LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems

Ibrahim Sabek
sabek@mit.edu
Massachusetts Institute of Technology
United States

Tenzin Samten Ukyab*
ukyab@berkeley.edu
University of California, Berkeley
United States

Tim Kraska
kraska@mit.edu
Massachusetts Institute of Technology
United States

ABSTRACT

Query scheduling is a crucial task for analytical database systems that can greatly affect the query latency. However, existing scheduling approaches are based on heuristics and not optimal. A recent trial proposed to use reinforcement learning for automatically learning end-to-end scheduling policies. However, such trial was not capable of considering the database-specific characteristics (e.g., operator types, pipelining), and hence becomes not efficient for analytical database systems. In this paper, we try to fill this gap and introduce *LSched* (Learned Scheduler), a fully learned workload-aware query scheduler for in-memory analytical database systems. *LSched* provides an efficient *inter-query* and *intra-query* scheduling for dynamic analytical workloads (i.e., different queries can arrive/depart at any time). We integrated *LSched* with an efficient in-memory analytical database system, and evaluated it with TPCH, SSB, and JOB benchmarks. Our evaluation shows that *LSched* improves over the performance of existing state-of-the-art query schedulers and heuristic-based ones by at least 35% and 50% in both streaming and batching query workloads.

CCS CONCEPTS

• **Information systems → Database query processing**.

KEYWORDS

Databases, Query Scheduling and Execution, Machine Learning, Reinforcement Learning

ACM Reference Format:

Ibrahim Sabek, Tenzin Samten Ukyab, and Tim Kraska. 2022. LSched: A Workload-Aware Learned Query Scheduler for Analytical Database Systems. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22), June 12–17, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3514221.3526158

1 INTRODUCTION

Scheduling of tasks in an analytical query processing system can have a profound impact on the query latency and query throughput [34, 43, 44, 58]. In modern database systems, a query is typically composed of one or more sub-tasks, i.e., operators, and a query scheduler is responsible of finding an execution order for all tasks from all currently running queries in order to optimize some goal (e.g., minimizing latency [57] or minimizing the schedule makespan [53]). The optimization goal then determines how resources (e.g., CPU, memory, threads) are shared among concurrent queries and how various parts of the system function together.

*Work done when author was at MIT.

This work is licensed under a Creative Commons Attribution International 4.0 License.

SIGMOD '22, June 12–17, 2022, Philadelphia, PA, USA.

© 2022 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-9249-5/22/06..
https://doi.org/10.1145/3514221.3526158

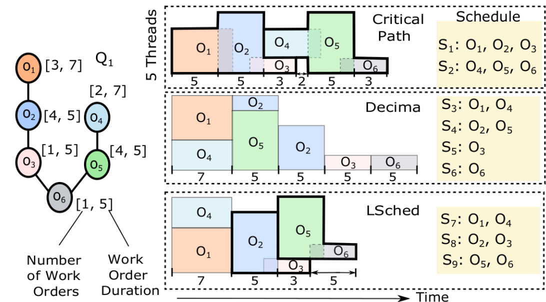

Figure 1: Example on the scheduling quality of different three query schedulers: Critical path, Decima and *LSched*.

However, for the most common optimization goals (e.g., overall latency), it is impossible or computationally impractical to derive an optimal algorithm. For example, scheduling N queries on M cores such that the makespan of the schedule is minimum, is NP-complete [53]. As a result, commonly used scheduling algorithms rely on specific heuristics including FIFO, fair scheduling [8, 13], short job first (SJF), highest priority first (HPF), packing [10], and approximation algorithms [47].

Although these standard techniques are easy-to-implement and transparent (i.e., decisions are taken based on guidelines and hence are understandable), they obviously miss major performance optimization opportunities when scheduling analytical workloads. Heuristics tend to make simplifying assumptions and it is notoriously hard to extend them to consider more complex system behavior. For example, it would be extremely hard to develop a good scheduling heuristic, which considers not only the hardware configuration, but also the current state of the system (e.g., what data is cached), the query plan structure, the type of operations (e.g., performance of hash vs index-joins), pipelining opportunities, as well as, the unpredictability because of data dependent factors (e.g., the selectivity of a predicate).

To overcome the complexity challenge of building a workload-specific scheduling policy, a recent attempt, namely Decima [34], proposed to use reinforcement learning (RL) to *automatically* build a specific end-to-end scheduling policy for the input workload. While the work laid the foundation to learn scheduling policies, it turned out that the neural network design and RL solutions of Decima

are not capable of considering the database-specific characteristics, such as the query structure, different operators (e.g., aggregation vs join), and the pipelining opportunities. For example, Decima assumes that any query is a DAG of tasks, where each task is a black box and can not be scheduled unless all its parent tasks are completed (i.e., no support for pipelining). However, in the case of relational queries, it is crucial to consider each task as a white box and optimize the scheduling policy according to its details and relationships with other tasks. Figure 1 shows an example on how using the proper degree of pipelining could affect the scheduling quality. In this example, a query Q_1, which consists of 5 select and 1 join operators (i.e., tasks), is scheduled using three different schedulers over 5 threads. These schedulers are (1) typical critical path pipelining [19] (a heuristic that runs the pipeline containing more aggregate work first), (2) Decima, and (3) our proposed learned scheduler. Q_1 has two sets of operators that can be pipelined. The first set has o_1, o_2, and o_3, while the second one has o_4, o_5 and o_6. The figure shows the scheduling decisions (yellow rectangle) for the three schedulers. As seen, the critical path results in two scheduling decisions, where each decision performs one aggressive pipelining (highlighted in bold). Decima has a lack of pipelining support, yet, it still learns a good scheduling policy that packs tasks efficiently over threads. In contrast to both critical path (aggressive pipelining) and Decima (non-pipelining), our proposed scheduler provides the best scheduling (the total time is 20 compared to 23 and 27 for other schedules) as it learns proper pipelining ($o2$ and o_3 in S_8, and o_5 and o_6 in S_9) as well as efficient tasks packing.

Besides Decima, the database literature has some trials to leverage machine learning (ML) for scheduling queries in analytical workloads. However, such trials tend to use ML to predict the query or task latency to make better decisions, rather than trying to learn an entire scheduling policy. For example, Quickstep [43] uses linear regression to predict the execution times of the future work orders for a given query based on its execution history. Such prediction is used to control the resource allocation decisions coming from the running heuristic scheduling policy (e.g., HPF). More recently, SelfTune [58] employs a constrained optimization technique to tune the hyper-parameters of its fixed scheduling policy for the input workload. In both works, the scheduling policy is still based on heuristics, and is not specifically built from scratch for the input workload. Another work [64] uses RL to automatically learn the order of executing different queries that maximizes the utilization of buffer pool items (i.e., improving the hits ratio). However, optimizing the schedule according to the buffer pool hits only, without considering other scheduling factors (e.g., parallelism degree for each query) and the current state of execution environment, does not necessarily lead to a better multi-query execution plan.

