Adaptive Concurrent Query Execution Framework for an Analytical In-Memory Database System

Harshad Deshmukh Hakan Memisoglu Jignesh M. Patel
University of Wisconsin - Madison
{harshad, memisoglu, jignesh}@cs.wisc.edu

***Abstract*—There is a growing need for in-memory database analytic services, especially in cloud settings. Concurrent query execution is common in such environments. A crucial deployment requirement is to employ a concurrent query execution scheduling framework that is flexible, precise, and adaptive to meet specified deployment goals. In addition, the framework must also aim to use all the underlying hardware resources effectively (for high performance and high cost efficiency). This paper focuses on the design and evaluation of such a scheduler framework. Our scheduler framework incorporates a design in which the scheduling policies are cleanly separated from the scheduling mechanisms, allowing the scheduler to support a variety of policies, such as fair and priority scheduling. The scheduler also contains a novel learning component to monitor and quickly adapt to changing resource requirements of concurrent queries. In addition, the scheduler easily incorporates a load controller to protect the system from thrashing in situations when resources are scarce/oversubscribed. We have implemented our scheduling framework in an in-memory database engine, and using this implementation we also demonstrate the effectiveness of our approach. Collectively, we present the design and implementation of a scheduling framework for in-memory database services on contemporary hardware in modern deployment settings.**

I. 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 [1], [2]. 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 [3]–[5]. 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 try to understand the goals for such a framework, and to do that, we relate it to a governance model. Because in essence, the framework *governs* the use of resources for execution of 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 [6]. 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 [7]. 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 policy 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 V) 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 the model to predict future resource consumption needs 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 II describes some preliminaries related to Quickstep. The architecture of the scheduler framework is described in Section III. Section IV describes the formulation of the policies and the load control mechanisms. Section V contains the experimental results. Related work is discussed in Section VI, and Section VII contains our concluding remarks.

II. Background

In this section we establish some prerequisites for Quickstep's proposed scheduler framework. Quickstep 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 [3], [4]), 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 [3]. 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. Some examples of work orders are described in Appendix A. The work order abstraction is closely related to the storage management in Quickstep, which we describe in Appendix B.

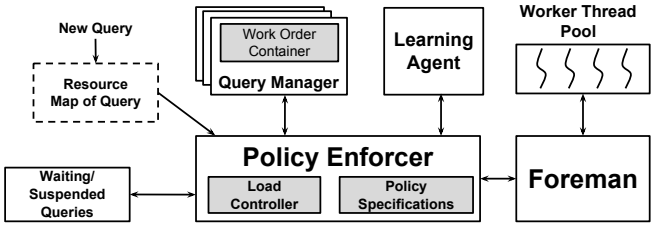


Fig. 1. Overview of the scheduler

III. Design of the Scheduler

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

A. Scheduler Architecture Overview

Figure 1 shows the architecture of the Quickstep scheduler. We begin by describing the *Query Manager*, that co-ordinates 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 II, 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 Appendix C) to determine when to activate nodes in the DAG.

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**: Provide work orders of the managed query for dispatching them for execution. **PE→QM**: Upon every work order completion, 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 III-B.

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

of a *Resource Map* to the Load Controller. A sample resource map is presented in Appendix D.

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 IV-D.

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.

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 Appendix E).

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 Appendix F.

B. 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 IV, 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 the 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.

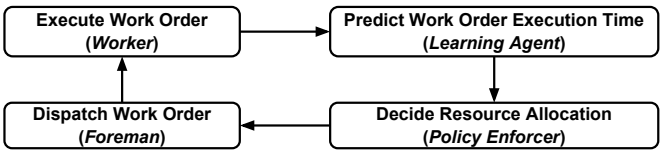


Fig. 2. Interactions among scheduler components

C. 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 IV 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 Appendix G.

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.

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 2 shows the Learning Agent's interaction with the 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 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

[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.

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 I
INTERPRETATIONS OF THE SAMPLE POLICIES IMPLEMENTED IN QUICKSTEP

requirements of the model. More details about our use of linear regression is described in Appendix H.

The problem of estimating the query execution time is well-studied, but requires complex methods [8]–[11]. The Learning Agent does not require such methods. However, it can combine estimates from other methods with its simple learning-based estimates.

IV. POLICY DERIVATIONS AND LOAD CONTROLLER IMPLEMENTATION

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

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 IV-D, we describe the load control mechanisms that are implemented in Quickstep. The load controller takes admission control and query suspension decisions based on the memory resource.

