during query execution). In many database systems various memory knobs have to be "set right" for high performance. Quickstep does not require such tuning. Similarly, Quickstep automatically senses the parallelism in the hardware environment (number of cores) and uses this information to naturally set the degree of parallelism for query execution. Thus, Quickstep can be spun up in any standalone server/container without requiring tuning knobs, and the system auto-tunes to exploit the full parallelism potential in the underlying hardware.

The contributions of this paper are as follows:

- We present the end-to-end design for Quickstep, which brings together in a single artifact a number of methods for in-memory query processing.
- We present how Quickstep uses query processing techniques for aggressively pushing down complex disjunctive predicates involving multiple relations, as well as for eliminating certain types of equijoins using *exact filters*.
- The design of the system also focuses on ease-of-use paying attention to a number of issues, including not relying on performance knobs by 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.
- We present results from an initial end-to-end evaluation that compares Quickstep with a number of other existing systems (Spark, PostgreSQL, MonetDB and VectorWise). Our results show that in some cases, Quickstep is faster by an order-of-magnitude over some of these existing systems.
- We make Quickstep available as open-source. We note that our community has attempted to take reproducibility seriously [12, 34, 35], which is naturally easier to achieve with open-source systems. However, open-sourcing a system goes beyond reproducibility as it allows for *transparency* that permits a deeper understanding of the end-to-end system effects. This aspect is especially critical when working on research problems where the impact of specific techniques is important only when cast within the context of the overall system behavior.

2 QUICKSTEP Architecture

At its core, 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 warehousing analytics (the focus of this paper) and other classes of analytics/machine learning (using the approach outlined in [17, 58]). This paper focuses only on SQL analytics.

2.1 Data Model and Query Language

Quickstep uses a relational data model, and SQL as the query language. Currently, the system supports the following basic types: `INTEGER` (32-bit signed), `BIGINT`/`LONG` (64-bit signed), `REAL`/`FLOAT` (IEEE 754 *binary32* format), `DOUBLE PRECISION` (IEEE 754 *binary64* format), fixed-point `DECIMAL`, fixed-length `CHAR` strings, variable-length `VARCHAR` strings, `DATETIME`/`TIMESTAMP` (with microsecond resolution), date-time `INTERVAL`, and year-month `INTERVAL`.

2.2 System Overview

The internal architecture of Quickstep resembles the architecture of a typical database engine, with perhaps the exception of giving a first-class role to the query scheduler (which is described in more detail in Section 4.2). 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 [20] 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, which includes 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 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 4.

The *relational operator library* contains implementation of various relational operators. Currently, the system has implementations for the following operators: select, project, join (equi-join, semi-join, anti-join and outer-join), 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 [8]). 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 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`, the operator computes local aggregates for each input block, which are then merged to compute the global aggregate. For aggregation with `GROUP BY`, the operator builds a global latch-free hash table of aggregation handles 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. In the second phase, runs of sorted blocks are merged to produce a fully sorted output relation.

3 Block-based Storage Manager

The Quickstep storage manager [13] is based on a block-based architecture. 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 4 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. Quickstep supports type-specific order-preserving compression schemes.

3.1 Template Metaprogramming

As noted above, Quickstep has a variety of different storage block formats. A template metaprogramming-based implementation is used 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 the STL, the implementations of containers (such as vectors and maps) and algorithms (like find and sort) are separated from each other. The crucial abstraction that enables this separation is the notion of iterators, which allow containers to expose common data access patterns (hiding implementation details) and algorithms to be implemented against the iterator interface (rather than containers directly).

Quickstep has an analogous design where various `ValueAccessor`s mediate access to data residing in corresponding storage block types, and all relational operators are implemented against this `ValueAccessor` interface. We then use C++ template metaprogramming to statically generate code for all execution choices (i.e., relational operators and block types). In contrast to run-time code generation techniques, there is no additional run-time cost, but there is an increase in the (one-time) compilation cost and the size of the compiled binary. A key benefit of this approach is that a clean and simple implementation of these `ValueAccessor`s (e.g., one that avoids branching and indirection) usually makes them amenable to hardware prefetching and compiler auto-vectorization.