In this paper, we introduce *LSched* (Learned Scheduler), a fully RL-based learned workload-aware query scheduler for in-memory analytical database systems. *LSched* provides an efficient *inter-query* and *intra-query* scheduling for dynamic analytical workloads (i.e., different queries can arrive and depart at any time). Given a set of executed query plans from the past (e.g., workload logs), *LSched* automatically learns how to make the following decisions at each scheduling event: (1) which subset of queries to execute, (2) which subset of operators from these queries to execute, and (3) how much resources (e.g., threads) to be assigned to these operators. *LSched* optimizes these scheduling decisions for the specific input workload and according to a user-defined high-level policy objective (e.g., minimizing the user-perceived latency).

LSched works as follows: it takes a query to be scheduled as an input (queries can arrive in batches or streaming fashion) and generates a query plan (i.e., DAG of operators) for it using a typical analytical DBMS (e.g., Quickstep [43]). Then, this query plan is fed to the *LSched*'s scheduling agent, which consists of a *query encoder* and a *scheduling predictor*, to predict a sequence of scheduling decisions for executing this query along with other queries that already exist in the system at the moment. Each scheduling decision is triggered by a scheduling event (e.g., arrival of a new query, or finishing the execution of an operator), and is designed to minimize the expected latency of all existing queries after running the whole sequence of scheduling decisions (i.e., long-term execution plan). As *LSched* discovers better scheduling decisions over time, *LSched*'s scheduling predictor improves and results in better execution plans. This improvement procedure continues until the predictor converges.

In summary, *LSched* has the following contributions that overcome the limitations of Decima and outline how an end-to-end query scheduling is learned in the database context:

(1) Employing efficient physical plan features. The accuracy of RL-based query scheduling mainly depends on extracting representative features that capture the relational query execution environment well. Therefore, unlike Decima [34] which only extracts black-box features (e.g., number of tasks), *LSched* extracts a representative set of white-box features from the physical plans of the running queries (e.g., fine-grained level work orders) as well as their interaction with the execution environment (e.g., status of running execution threads).

(2) Efficient and accurate query encoding. Query encoding is a crucial part in the learned scheduler as it has to digest large amount of information, coming from the extracted features, accurately and in an efficient manner. Therefore, unlike Decima [34] which uses graph convolution networks (GCN) [23], *LSched* proposes a novel *Query Encoder* that combines a customized tree convolution [38] technique with importance weighting mechanisms from graph attention networks (GAT) [56] to learn an efficient and accurate query encoding that suits the nature of our scheduling problem.

(3) Flexible scheduling decisions. *LSched* proposes a *Scheduling Predictor*, which is a deep neural network that efficiently leverages the *Query Encoder* output, to decide which relational operators from running queries to be scheduled next and how much resources (e.g., threads) are needed for them. In addition, the *Scheduling Predictor* can explicitly determine the effective degree of operators pipelining that should be used at each execution step (i.e., neither aggressively pipeline nor completely execute in a sequential mode). As far as we know, *automatically* controlling the operators pipelining was never introduced before by previous schedulers including Decima.

(4) Balance between average and tail latency. Unlike Decima which focuses only on minimizing average query time, *LSched* proposes a novel rewarding function that optimizes the scheduling policy to minimize both average and tail latency at the same time.

(5) Effective training time. *LSched* performs its training in an efficient manner. Moreover, unlike Decima which learns any new

scheduling policy completely from scratch, *LSched* adapts an efficient transfer learning technique [2] to reduce the expensive training cost and avoid the need to start from scratch.

LSched can be integrated with any in-memory block-based database systems (e.g., [18, 20, 43]). In this paper, as a prototype implementation, we integrated *LSched* with Quickstep [43] and evaluated it with TPCH, SSB and JOB benchmarks. Our evaluation shows that *LSched* outperforms both existing state-of-the-art query schedulers and heuristic-based ones. In particular, *LSched* can improve average query duration over Decima [34] by 35% and 50% in both streaming and batching query workloads, respectively, and improves over Quickstep and SelfTune [58] by at least 60% and 55% for the same types of workloads.

2 PRELIMINARIES: QUICKSTEP

In this section, we give an overview of Quickstep [43], an efficient in-memory analytical database system, which we integrate our learned query scheduler with.

Block-based Storage. Quickstep manages its table storage as a set of *blocks*. Each block is considered as a mini self-contained database, where it consists of 1) sub-blocks of data, for both raw and indexed tuples, and 2) a metadata header to describe the block's contents. Blocks can be of different sizes, and support different data layouts, such as row and column store formats.

Work Order-based Operators. Quickstep represents each relational operator in the query plan (i.e., DAG) as a set of *work orders* (similar to morsels in HyPer [29]), which are generated at the block level. Specifically, each work order is defined over one block, and contains a set of information about the operator's inputs, parameters and pipelining status (i.e., whether the execution of this operator is blocked on its parent operator or not). For example, when applying a *select* operator on an input relation, Quickstep generates as many work orders as there are blocks in the input relation, and encodes in each work order, 1) a reference to its input relation, 2) a filtering predicate, 3) a projection list of attributes (or expressions), and 4) a reference to a particular input block. Quickstep supports work order implementations for 29 operators, including variations of complex operators such as *aggregate* and *union*.

Work Order-based Scheduling and Execution. For each query, the Quickstep scheduler selects operators that are ready to be scheduled (a.k.a active nodes) using a DAG traversal-based algorithm, then fetches their work orders and executes them in parallel on available threads. Operators, such as *aggregate* and *ProbeHash*, are not activated till all their blocking dependencies complete execution. When work orders are completed, the execution threads send completion messages to the scheduler, which include *execution statistics* that can be used to analyze the query execution behavior. Quickstep employs heuristics-based scheduling policies (e.g., fair scheduling) to select among the active nodes of different queries. Note that Quickstep supports both streaming and batched queries execution, yet, it neither supports multi-query optimization (i.e., co-optimizing the plans of multiple queries together) nor work sharing (e.g., operators sharing).

Why Selecting Quickstep? There are three main reasons that motivated us to integrate *LSched* with Quickstep in our prototype implementation. First, Quickstep provides an access to a rich set of fine-grained information (i.e., at the work order level) about the query execution plans. Such information allows *LSched* to be trained with more accurate and specific query execution features, rather than the high-level ones coming from typical query optimizers. Second, the Quickstep scheduler design separates the choice of policies from the execution mechanisms, which significantly facilitates the integration with *LSched*. Third, Quickstep is an open-source project [46], and is easy to modify its core engine.