The scheduling policies, described below in Sections IV-A, IV-B, and IV-C 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 I. 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 II for notations).

A. Fair Policy Implementation

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

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

TABLE II
DESCRIPTION OF NOTATIONS

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

Notice that $\forall i, j$ such that $1 \leq i, j \leq k$, $pb_i/pb_j = t_j/t_i$.

If $t_i > t_j$, it means that the work orders for query q_i take longer time to execute than the work orders for query q_j. Thus, in a given time interval, fewer work orders of q_i must be scheduled as compared to the query q_j.

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

B. 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 to all the active queries in that priority class.

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

C. Proportional Priority (PP) Implementation

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

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

We now estimate the number of work orders of priority class PV_i that can be executed in its allotted time. For this estimation, 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 priority value PV_i. Therefore, assuming m queries in a given class and the individual estimates of work order execution times for queries with priority PV_i are $t_1, t_2, \ldots, t_m$,

then the predicted work order execution time for the class is $w_{PV_i} = \sum_{j=1}^{m} t_j/m$. Therefore the estimated number of work orders executed for priority class PV_i is $n_{PV_i} = t_{PV_i}/w_{PV_i}$.

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

Next we take a look at the load control mechanism.

D. Load Control Mechanism

As described earlier, the load control mechanism in Quickstep is designed to ensure the availability of memory resource to the queries in the system. This 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 III, 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 waitlists the new query and waits 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 waitlists the query. The waitlisted 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.

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 III
EVALUATION PLATFORM

V. 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.

Our evaluation platform is described in Table III.

A. Quickstep Specifications

We now describe Quickstep's configuration parameters that are used in the experiments. All 40 threads in the system are used as worker threads. The buffer pool is configured with 80% of the available system memory (126 GB). Memory for storage blocks, temporary tables, and hash tables is allocated from the buffer pool. The block sizes 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.

B. Experimental Workload

For the evaluation, we use the Star Schema Benchmark (SSB) [12]. We justify the choice of SSB for our evaluation in Appendix J. We use two variants of SSB SF 100 dataset, namely uniform and skewed. The skew is present in the *lo_quantity* column of *lineorder* table, as described by Seelam et al. [13]. 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].

C. Evaluation of Policies

In this section, we evaluate various policies 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.

1) Fair: In this experiment, we execute all 13 queries from the SSB concurrently using the fair policy. As described in Section IV-A, the policy specification implies a fair sharing of CPU resources among concurrent queries. The CPU utilization of the queries is depicted in Figure 3. As we can see, the CPU

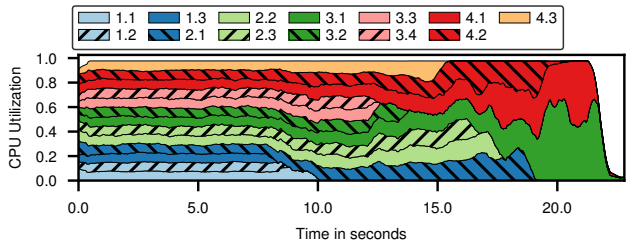


Fig. 3. CPU utilization of queries in fair policy

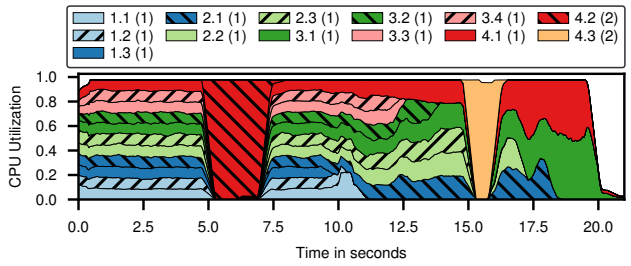


Fig. 4. CPU utilization in HPF policy. Note that $a.b(N)$ denotes a SSB query $a.b$ with priority N

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 15 seconds marks) when a query finishes its execution. This elasticity behavior allows Quickstep to fully utilize the CPU resources at *all* times.

2) HPF: Now we validate if the implementation matches the specification of HPF (Highest Priority First, cf. Section IV-B) policy.

All queries have the same priority value (1) except Q4.2 and Q4.3 which have a higher priority value (2). The execution begins with 11 queries having the same priority value. We inject Q4.2 in the system at around 5 seconds and Q4.3 at around 15 seconds. Figure 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 present the evaluation of Proportional Priority policy in Appendix K.