3.2 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 [39].

Data from the storage manager can be persisted through a file manager abstraction that currently supports the Linux file system (default), and also HDFS [4].

4 Query 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 4.1).

Second, in our scaling-in approach, it is crucial to maximally 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 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 concurrently executing queries, the scheduler design separates the choice of policies from the mechanism of their implementation (cf. Section 4.3).

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 6.8.

The workers are responsible for executing the relational operation tasks that are scheduled by the scheduler. 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 cost. The workers are stateless; thus, the worker pool can *elastically* grow or shrink dynamically.

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 a physical query plan represented as a directed acyclic graph (DAG) in which each node is a relational operator. Figure 2 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.

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

4.2.1 Work Order

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

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 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 results are 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.

4.2.3 Implementation of Pipelining

In our example DAG, 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.

4.2.4 Output Management During Query Execution

During query execution, intermediate results are written to temporary blocks. To minimize internal fragmentation, 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.

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 query $\rightarrow$ dispatching the work order to a worker thread. When concurrent

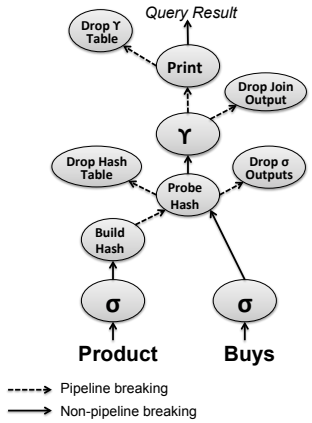


Figure 2: DAG plan for the sample query.

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*. One concrete example of such policy is *Fair* which can be interpreted as follows: 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 6.7.) 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. 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 [29].

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 mathematical formulation of probability values for different policies implemented in Quickstep and their relation with the estimated work order execution time is presented in [16].

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 6.8).

5 Efficient Query Processing

Quickstep builds on a number of existing query processing methods (as described in Section 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.

5.1 Partial Predicate Push-down

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

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

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

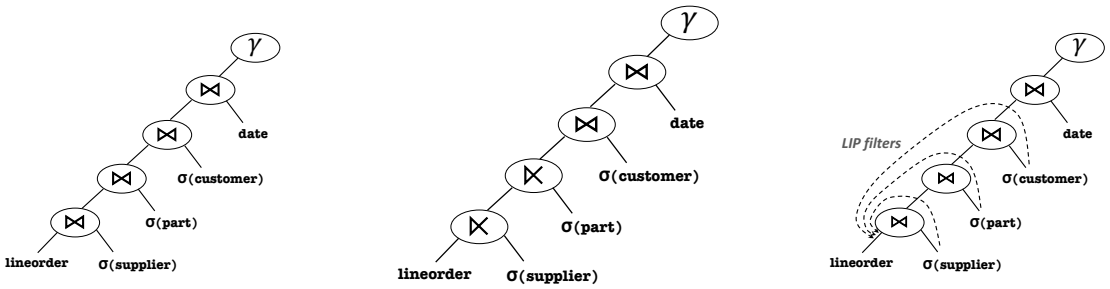
The predicate P' can now be pushed down to R, which often leads to significantly fewer redundant tuples flowing through the rest of the plan. However, since the exact predicate must later be evaluated again, such a partial push down is only useful if the predicate is selective. Quickstep uses a rule-based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

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*, and we describe it in more detail below.

To illustrate this technique, consider the SSB query Q4.1 (see Figure 3a). 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. Based on these observations, we can transform the `lineorder` – `part` join into a semi-join, as shown in Figure 3b. 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 Exact Filter is then probed using the `lineorder` table. The `lineorder` – `supplier` join also benefits from this optimization.

The implementation of semi-join operation using Exact Filter 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. Further, since the filter is far more succinct than a hash table, it greatly



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

Figure 3: SSB Query 4.1

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 Exact Filters.

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 [60] for more details.

Consider SSB Query 4.1 from Figure 3a 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 [9].