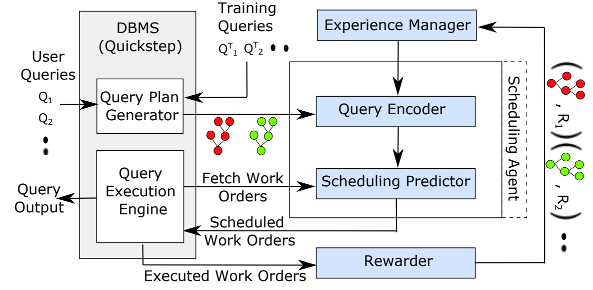

Figure 2: *LSched* system overview.

3 SCHEDULER OVERVIEW

In this section, we give an overview of *LSched* and its workflow. Figure 2 shows the different components of *LSched* (highlighted in blue) and how a typical database system interacts with them.

***LSched* in Action.** *Scheduling Agent* is the main component of *LSched* and is responsible for predicting the scheduling decisions. It has two main modules, namely *Query Encoder* and *Scheduling Predictor*. When a user query arrives, the DBMS first generates a DAG plan of physical operators to execute this query. Then, this plan along with other queries' plans and the status of execution environment (e.g., available threads) are submitted to the *Query Encoder* to obtain an efficient *embedding* representation (i.e., encoding) that will be used in the scheduling prediction later. The *Query Encoder* employs a novel, yet scalable, approach that combines both tree convolution [38] and graph attention networks (GAT) [56] to embed the huge information related to the current queries and execution environment (Section 4). Once the encoding is done, the embeddings are passed to the *Scheduling Predictor*, which is a deep neural network, to predict the final scheduling decisions (Section 5). Specifically, it should predict (1) which operators (and whether their pipelining, if any, is preserved or not) from which queries to be scheduled, and 2) how many threads should be assigned to each running query. Typically, executing one or more concurrent queries requires multiple iterations of scheduling decisions. These iterations occur on certain scheduling events, such as the arrival of a new query or completing the execution of one or more of the currently schedule operators (i.e., some execution threads are free). In each iteration, the DBMS query execution engine invokes the *Scheduling Predictor* to schedule new operators and assign resources (e.g., threads) to them. Then, the predictor answers with its scheduling decision that will be translated into a set of work orders to be executed by the query execution engine, which is Quickstep in our case. Whenever a scheduling decision is completely executed, its effect is rewarded and sent back to the agent to improve its performance accordingly.

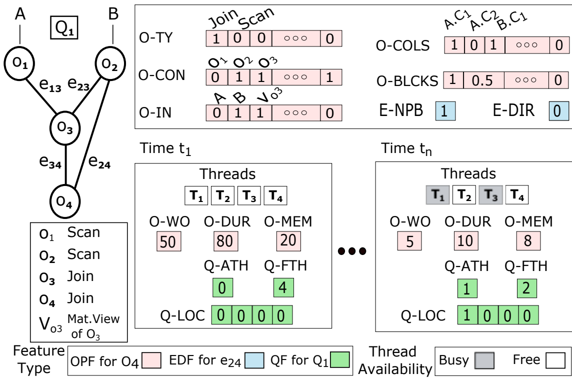

Figure 3: Example on the generated features in *LSched*.

***LSched* Training.** *LSched* uses RL to train its *Scheduling Predictor* through sample *Training Queries* that can be gathered from the workload logs. In each training episode, *LSched* attempts to schedule a query workload (coming in both streaming and batching modes) using the *Scheduling Predictor*, observes the overall execution latency, and provides the predictor with a reward on its decisions (Section 6). The reward reflects how much the predictor, with its current parameters, achieves the high-level scheduling policy objective (e.g., minimizing the user-perceived latency). The RL algorithm gradually improves the efficiency of the *Scheduling Predictor* using these rewards. In the online mode, the completely executed scheduling decisions are also rewarded and used for self-correcting the predictor either on a query-by-query basis or at checkpoints (controlled by the user). All these reward experiences, from both training and online modes, are stored and managed by the *Experience Manager*.

Since *LSched* customizes the scheduling policies for each workload, it requires retraining its *Scheduling Predictor* from one workload to another. However, training the predictor from scratch is extremely expensive and unaffordable for DBMSs because each training episode requires executing a batch or a stream of queries and it is still cumbersome for users to execute thousands of training iterations before using the scheduler for a new workload. Moreover, some workloads may not have enough logs (i.e., training queries) at the beginning to bootstrap the scheduler. To alleviate this issue, we adapt an efficient transfer learning technique [2] to reduce the training cost and avoid the need to start from scratch (Section 6).

4 QUERY ENCODER

Query encoding in *LSched* is done on two steps. First, a set of features are collected from the physical plan of each query and its interaction with the execution environment (Section 4.1). Then, these features are used to generate efficient encoding within the query itself (Section 4.2) and across the whole queries (Section 4.3).

4.1 Physical Plan Features Extraction

The main objective of *LSched* is to capture the query execution environment while scheduling concurrent queries. To properly achieve that, *LSched* extracts a set of useful *features* (discussed in the rest of this subsection) from the *physical plans* of the running queries as well as *their interaction with the dynamic execution environment*. In our prototype of *LSched*, we use the query execution engine of Quickstep to extract these features. However, such features can be extracted while using any modern block-based database system (e.g., Quickstep[43], MonetDB [18], Hyrise [12], RocksDB [6], Umbra [40], HyPer [20]), that adapts work order/morsel-based implementation of its physical operators [40, 43].

There are three types of features in *LSched*: *Operator Features* (OPF), *Edge Features* (EDF), and *Query Features* (QF).

Operator Features (OPF). *LSched* generates the following eight features at each *operator* in the physical query plan.

Operator Type (O-TY): A vector that encodes the operator type (e.g., select, join), stored in a "1-Hot" representation.

Operator Connectivity (O-CON): An adjacency list showing the operator connectivity with other operators in the query plan.

Input Relations (O-IN): A vector that encodes the input relations of the operator (both base and intermediate relations), stored in a "1-Hot" representation.

Columns (O-COLS): A vector that encodes the used columns (i.e., attributes) in the operator from the input relations, stored in a "1-Hot" representation (e.g., in case of the *select* operator, the column used in the filtering predicate).

Blocks (O-BLCKS): A vector that encodes the data blocks, which are planned to be processed in the work orders of the operator. Such information about blocks for each operator are available from the query optimizer in block-based database systems (e.g., Quickstep[43], MonetDB [18], Hyrise[12], RocksDB [6]). Therefore, no need to reference the blocks during scheduling. Typically, the number of blocks can be very high and different from one relation to another, and representing the blocks using a bitmap of 1-Hot representation is extremely inefficient. Therefore, inspired by [64], *LSched* uses a simple moving average over the number of blocks to reduce the O-BLCKS feature size. Given d as the new feature array (i.e., downsized array) and b as the original array, an entry d_j at index j in the new array d can be calculated as:

$$d_j = \frac{|d|}{|b|} \sum_{k=j\times\frac{|b|}{|d|}}^{(j+1)\times\frac{|b|}{|d|}} b_k \tag{1}$$

For example, assume that an original blocks bitmap $b = \{1, 1, 0, 1, 1, 0\}$ will be reduced to a feature array d of size 3. The value of $d[0]$ will be $\frac{3}{6}(1+1+0) = 1$, and analogously $d[1]$ and $d[2]$ will be 1 and 0.5.