D. 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 5 shows the results of this experiment.

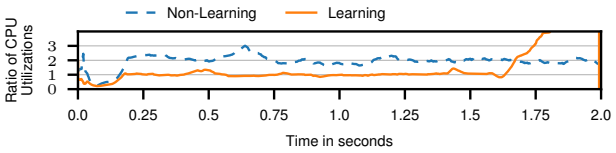


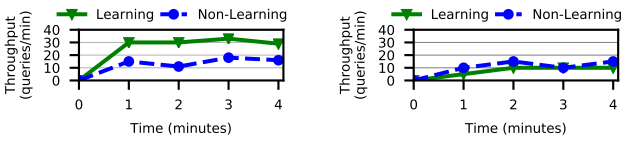
Fig. 5. Ratio of CPU utilizations $\frac{Q4.1}{Q1.1}$ for learning and non-learning implementations

Observe in Figure 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 10 in Appendix G). 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.

E. Impact of Learning on Performance

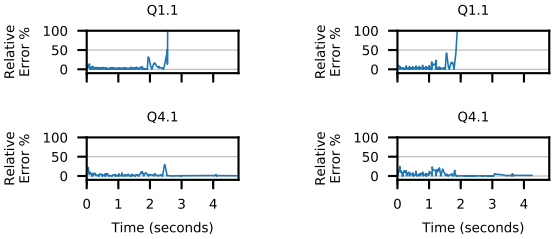
Here we analyze the impact of the learning-based approach on the performance of queries. We use two query streams, one for $Q1.1$ and another for $Q4.1$. As one instance of $Q4.1$ finishes execution, another instance of $Q4.1$ enters the system (likewise for $Q1.1$). We compare the throughput for both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

Figure 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 5), in the non-learning implementation, $Q1.1$ which has shorter work orders, gets 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).



(a) $Q1.1$ (b) $Q4.1$

Fig. 6. Impact of learning on the throughput



(a) Skewed dataset (b) Uniform dataset

Fig. 7. Comparison of predicted and observed time per work order

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.

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

G. Load Control

In this experiment, we examine Quickstep's load control capabilities. 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

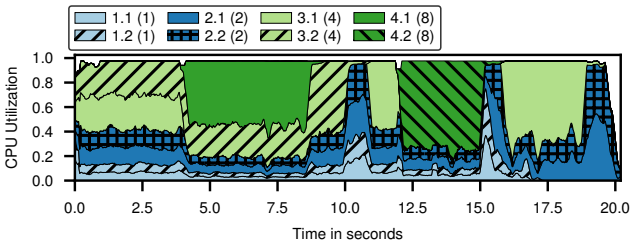


Fig. 8. Load control: An SSB query $a.b$ with priority N is denoted as $a.b(N)$

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 8 shows the CPU utilization of the queries.

The execution begins with all the 8 queries. At around 4 seconds, a high 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 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.

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.

VI. Related Work

We now describe the related work on scheduling in database systems, operating systems (OS), and networks.

The *work orders* abstraction is similar to other abstractions like *morsels* in Hyper [4] and the *segment-based parallelism* [5]. Such abstractions provide a means to achieve high intra-query, intra-operator data parallelism. Hyper [4] 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 [5]), cache-conscious task scheduling.

The elastic pipelining implementation [5] uses a *scalability vector* to vary the degree of parallelism of segments of the

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

There is related work [14] 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 [15]–[20] 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 [21], [22] 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 [8], [9], [25] includes analytical model 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 [26] 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. execution time

Quickstep's scheduler design aligns to the idea that scheduling must be transparent, fine-grained, and most importantly the need to separate mechanisms from policies [6] - a common theme found in the OS literature.

Deficit Round Robin (DRR) [27] 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.

VII. 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.

References

[1] V. R. Narasayya, I. Menache *et al.*, "Sharing buffer pool memory in multi-tenant relational database-as-a-service," *PVLDB*, 2015.

[2] V. R. Narasayya, S. Das *et al.*, "SQLVM: performance isolation in multi-tenant relational database-as-a-service," in *CIDR*, 2013.

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

[4] V. Leis, P. A. Boncz *et al.*, "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age," in *SIGMOD*, 2014.

[5] L. Wang, M. Zhou *et al.*, "Elastic pipelining in an in-memory database cluster," in *SIGMOD*, 2016.