The LIP technique is based on semi-join processing and sideways information passing [6, 7, 25], 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 3c. 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. These details are discussed in [60].

6 Evaluation

In this section, we present results from an empirical evaluation comparing Quickstep with a number of other systems. We note that a large number of different SQL data platforms have been built over the past four decades. A comparison of all systems in this ecosystem is beyond the scope of this paper. Thus for this evaluation, we chose three other open-source systems and one commercial system that each have different approaches to high performance analytics, and support stand-alone/single node in-memory query execution.

The three open-source systems that we use are MonetDB, PostgreSQL and Spark and the commercial system is VectorWise [61]. (We would have liked to try Hyper [27], 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.) We also note that the open-source nature of Quickstep means that anyone can use Quickstep for benchmarking without needing to hide the product name, allowing more transparent comparisons across different papers on the same topic. Furthermore, access to the source code allows one to better understand the reasons behind certain performance behaviors, which is otherwise hard to do when only binaries are available.

Next, we outline our reasons for choosing these four systems. MonetDB [24], is an early column-store database engine that has seen over two decades of development. We use their latest release (December 2016 release and the associated bugfixes). We also compare with VectorWise, which is a commercial column store system

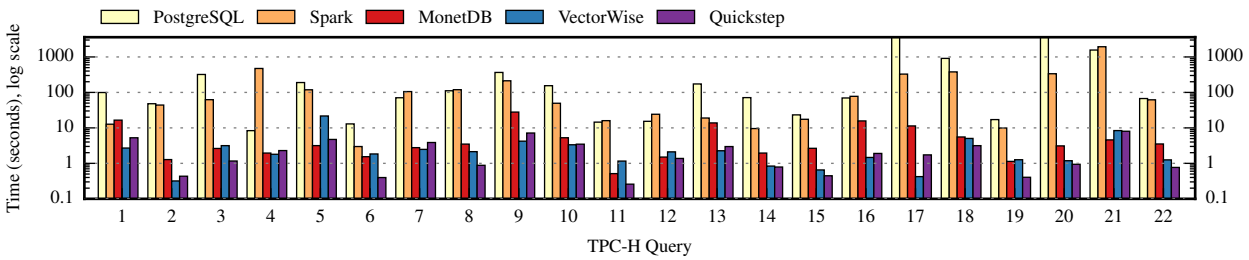


Figure 4: Comparison with TPC-H, scale factor 100. Q17 and Q20 did not finish on PostgreSQL after an hour.

with origins in MonetDB. We use the latest release that is available for free evaluation. PostgreSQL [42] 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 [14], GreenPlum [21], and Redshift [47]. We use the latest release of PostgreSQL, namely v. 9.6.2, which includes about a decade's worth of work by the community to add intra-query parallelism [43]. We chose Spark [5, 57] as it is an increasingly popular, and arguably the dominant, 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 [48].

6.1 Workload

For the evaluation, we use the TPC-H benchmark at scale factor 100 (~100GB in size) as well as the Star Schema Benchmark (SSB) at scale factors 50 and 100 (~50 GB and 100 GB in size). Both these benchmarks illustrate workloads for decision support systems. The TPC-H database contains 8 tables in a snowflake schema, with two large fact tables (`lineitem` and `orders`) and six dimension tables of widely varying sizes. For evaluation, we use the 22 read-only queries in the benchmark, which vary greatly in complexity.

We also use the Star Schema Benchmark (SSB) [40] which is a simpler version of the TPC-H benchmark. The SSB database contains 5 tables in a star schema, reflecting the data model often resulting from the popular Kimball [28] approach for data warehouse schema creation. The workload consists of a set of 13 read-only queries that are significantly simpler than the TPC-H queries. The simplicity of SSB makes it an appealing (albeit simpler) workload to reason about the effects of various system design choices, and as such, has been used extensively in prior work, e.g. [2, 53].

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.

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. The server has two Intel Xeon Intel E5-2660 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. This server also has two 1.2 TB 10K RPM SAS HDDs, and one 480 GB SAS SSD device.