Remaining Work Orders (O-WO): A single value that encodes the remaining amount of work orders on the operator when the scheduler is invoked at time t. This number of remaining work orders is calculated by subtracting the completed work orders so far from the original number of work orders determined by the query optimizer (note that this number will be an estimate if the operator input is generated in a non-pipeline breaking mode).

Estimated Duration (O-DUR): A single value that encodes the total estimate of duration (i.e., expected running time) of the remaining work orders on the operator when the scheduler is invoked at time t. This total estimate of duration is calculated on two steps. First, *LSched* estimates the duration of a single *next* work order using a linear regression over the execution times of previously

completed work orders on this operator[1]. Then, this duration estimate is multiplied by the number of remaining work orders.

Estimated Memory (O-MEM): A single value that encodes the total estimate of memory usage of the remaining work orders on the operator when the scheduler is invoked at time t. It is calculated similar to O-DUR, where the linear regression is applied on memory usages instead of execution times.

Edge Features (EDF). *LSched* generates the following two features at each *edge* in the physical query plan.

Non-pipeline Breaking Status (E-NPB): A single value, either 0 or 1, for each edge that encodes whether the two operators connected through this edge support pipelining execution or not (i.e., value is 1 if non-pipeline breaking). For example, if the edge connects two *select* operators, then it is non-pipeline breaking because the child *select* operator can start execution before the parent *select* finishes. In contrast, if the edge is between *BuildHash* and *ProbeHash* operators, then it is pipeline breaking because *ProbeHash* will be blocked till *BuildHash* completes execution.

Pipeline Direction (E-DIR): A single value, either 0 or 1, for each edge that encodes the direction of pipelining execution (i.e., which operator is the source).

Query Features (QF). When the scheduler is invoked, *LSched* generates the following three features for each individual query.

Assigned Threads (Q-ATH): A single value that encodes the number of assigned execution threads to the query at the moment.

Free Threads (Q-FTH): A single value that encodes the number of available threads which can be assigned to the query at the moment.

Threads Locality (Q-LOC): A vector that encodes the locality status of each available thread for this query, stored in a "1-Hot" representation (i.e., whether this thread executed previous operators from this query or not).

4.1.1 Features Generation. The previously discussed features can be categorized into two main categories: *static* and *dynamic*. *Static* features include O-TY, O-CON, O-IN, O-COLS, O-BLCKS, E-NPB, and E-DIR. In this category, features are populated once at the beginning of the query execution (i.e., when the physical plan obtained) and remain unchanging till the query completes its execution. They are calculated based on the information in the headers of work orders in each physical plan operator. *Dynamic* features include O-WO, O-DUR, O-MEM, Q-ATH, Q-FTH, and Q-LOC. In this category, features are (re)calculated whenever there is a scheduling event, and based on the execution statistics that are reported by the execution monitor[2] (e.g., Quickstep's resources estimator [43], and RocksDB's statistics collector [6]). Such statistics are updated whenever the scheduled work orders complete their execution.

Note that *LSched* currently has no support for generating explicit features over multiple running queries. This is mainly because existing block-based database systems, which *LSched* is designed for, generate the physical plan for each query individually (i.e., no support for multi-query optimization nor shared execution yet). Therefore, *LSched* is forced to rely on such individual plans when generating features. That being said, although there are no specific features defined over multiple concurrent queries, the aforementioned *dynamic* features ensure that *LSched* captures the status of multi-query execution environment, because these features are estimated while other concurrent queries are running.

[1]We chose linear regression as it is computationally-efficient and provides accurate predictions. For efficiency, to predict duration D_{w_t} of work order w_t at time t, we fit a linear regression model only on durations of work orders within the last time window k (i.e., $D_{w_{t-k}}, D_{w_{t-k+1}}, ..., D_{w_{t-1}}$).

[2]All OPF, EDF and QF features are "not" specific to Quickstep and can be implemented in any block-based database system.

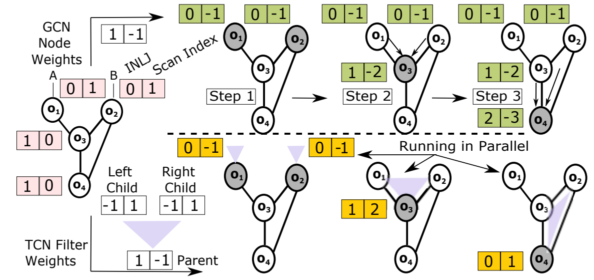

Figure 4: Example on detecting patterns in queries using (1) GCN with sequential message passing and (2) TCN.

Example. Figure 3 depicts an example on some of the generated features for a query Q_1, that has four different operators. It presents the OPF for one operator o_4, EDF for one edge e_{24}, and the QF for the whole query. The changed dynamic features O-WO, O-DUR, O-MEM, Q-ATH, Q-FTH, and Q-LOC are shown for two different scheduling times t_1 and t_n.

4.2 Single Query Encoding

At each scheduling step, the learned scheduler should leverage a large amount of features (Section 4.1) coming from hundreds of concurrent queries and a dynamic execution environment.

As shown in Decima [34], one straightforward solution to efficiently encode such large information is to use a modified variation of graph convolutional network (GCN) [1, 23] that supports convolution over directed acyclic graphs (DAGs). In traditional GCN, the information from the neighbours of each node in the graph is propagated simultaneously through message passing [23]. However, this is not suitable for capturing efficient patterns in scheduling and executing DAGs. In DAGs, the execution of nodes (which are operators in our case) has a hierarchical nature, and there are many local parent-children relations among nodes (for example, in our case, a *BuildHash* operator is typically followed by a *ProbeHash* one or a *MergeJoin* operator is applied on a sorted input). Therefore, Decima proposed to perform message passing in each convolution iteration through a set of *sequential steps*, where the message passing in each step s is restricted to be only from the children nodes in the previous step $s-1$ to their parents in the current step s. Although this solution can preserve the DAG structure, it still has two main limitations. The first limitation is that sequential message passing, *within the same convolution iteration*, results in *over-smoothing* problem [60]. In typical GCN, the message passing within one convolution iteration calculates the embedding at each node by fusing the information from its direct neighbours only, while the information from indirect neighbours (i.e., larger neighbourhoods) is fused over the *multiple* convolution iterations. Fusing indirect information helps in detecting complex relationships among nodes, yet, the exaggeration of doing that leads to high error propagation and values over-smoothing [60]. Sequential

message passing, as in Decima, fuses information from indirect neighbours *across multiple iterations* and *within the same iteration* as well, which leads to higher over-smoothing. For example, within the same iteration of sequential message passing, the embedding of operator o_s at step s is fused in its parent o_{s+1} at step $s+1$, and in turn o_{s+1} is fused in its parent o_{s+2} (i.e., o_s's grand parent) at step $s+2$. This means that o_s is already fused in o_{s+2} indirectly within the same iteration. The second limitation is that GCN propagates information among nodes with *same importance*. However, this might be inefficient in some query execution cases, where operators can have different impacts on each other [28].