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

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

[8] J. Duggan, U. Çetintemel *et al.*, "Performance prediction for concurrent database workloads," in *SIGMOD*, 2011.

[9] W. Wu, Y. Chi *et al.*, "Towards predicting query execution time for concurrent and dynamic database workloads," *PVLDB*, 2013.

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

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

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

[13] S. Seelam, P. Tuma *et al.*, Eds., *ICPE*. ACM, 2013.

[14] C. Gupta, A. Mehta *et al.*, "Fair, effective, efficient and differentiated scheduling in an enterprise data warehouse," in *EDBT*, 2009.

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

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

[17] "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.

[18] "Teradata workload management," http://www.teradata.com/Teradata_Workload_Management.

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

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

[21] S. Krompass, U. Dayal *et al.*, "Dynamic workload management for very large data warehouses: Juggling feathers and bowling balls," in *VLDB*, 2007.

[22] S. Krompass, D. Gmach *et al.*, "Quality of service enabled database applications," in *ICSOC*, 2006.

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

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

[25] W. Wu, X. Wu *et al.*, "Uncertainty aware query execution time prediction," *PVLDB*, 2014.

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

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

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

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

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

APPENDIX A
WORK ORDERS

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

Consider the following full table scan query to illustrate the work order concept:

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

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

- Relation: `Employee`, attribute: `name`
- Predicate: `city='San Diego'`
- The unique ID of an input block from the `Employee` table

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

APPENDIX B
STORAGE MANAGEMENT IN QUICKSTEP

Data organization in the Quickstep storage manager holds the key to intra-query parallelism [3]. Data in a relation are organized in the form of blocks. Each block holds a collection of tuples from a single table. A unique aspect of the storage organization in Quickstep is that blocks are considered to be independent and self-contained mini-databases. Thus, when creating an index, instead of creating a global index with "pointers" to the tuples in blocks, the index fragments are stored within the blocks. Each block is internally organized into *sub-blocks*. There is one *tuple storage sub-block*, which could be in a row store or a column store format. In addition, each block has one sub-block for each index created on the table. CSB+-tree [28] and BitWeaving [29] indices are currently supported. The blocks are free to self-organize themselves and thus a given table may have blocks in different formats. For example, new blocks in a table may be in a row store format, while older blocks may be in a column store format.

This storage block design, as articulated earlier in [3] enables the query execution to be broken down into a set of independent tasks on each block. This is a crucial aspect that we leverage in the design of our scheduler.

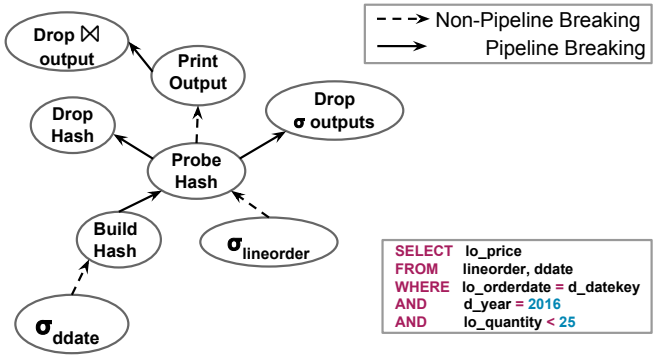


Fig. 9. A join query and its DAG

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

APPENDIX C
DAG TRAVERSAL ALGORITHM

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

Consider a sample join query and its DAG showed in Figure 9. The solid arrows in the DAG correspond to "blocking" dependencies, and the dashed arrows indicate pipelineable/non-blocking dependencies. To execute this query we need to select tuples from the `ddate` table, stream them to a hash table, which can then be probed by tuples that are created by the selection operator on the `lineorder` table. The output of the probe hash operation can be sent to the print operator, which displays the result. Note that the "drop hash" operator is used to drop the hash table, but only after the "probe hash" operation is complete. Similarly, the other drop operators indicate when intermediate data can be deleted.

The Query Manager uses a DAG Traversal algorithm (cf. Algorithm 1) to process the DAG, which essentially is an iterative graph traversal method. The algorithm simply finds nodes in the DAG that have all their dependencies met, and marks such nodes as "active" (line 13). Work orders are requested and scheduled for all active nodes (line 15), and

Algorithm 1 DAG Traversal

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

the completion of work orders is monitored. Operators are stateful and they produce work orders when they have the necessary data. The work order generation stops (line 16) when the operators no longer have any input to produce additional work orders. When no more work orders can be generated for a node, that node is marked as "completed" (line 17). When a node is marked as completed, all outgoing blocking edges (the solid lines in Figure 9) are "activated" (line 19). Pipelining is achieved as all non-blocking edges (dotted lines in Figure 9) are marked as active upfront (line 2). The query is deemed as completed when all nodes are marked as "completed."