6.3 System Tuning

Tuning systems for optimal performance is a cumbersome task. One of the goals of Quickstep is to operate at high performance without requiring the user to set performance "knobs." 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. Thus, there are no tuning knobs in Quickstep, making it easy to operate in any environment, including containers of varying sizes.

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.

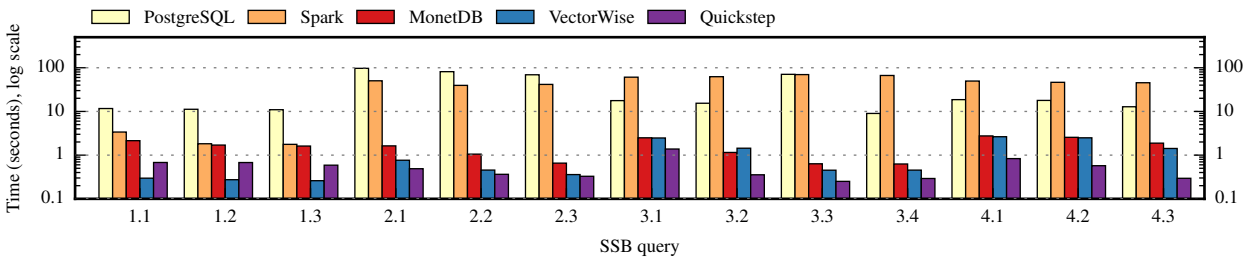


Figure 5: Comparison with SSB, scale factor 100.

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. These queries included the three most complex queries in the SSB benchmark (the Q4.X queries). Spark chose a Cartesian product of the dimension tables for these queries. As a result, these queries ran for a long time and eventually 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.

6.4 TPC-H at Scale Factor 100

Figure 4 shows the results for all systems when using the TPC-H dataset at SF 100 (~100GB dataset).

As can be seen in Figure 4, 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 53s and 70s respectively, making Quickstep 25% faster than VectorWise. Across each query, there are queries where each system outperforms the other significantly. 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 4 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 [49], 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). Finally, the lack of code-generation methods for predicate evaluation in Quickstep leads to a 2.5X slowdown in query 1, which contains many arithmetic expressions. (Addressing this is part of future work.)

On the other hand, Quickstep significantly outperforms VectorWise (at least 50% speedup) in 9 of the 22 queries. Across the board, the use of LIP and exact filters improves Quickstep's performance by about 2X. In particular, we believe that 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

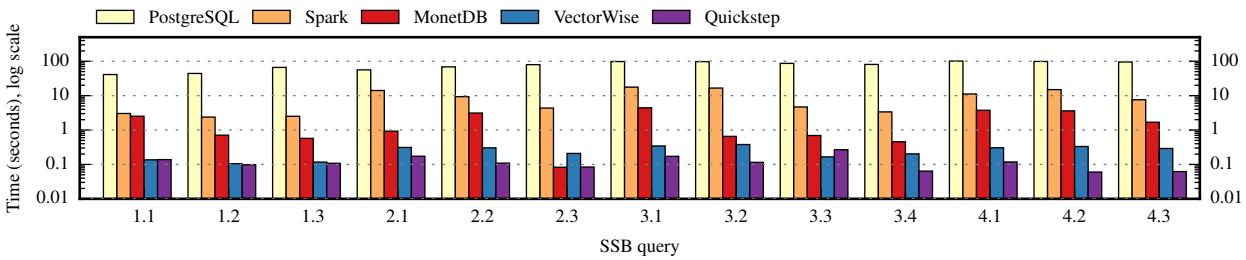


Figure 6: Comparison with denormalized SSB, scale factor 50.

in Quickstep. This query benefits significantly from the partial predicate push-down technique (cf. Section 5.1). VectorWise appears to also do predicate pushdown [11], but its approach may not be as general as our approach.

For the remaining 9 queries, Quickstep and VectorWise have comparable running times.