To avoid the limitations of GCN, *LSched* proposes to perform query encoding through a combination of tree convolution [38] with graph attention networks (GAT) [56]. The success of using tree convolution in a recent work for learning query optimization [37] inspired us to customize a variation of tree convolution filters for capturing hierarchical scheduling and execution patterns (Section 4.2.1). To further tune the tree convolution output, we use GAT to allow each operator to aggregate information from its child operators and connected edges *while assigning them different importance* (Section 4.2.2).

4.2.1 *Customized Tree Convolution.*

Effective convolution methods should have a strong *inductive bias* [35]: the convolution kernel should be designed according to a specific intuition or understanding. For example, convolutional neural networks (CNN) [32] detects visual features from an image by sliding a kernel over the different image regions, which is very similar to what happens in the visual cortex system of humans. This makes CNN has a strong *spatial locality* inductive bias. Similarly, tree convolution [38] has been shown to have a strong inductive bias in extracting local parent-children patterns over tree structures (e.g., query plans) [37].

Typical tree convolution assumes a binary tree structure of features that need to be convoluted, where each node in the tree has a flat vector of features. Then, the tree convolution slides a set of shared filters (i.e., kernels) over each local parent-children triangle in the tree to apply the convolution operation and generate *embedding* vectors (i.e., new feature representations) for all nodes. This is similar to the convolution in CNN, but with triangle-like filters not square ones, and applied on trees not grids. Usually, multiple tree convolution layers are "stacked" to capture a wide range of local parent-children patterns.

TCN vs GCN Example. Figure 4 shows an example on how the convolution in TCN can detect interesting patterns over queries more efficiently than in sequential message passing GCN. In this example, we assume a simple query with two index-nested-loop join (INLJ) operators (o_3, o_4) and two index-scan operators (o_1, o_2), defined over two base relations A and B. In this query, each node has a simple OPF feature vector (check Section 4.1) that only has O-TY with two values for encoding whether the node is either INLJ, index-scan or not. We target to detect a pattern of having INLJ operation on top of one (or two) index-scan operation(s).

GCN Computation. The top portion of Figure 4 depicts how the first convolution layer of sequential message passing works over DAGs, where the embeddings of nodes are calculated sequentially on different steps. As a concrete example, the embedding of o_3 is calculated at step 2, after the embeddings of o_1 and o_2 being calculated at step 1. The embedding of $o_3 = ([1,-1]\odot[1,0])+[0,-1]+[0,-1] = [1,-2]$, where $\odot$ is the Hadamard product (i.e., element-wise product), $[1,-1]$ is the node's weight vector, $[1,0]$ is the current node's embedding (since this is the first convolution layer, the node's feature vector becomes the current node's embedding), and $[0,-1]$ is the previously calculated embeddings for o_1 and o_2. Note that node's weight vector $[1,-1]$ rewards (penalizes) the node if it is an INLJ (index-scan) operation.

TCN Computation. The bottom portion of Figure 4 shows how the first convolution layer of TCN works. In this example, a tree convolution filter consists of three weight vectors with $[1,-1]$ at the parent (similar to GCN node weights for fair comparison), and two weight vectors with $[-1,1]$ at each child (to capture whether any of the child nodes is an index-scan or not). Given a parent node n_p and its children n_l and n_r, the embedding at n_p is calculated by first applying the Hadamard product between the embedding of each one of the n_p, n_l and n_r nodes, coming from the previous convolution layer (since this is the first convolution layer, we use the node's feature vector as a previous embedding), and its corresponding weight vector from the filter. Then, the output products are added together to form the final embedding at n_p. As a concrete example, the embedding of $o_3 = ([1,-1]\odot[1,0])+([-1,1]\odot[0,1])+([-1,1]\odot[0,1]) = [1,2]$. Note that the embedding at any node n_p using TCN does not depend on the children embeddings that are calculated in the same convolution layer, and hence, unlike GCN, the embeddings at different nodes within the same layer can be parallelized and do not suffer from over-smoothing [60].

Quality Comparison. In this example, both the GCN node's weight and the TCN tree filter were designed to only reward the nodes that achieve the INLJ pattern, and punish others. Therefore, the higher non-negative embedding values at nodes that match the required pattern, the better convolution it is. By comparing the output embeddings obtained from GCN and TCN at all nodes in this example, we can see that TCN detects the pattern better. In particular, the TCN embeddings of o_3 and o_4 (the two nodes that achieve the INLJ pattern) are non-negative, where their counterparts in GCN have negative values. This is mainly because 1) the TCN filter employs different weight vectors that accurately capture the different parent-child relationships within the pattern, and 2) In sequential message passing, there is an over-smoothing happening at the nodes that achieve the pattern, which reduces their embedding values (e.g., the indirect fusion of o_1 in o_4 reduces the embedding values at o_4).

Edges Support. The existing triangle filters in typical tree convolution [37, 38] are designed to only process the nodes in the local parent-children triangle (e.g., Figure 4), and ignore the edge information (i.e., support features on nodes only). Therefore, it is not suitable to use these filters "as is" with *LSched*' features (Section 4.1), which include both *node* and *edge* features. To solve this issue, we provided a variation of tree convolution filter that takes into account the edge features as well. Given a tree convolution layer l, the modified tree convolution filter is defined, for any parent node p and its right m and left n child nodes, as follows:

$$x_p'^l = \sigma(w_p^l \odot x_p^l + w_{p,m}^l \odot e_{p,m}^l + w_m^l \odot x_m^l + w_{p,n}^l \odot e_{p,n}^l + w_n^l \odot x_n^l) \quad (2)$$

where x_p^l and $x_p'^l$ are the current and updated embeddings of parent node p, x_m^l and $e_{p,m}^l$ are the embedding vectors for the

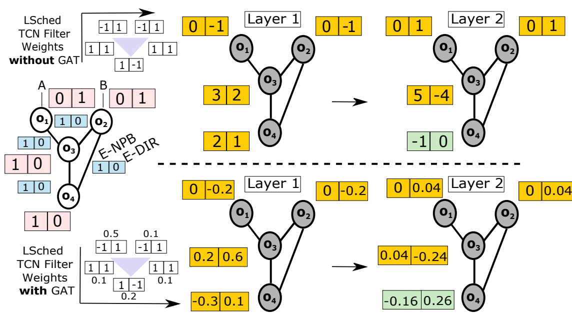

Figure 5: Example on the effect of using GAT scores in TCN.

right child node and its edge to the parent, x^l_n and $e^l_{p,n}$ are the embedding vectors for the left child node and its edge to the parent. Here, the filter consists of five weight vectors w^l_p, w^l_m, $w^l_{p,m}$, w^l_n, and $w^l_{p,n}$ corresponding to each of the embedding vectors, where a Hadamard product $\odot$ (i.e., element-wise product) is applied on each embedding-weight pair of vectors. σ is non-linear activation function (e.g., ReLU [9]).