Appendix D
Resource Map Discussion

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

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

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

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

Appendix E
Pipelining in Quickstep

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

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

Appendix F
Thread Model

Quickstep currently runs as a single server process, with mutiple user-space threads. There are two kinds of threads. There is one *Scheduler* thread, and a pool of *Worker* threads. All the components in Figure 1 except the worker thread pool run in the scheduler thread. In the current implementation, all threads are spun upfront when the database server process is instantiated, and stay alive until the server process terminates.

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

Every worker thread receives a work order from the Foreman, executes it and then waits for the next work order. In order to minimize worker's idle time, typically each worker is issued multiple work orders at any given time. Thread-safe queues are used to communicate between the threads. The communication happens through light-weight messages from the sender to the receiver thread, which is internally implemented as placing a message object on the receiver's queue. A receiver reads messages from its queue. A thread (and its queue) is uniquely identified by its thread ID.

The thread communication infrastructure also implements additional features like the ability to query the lengths of any queue in the system, and cancellation of an unread message. For simplicity, we omit discussion of these aspects.

Appendix G
Motivation for the Learning Agent Module

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

We perform an experiment, where the goal is to analyze the patterns in work order execution times of two queries.

The dataset used for the experiment comes from the Star Schema Benchmark (SSB) [12] at a scale factor of 100 (c.f. Section V-B for benchmark details). We pick two SSB queries $Q1.1$ and $Q4.1$, and execute them on a machine with 40 CPU cores. $Q1.1$ has a single join operation and $Q4.1$ has four join operations. Figure 10 shows the observed average time per work order for both queries. We now describe the trends in time per work order for the queries.

We can observe $Q1.1$'s execution pattern denoted by the dashed line in Figure 10. The time per work order remains fairly stable (barring some intermittent fluctuations) from the beginning until 1.8 s. This phase corresponds to the selection operation in $Q1.1$ which evaluates predicates on the *lineorder* (fact) table. A small bump in time per work order can be observed at the 1.8 s mark, when the probe phase of $Q1.1$ begins and continues until 2 s. Towards the end of the execution of $Q1.1$, (2.2 s) there is a spike in time per work order when the query enters the aggregation phase. The output of the hash join is fed to the aggregation operation. The results of aggregation are stored in per-thread private hash tables, which are later merged to produce the final output.

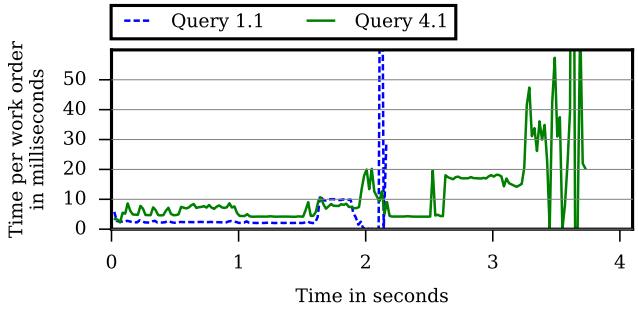


Fig. 10. Time per work order for Q1.1 and Q4.1

Now we analyze the execution pattern for $Q4.1$ which has 4 join operations. This query is more complex than $Q1.1$.Therefore, the execution pattern of $Q4.1$ exhibits more phases, with different times per work order as compared to $Q1.1$. Various small phases before the 0.5 s mark correspond to the selection predicates that are applied to the dimension tables. (Note that in $Q4.1$ there is no selection filter on the *lineorder* table). The selections on dimension tables get executed quickly. The longer phases denote the different probe hash table operations in the query. Towards the end, similar to $Q1.1$, there is a spike in the execution time per work order which correspond to the aggregation phase.