6.5 SSB at Scale Factor 100

In this next experiment, we use the SSB benchmark at scale factor 100. Compared to the previous experiment, this benchmark shows the impact of using a simpler schema and simpler queries for warehousing workloads. The 13 queries in SSB are divided into four classes, and the queries in each class are similar in their structure and complexity (number of joins and tables). In the discussion below, we refer to each query as **QX.Y**, where X is the class ($1 \leq X \leq 4$) and Y is the query number within the class. Classes 1, 2 and 4 have three queries each, while class 3 has four queries. Queries in the first class have one join operation, queries in the second and third classes have three join operations, and queries in the last class have four join operations. This arrangement of query classes in SSB allows one to examine the impact of increasing join query complexity on the systems.

The results for this experiment are shown in Figure 5. This figure shows that compared to MonetDB, PostgreSQL and Spark, the gains for Quickstep are consistent across the board, from the simpler queries (Q1.Y which have only one join) to the more complex queries (Q4.Y which have four joins). Quickstep is often **more than 10X** faster than PostgreSQL and Spark.

Quickstep is also faster than MonetDB and VectorWise across the queries. The total execution time for all the queries with Quickstep, VectorWise, and MonetDB is 7s, 14s, and 21s, respectively, resulting in an overall **2X** and **3X** improvement in performance compared to VectorWise and MonetDB respectively. The gains for Quickstep come from its ability to naturally run each operator using a high level of intra-operator parallelism and the use of the Exact Filter and LIP (cf. Section 5) methods. (There are no opportunities for the push-down technique discussed in Section 5.1 with SSB.) Quickstep particularly outperforms MonetDB and VectorWise on the more complex Q4.X queries that have a long chain of join, which are particularly amenable for the Exact Filter and LIP techniques.

6.6 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. [23, 36, 52]).

For this experiment, we use a specific schema-based denormalization technique that has been previously proposed [33]. 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 6. The total time to run all the thirteen queries was 1.6s, 3.2s, 23.2s, 1,014s, and 111.9s across Quickstep, VectorWise, MonetDB, PostgreSQL and Spark respectively. Quickstep's relative advantage over MonetDB now increases to over an order-of-magnitude (**14X**) across most of the 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

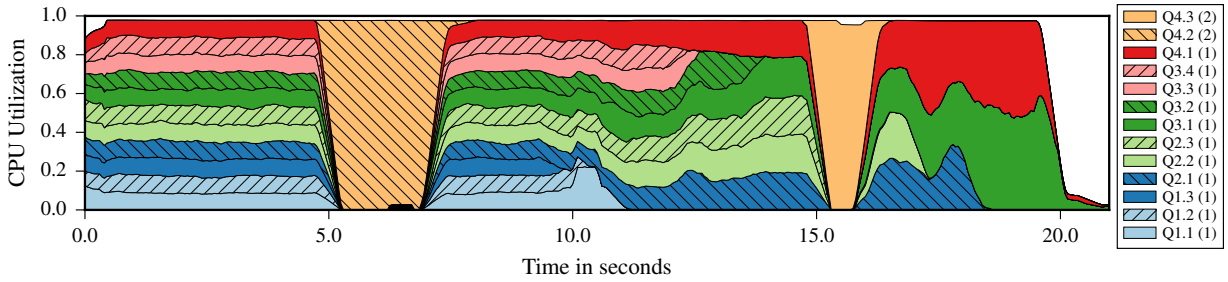


Figure 7: Prioritized query execution. QX.Y(1) indicates that Query X.Y has a priority 1. Q4.2 and Q4.3 have higher priority (2) than the other queries (1).

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.

6.7 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 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 7. 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 7, 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 stops scheduling work orders from 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 [15]), which is an important concern for managing resources in actual deployments.

6.8 Built-in Query Progress Monitoring

An interesting aspect of using a work-order based scheduler (described in Section 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 graphically shown to the user. As an example, Figure 8 shows the progress of a query with three join operations, one aggregation, and one sort operation.