The triangle filter in Equation 2 is defined over a parent and two child nodes only. We choose this as it is compatible with the physical query plans in our prototype implementation with Quickstep. However, it can be easily adapted for a higher number of children and edges by adding more weighted terms (in this case it will not be a triangle filter). In practice, a set of triangle filters (e.g., hundreds) are defined on different tree convolution layers. This significantly helps capture various hierarchical patterns with different characteristics. The weights of all filters are learned during the training of the scheduling neural networks (sections 5 and 6).

4.2.2 Graph Attention Networks (GAT). When stacking multiple tree convolution layers (i.e., the output of layer l is fed to layer $l+1$), the amount of information seen by any filter (a.k.a filter's *receptive field* [33]) increases as it leverages what is already learned so far by all layers. This is necessary to generate embeddings that encode complex and deeper scheduling decisions (e.g., schedule the execution of three consecutive push-down *selects* in the physical query plan as one pipeline on the same thread). However, another interesting observation comes up: the execution behaviour of one node could be affected by the behaviour of its grand children from one child node more than the other one. For example, let us revisit the query defined in Figure 4. Assume that the relation A is 10 times larger than B. This makes the latency of the join operator o_4 more dependable on the latency of o_3 than on o_2, because o_3 will probably become an execution bottleneck (assuming that the output of o_3 is proportional to the size of the largest relation, i.e., A). However, there is no explicit way to increase and decrease the importance of o_3 and o_2, respectively, for o_4 during the embedding calculation. This is a known issue, in both GCN and tree convolution, where the convolution executes an *isotropic aggregation* [14, 38], in which each neighbor node contributes *equally* to update the representation (i.e., embeddings) of the central node.

To remedy this issue, we apply graph attention networks (GAT) [56] to assign different importance to each node and edge contributing to the tree convolution output. The main idea is to compute a "learnable" pair-wise *attention score* between the parent node p and each node or edge used in calculating the tree convolution value x'^l_p at the parent p in Equation 2. Since the previous embedding values of p (i.e., x^l_p) is used in calculating x'^l_p, we consider adding an attention score for p itself as well. This means that the triangle filter at any convolution layer l has five attention scores in total. Each attention score is used to weigh the corresponding embedding when calculating the final value of x'^l_p.

Since all attention scores are calculated in the same way, we will show, as an example, the details of computing one attention score, which is z^l_m (i.e., the attention score between the parent node p and its right child). Let us assume x^{*l}_m to be the current embedding of right child weighted by its corresponding weight vector w^l_m in the tree convolution filter (i.e., $x^{*l}_m = w^l_m \odot x^l_m$, where $\odot$ is the Hadamard product). First, a pair-wise *un-normalized* attention score y^l_m between the parent node p and its right child is calculated as:

$$y^l_m = \sigma(a^l \odot (x^l_p || x^{*l}_m)) \tag{3}$$

where || donates the concatenation of the x^l_p and x^{*l}_m embeddings. Basically, a Hadamard product is applied between the concatenated embeddings and a learnable weight vector a^l. The vector a^l represents a shared single-layer neural network which is learned during the training phase of the scheduling neural networks (Section 5). σ is a non-linear transformation function (e.g., LeakyReLU [56]). The score y^l_m indicates the importance of the information coming from the right child during applying the tree convolution process. However, to have comparable attention scores across different nodes, each un-normalized score should be normalized using softmax function as follows:

$$z^l_m = \frac{exp(y^l_m)}{exp(y^l_m) + exp(y^l_n) + exp(y^l_{p,m}) + exp(y^l_{p,n}) + exp(y^l_p)} \tag{4}$$

Here, z^l_m is final normalized score for y^l_m, and is calculated using the other un-normalized scores. Once the final five normalized attention scores are calculated, the tree convolution value x'^l_p at parent node p can be calculated as follows:

$$x'^l_p = \sigma(z^l_p * x^{*l}_p + z^l_{p,m} * e^{*l}_{p,m} + z^l_m * x^{*l}_m + z^l_{p,n} * e^{*l}_{p,n} + z^l_n * x^{*l}_n) \tag{5}$$

where x^{*l}_p, $e^{*l}_{p,m}$, $e^{*l}_{p,n}$, and x^{*l}_n are calculated similar to x^{*l}_m. Equation 5 is very similar to Equation 2, yet, the weighted embeddings are now scaled by the attention scores, reflecting their importance.
Example. Figure 5 shows the effect of using GAT scores on the embedding outputs of 2-layers TCN. In this example, we revisit the same query in Figure 4, while assuming that the relation A is significantly larger than B as discussed earlier (i.e., o_3 is more important than o_2 for o_4). We also included the edge features EDF (check Section 4.1) and their corresponding edge weights to the tree convolution filter. As we can see, using the learned GAT scores, that reflect the correct impact of operators on each other, results in better embedding at o_4 in layer 2 (i.e., less negative and higher positive values). Note that embeddings are calculated using Equation 5, while using the GAT scores in Figure 5 and $\sigma = 1$ for simplicity.

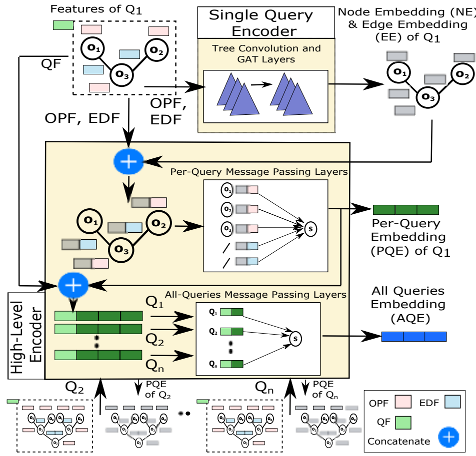

Figure 6: Neural networks architecture of *Query Encoder*.