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

Appendix H
Usage of Linear Regression in Learning Agent

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

Appendix I
Resource Choices for Policy Implementations and Load Controller Implementations

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

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

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

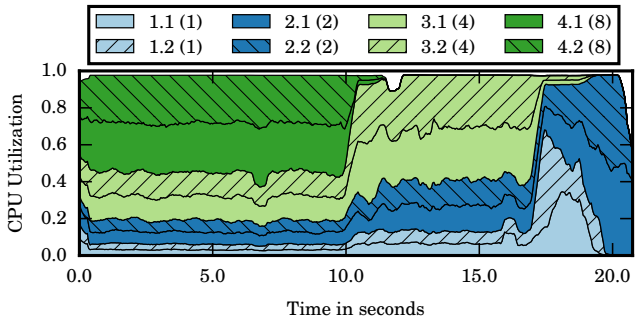


Fig. 11. CPU allocation for proportional priority policy. Note that $a.b(N)$ denotes a SSB query $a.b$ with priority N

APPENDIX J
APPLICABILITY OF SSB FOR OUR EVALUATION

The SSB is based on the TPC-H benchmark, and is designed to measure the query performance when the data warehouse uses the popular Kimball [30] approach. At a scale factor of X, the benchmark corresponds to about X GB of data in the corresponding TPC-H warehouse. The SSB benchmark has 13 queries, divided in four categories. Each query is identified as **qX.Y**, where X is the class and Y is the query number within the class. There are four query classes, i.e. $1 \leq X \leq 4$. The first and second classes have three queries each, the third class has four queries, and the fourth class has three queries. The queries in each category are similar with respect to aspects such as the number of joins in the query, the relations being joined, the filter and aggregation attributes. The grouping of queries in various classes makes this benchmark suitable for our experiments, as it provides a way of assigning priorities to the queries based on their class.

APPENDIX K
EVALUATION OF PROPORTIONAL PRIORITY POLICY

Now we examine the scheduler's behavior to the proportional priority policy (cf. Section IV-C, note that higher priority integer implies higher importance).

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

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

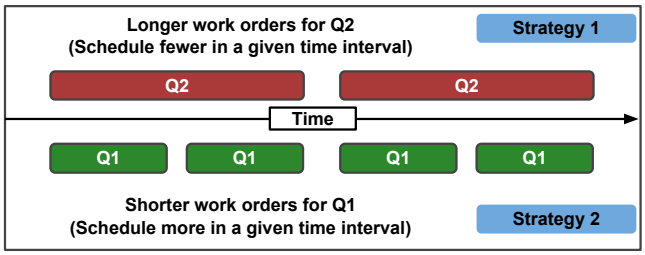


Fig. 12. Scheduling queries having different work order execution times for the fair policy. The solid boxes with Q_i depicts the lifetime of a work order from the Query Q_i.

APPENDIX L
POLICY ENFORCER

The Policy Enforcer applies a high level policy for resource allocation among concurrent queries. It uses a probabilistic-framework to select work orders from a pool of work orders belonging to different concurrent queries for scheduling. The Policy Enforcer assigns a probability to each active query, which indicates the likelihood of a work order from that query getting scheduled for execution in the near future. The probability-based work order selection strategy brings powerful control to the scheduler through a single parameter – i.e. by controlling the probability setting, the scheduler can control the resource sharing among concurrent queries.

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

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

Let us assume that as per the policy specifications, in a given time interval, the CPU resources should be shared equally. Suppose that work orders for q_1 take less time to execute than work orders for q_2, as shown in Figure 12. As the Policy Enforcer aims to allocate equal share of the CPU to q_1 and q_2, a simple strategy can be to schedule proportionally more work orders of q_1 than those of q_2, in a given time window. The number of scheduled work orders is inversely related to the work order execution time. This proportion can be determined by the probabilities pb_1 and pb_2 for queries q_1 and q_2, respectively. The probability pb_i is the likelihood of the scheduler scheduling next work order from query i. The probability is assigned by the Policy Enforcer to each active query in the system. Note that, $pb_1 > pb_2$ and $pb_1 + pb_2 = 1$.

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

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

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

There could be many reasons for categorizing queries in classes, including the need to associate some form of urgency (e.g. interactive vs batch queries), or marking the importance of the query source (e.g. the position of the query submitter in an organizational hierarchy). In addition, the resource allocations across different classes can also be chosen based on various scales, such as linear or exponential scale allocations based on the class number. An attractive feature of the Policy Enforcer is that it can be easily configured for use in a variety of ways. Under the covers, the Policy Enforcer simply maps each class to a collective class probability value, and then maps each query in each class to another probability. Once these probabilities are calculated, the remaining mechanisms simply use them to appropriately allocate resources to achieve the desired policy goal.