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

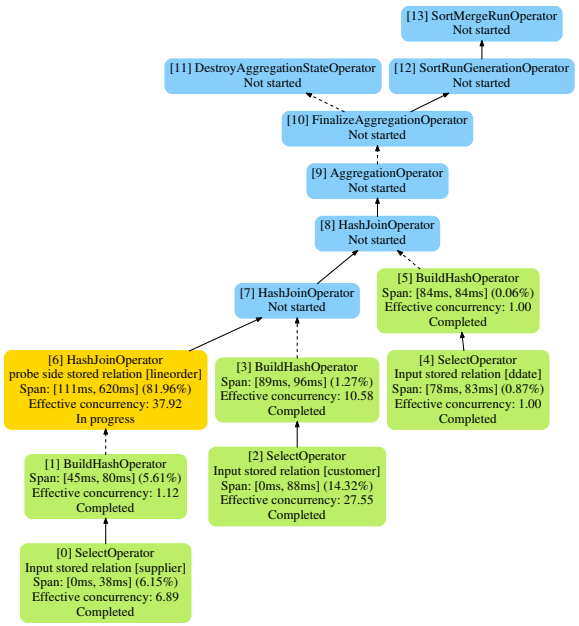


Figure 8: Query progress status. Green nodes (0-5) indicate work that is completed, the yellow node (6) corresponds to operators whose work-orders are currently being executed, and the blue nodes (7-13) show the work that has yet to be started.

developed, including [3,5,10,18,27,30,45,56,61], 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 6.7).

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 [1,19,26,32, 37, 44, 46, 54, 55, 59]), and we plan to add run-time code generation to the system in the future.

Our use of a block-based storage design naturally leads to a block-based scheduling method for query processing, and this connection was first articulated in [13]. The recent morsel-based query processing [31] method also philosophically belongs to this style of query processing.

The drive to extract higher performance from existing hardware has re-kindled interest in using code generation for queries at run-time. While this technique has been around for many decades [22], it is now making a come-back (e.g. [5, 38]). In contrast, our template metaprogramming-based approach in Quickstep 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 prohibitively expensive for short-running queries. An interesting direction for future work is to consider combining these two approaches.

There is a quick reference to a join-dependent expression filter pushdown technique in [11], 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. The exact filters build on the rich history of semi-join optimization dating back at least to Bernstein and Chiu [7]. The LIP technique presented in Section 5.3 also draws on similar ideas, and is described in greater detail in [60].

Overall, we articulate the growing need for the scaling-in 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.

8 Conclusions and 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-in" 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-in 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.

References

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

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

[3] 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.

[4] Apache Foundation. The Hadoop Distributed File System. `https://hadoop.apache.org`, 2015.

[5] 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.

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

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

[8] 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.

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

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

[11] 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.

[12] 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.

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

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

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

[16] H. Deshmukh, H. Memisoglu, and J. M. Patel. Adaptive concurrent query execution framework for an analytical in-memory database system. *IEEE Big-Data Congress (to appear)*, 2017.

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

[18] 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.

[19] 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.

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

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

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

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

[24] 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.

[25] Z. G. Ives and N. E. Taylor. Sideways information passing for push-style query processing. In *ICDE '08*, pages 774–783, 2008.

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

[27] 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.

[28] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2002.

[29] B. W. Lampson and H. E. Sturgis. Reflections on an operating system design. *Commun. ACM*, 1976.

[30] 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 *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, pages 1159–1168, 2013.

[31] 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.

[32] Y. Li and J. M. Patel. Bitweaving: Fast scans for main memory data processing. In *SIGMOD*, pages 289–300, 2013.

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

[34] 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.

[35] 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.

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

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

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

[39] 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.

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

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

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

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

[44] 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.

[45] 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.

[46] 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.

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

[48] Reynold Xin. Technical Preview of Apache Spark 2.0. `https://databricks.com/blog/2016/05/11/apache-spark-2-0-technical-preview-easier-fast`
`html`.

[49] 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.

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

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

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

[53] S. Tu, M. F. Kaashoek, S. Madden, and N. Zeldovich. Processing analytical queries over encrypted data. *PVLDB*, 6(5):289–300, 2013.

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

[55] 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.

[56] 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.

[57] 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.

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

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

[60] J. Zhu, N. Potti, S. Saurabh, and J. M. Patel. Looking ahead makes query plans robust. *Proceedings of the VLDB Endowment*, 10(8), 2017.

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