4.3 High-level Encoding

In addition to encoding the different operators in a single query, *LSched* requires a higher-level representation that captures the whole query as one unit (e.g., one embedding vector that summarizes the total work orders in the query), and similarly, another global representation that captures whole running queries at the moment (e.g., capturing the system resources utilization). In general, the idea of generating high-level encodings is not new, and has been studied before in Decima [34]. Figure 6 shows the neural networks architecture of *Query Encoder* in *LSched*, with its two components: *Single Query Encoder* (check Section 4.2) and *High-Level Encoder*, which generates two types of summarized embeddings: *Per-Query Embedding* (PQE) and *All-Queries Embedding* (AQE).

The *PQE Embedding* is created for each query through a summarization neural network that connects false directed edges (with no features) from the query nodes and edges to a dummy summary node. As a result, all the nodes and edges become children to this summary node. Figure 6 depicts the details of PQE embedding for a simple join query Q_1. Once the node (NE) and edge (EE) embeddings of Q_1 are generated by the single query encoder as shown in Section 4.2, these embeddings are concatenated with the OPF (i.e., node) and EDF (i.e., edge) features (check Section 4.1) as shown in the figure. Then, the concatenation output is passed to the PQE summarization network to generate the PQE embedding of this query at the summary node (1-dimension vector).

The *AQE Embedding* is created for all running queries at the moment through another summarization neural network that connects all PQE summaries of these queries to a single global summary node (i.e., the PQE summaries become children of this global node) to obtain a global embedding. Figure 6 shows the details of AQE embedding for multiple queries $Q_1, Q_2, ..., Q_n$. Once the PQE embeddings for all queries are generated, each embedding is concatenated with the QF (i.e., query) features (check Section 4.1) of its corresponding query. Then, the concatenation output for all queries is passed to the AQE summarization network to generate the final AQE embedding (1-dimension vector).

Both PQE and AQE summarization networks are implemented using a typical message passing-based [23] neural network.

5 SCHEDULING PREDICTOR

In this section, we describe the details of the core component of *LSched*'s scheduling agent, namely *Scheduling Predictor*, which is a deep neural network that leverages the information provided by the *Query Encoder* (Section 4). We first describe the execution model assumptions that *LSched* works within (Section 5.1). Then, we describe when the *Scheduling Predictor* should be triggered (Section 5.2). Finally, we present the details of of scheduling decisions that should be taken at each scheduling event along with the neural networks architecture to perform these decisions (Section 5.3).

5.1 Execution Model Assumptions

The query scheduler should take decisions that are consistent with the execution model of the underlying query execution engine. Since different database systems could have different implementation details and optimizations for their execution engines, we choose to build *LSched* for a model, that is widely-used among modern database systems nowadays. In this model, the execution engine consists of a single scheduler thread and a pool of worker threads. The scheduler thread uses the query physical plans to generate and schedule work for the worker threads. Each worker thread is pinned to a CPU core (could be a virtual core), and is responsible for executing the work orders coming from a certain scheduled operator. By default, worker threads are created when the database system is instantiated and kept alive across query executions to minimize the initialization costs. However, the worker threads pool can shrink or grow dynamically during execution. Note that Quickstep follows this execution model to a great extent, and hence can make a full use of *LSched*'s scheduling decisions. For convenience, we refer to the worker threads as *execution threads* or just *threads* during the rest of this section.

5.2 Scheduling Triggers

An important design aspect of *LSched* is deciding when it should be invoked. Using RL to formulate the query scheduling problem gives us a large action space to select from. For example, one straightforward option is to let *LSched* schedule all available operators from all existing queries in one scheduling decision. However, such decision is extremely inefficient for two reasons. First, the scheduler becomes less interactive as it schedules everything once, and will wait for a while till the next scheduling event happens. The repercussions of this could be severe specially in large analytical queries that might take seconds to complete their execution. This is long enough for the execution environment to significantly change and hence the taken decision becomes outdated. Second, it requires encoding an extremely large action space since there is a large set of possible scheduling combinations defined over operators and threads. Another possible option is to invoke the scheduler whenever a single work order completes. This option considers the execution environment changes in a better way, however, it is still an inefficient decision as aggressively invoking the scheduler will incur a substantial scheduling overhead that could outweigh the scheduling benefit (Section 7 has reported overhead numbers).

To balance between the scheduling overhead and benefits, we design *LSched* such that its *Scheduling Predictor* is triggered only on the major events happening to *threads* and/or *queries* (we refer to them as *scheduling events*). These events include: (1) adding or

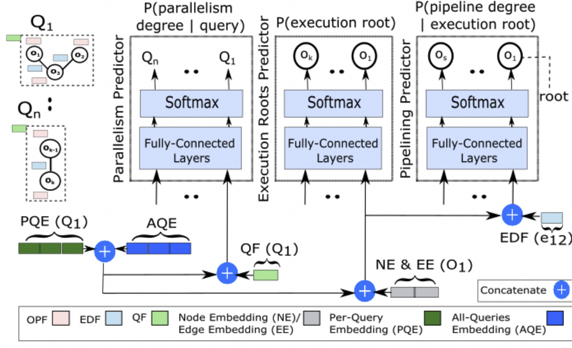

Figure 7: Neural networks architecture of *Scheduling Predictor*.

removing a thread to the pool, (2) a thread finished the execution of all its assigned work orders from one or more scheduled operators, (3) a new query arrives to the system, and (4) a scheduled operator has been completely executed. Note that the scheduler will not make any decisions if all threads are assigned work orders from scheduled operators, or there are no more operators to schedule.

5.3 Scheduling Decisions

The main objective of *LSched* is to optimize both *inter-query* and *intra-query* scheduling. To support that, the *Scheduling Predictor* is designed to learn and perform the following three scheduling decisions at any scheduling event: (1) which operators to start execution from (we refer to them as *execution roots*) and from which queries (Section 5.3.1), (2) the proper degree of pipelining that starts from each execution root (Section 5.3.2), and (3) the proper degree of parallelism for each running query (Section 5.3.3).

Figure 7 shows the neural networks architecture of the *Scheduling Predictor*. Once the generated embeddings from the *Query Encoder* (shown in Figure 6) become ready, the three scheduling decisions can be calculated. Basically, each decision is the output of a fully-connected layers network with a softmax operation for normalization. However, each decision network differs from the others in its objective and input. Once these scheduling decisions are made, they are sent to the query executor to prepare all work orders from the scheduled operators and execute them on the specified threads.

5.3.1 Execution Roots Predictor. In this scheduling decision, we calculate the probability of each operator (in each query) to be selected for scheduling. An operator is schedulable if all its blocking parents are completely executed. For each operator selected to be scheduled, *LSched* considers it as a root (i.e., source) of a potential pipeline, and then this root is passed to the *Pipeline Degree Predictor* to figure out the proper degree of this pipeline (i.e., how many subsequent pipelined operators to run along with this root).

As seen in Figure 7, the concatenated input of this decision neural network for each operator (e.g., o_1) consists of (1) the node (i.e., operator) and edge embeddings (NE and EE) of this operator and (2) the per-query embeddings (PQE) of this operator's query.

5.3.2 Pipeline Degree Predictor. In this scheduling decision, we calculate, for each execution root (i.e., operator selected by the *Execution Roots Predictor*), the probability of selecting a degree d between 1 and the length of the longest pipelining path starting from this operator (i.e., scheduling a pipeline of d operators starting from the execution root). In case there is a pipeline breaking mode or no more operators exist in the path starting from the *root*, this number is set to 0 and the execution root will be scheduled only.

Predicting the pipeline degree is a major difference from Decima [34], which can not schedule two or more pipelined operators from one query at the same time, to be running on the same thread. Pipelining is very crucial for database systems because it increases caching utilization as well as the query throughput. However, being greedy in simultaneously running all operators in long pipelines (e.g., a long sequence of *select* operators) consumes memory buffers at a high rate and causes thrashing, that will hurt the overall performance at the end. Therefore, it is required to learn the effective degree of pipelining that can be used based on the input workload, which is exactly supported in *LSched*.

As seen in Figure 7, the concatenated input of this decision neural network, for each root operator (e.g., o_1), is exactly the same as in the *Execution Roots Predictor* in addition to the edge features (EDF) of all edges connected to this root operator to obtain the pipeline.

5.3.3 Parallelism Degree Predictor. In this scheduling decision, we calculate for each query the probability of selecting a number of threads between 1 and maximum number of available threads to be used with this query. If this query currently uses less number of threads than the predicted maximum, *LSched* assigns threads for the scheduled operators, if any, up to this maximum. The main intuition behind that is if the query, that the scheduled operators belong to, had previous operators that were running on some threads, then reusing these threads with the newly scheduled operators will improve the locality.

As seen in Figure 7, the concatenated input of this decision neural network for each query (e.g., Q_1) consists of (1) the all-queries embedding (AQE), (2) the per-query embeddings (PQE) of this query, and (3) the query features (QF) as in Section 4.1.

6 SCHEDULER TRAINING

Reducing Average and Tail Latency. We perform the training of *LSched* in episodes, where each episode represents a sample workload of queries with different arrival patterns. Within each episode, the scheduler takes many scheduling decisions, where at each decision d, the scheduling agent collects a record of state s_d, action a_d, and reward r_d and adds it to a list of reward experiences. Although any RL algorithm can be used to train the neural network parameters of *LSched*, we chose the REINFORCE policy gradient algorithm [62] for two main reasons. First, it updates its policy using Monte-Carlo sampling [15], which balances between accuracy and runtime efficiency. Second, it reduces the variance in the parameters estimation process by using reward baselines (e.g., [61]).

A key factor to perform an efficient training using REINFORCE or any policy gradient algorithm is the design of the reward r_d because the gradient algorithm performs a gradient descent on the neural network parameters using the whole observed rewards during each training episode. In *LSched*, we design the reward r_d with an objective of minimizing both *average* and *tail* latency of running queries. Let t_d and t_{d-1} be the times of taking the scheduling decisions d and $d-1$ within a training episode, and Q_d be the number of existing queries in the system in the time interval

between t_{d-1} and t_d. We use the quantity $H_d = (t_d - t_{d-1})Q_d$ as an approximation for the latency of the Q_d queries since these queries are still running and do not have actual completion times. We generate a list H of all latency approximations corresponding to all scheduling decisions in the same training episode. At the end of the episode, we calculate the 90th percentile, referred to as P, of all the latency approximation values in H and use it as an indicator for the tail latency of all queries that ran in the episode. Using the calculated H_d and P values, *LSched* defines two rewards: $r_d^1 = -H_d$, and $r_d^2 = -(H_d - P)$, which target minimizing the *average* and *tail* latency, respectively. The final reward r_d is a weighted average of r_d^1 and r_d^2, where $r_d = \frac{w_1 r_d^1 + w_2 r_d^2}{w_1 + w_2}$. The weights w_1 and w_2 are hyper-parameters set by users to control the rewarding criteria.

Transfer Learning. Inspired by the recent advances of transfer learning in deep learning [2], we can train neural network models (e.g., weights of hidden layers) for the *Query Encoder* and *Scheduling Predictor* on one query workload, and then apply them on a new workload with some customization. In this way, we can avoid building the models from scratch and shorten the training cycle (i.e., reduce the number of needed training episodes). The main intuition behind that is, in many neural networks, the first layers (i.e., layers close to the input) refer to general embeddings, while last layers (i.e., layers close to the output) refer to specific embeddings that are problem dependent. This is valid in our scheduling problem: for example, the first layers of the tree convolution process could recommend aggressive scheduling for all *select* operators (workload-independent guideline), while the last layers could specifically recommend scheduling *select* operators over relation A before the ones over relation B (workload-dependent guideline).

LSched exploits this observation and allows reusing its previously-learnt neural network layers, while re-training some layers and freezing others. Specifically, the neural networks, used in both the query encoder and scheduling predictor, can apply transfer learning by freezing all their convolution and hidden layers, except the layers connected to the input and output of each network, which should be retrained according to the new workload (i.e., retraining the layers' weights). Note that reusing the frozen layers is permissible in *LSched* as the dimensions of these layers remain the same among different workloads.

7 EXPERIMENTAL EVALUATION

We evaluated the performance of *LSched* using different benchmarks to answer the following questions: (1) how does the performance of *LSched* compare with both state-of-the-art query schedulers and carefully-tuned heuristics based schedulers (Section 7.2)? (2) how do the different environment and query scenarios impact the overall performance (Section 7.3)? (3) what is the scheduling overhead (Section 7.4)? (4) how does the training affect the overall performance (Section 7.5)? (5) How does each of the proposed ideas contribute to *LSched*'s performance (Section 7.6)?

7.1 Experimental Setup

Competitors. We compare *LSched* against five baselines: (1) Decima: an open-source [3] learned task scheduler for cluster computing environments [34]. (2) SelfTune: a recent query scheduler [58] for in-memory analytical workloads (we got its executable from the authors). (3) Quickstep: an open-source [46] analytical database system [43], which has a built-in query scheduler. (4) Fair scheduling: a carefully-tuned weighted fair scheduling provided by Quickstep. (5) FIFO scheduling: a naive scheduler which runs queries in the same order they arrive, implemented in Quickstep.

Hardware and Settings. All experiments are conducted on an Arch Linux machine with 256 GB of RAM and an Intel(R) Xeon(R) Gold 6230 CPU @ 2.10GHz with Skylake micro architecture (SKX).

Benchmarks. We use three benchmarks: (1) TPCH: the default TPCH [52], with scale factors 2, 5, 10, 50 and 100. (1) SSB: the Star Schema Benchmark [41], with scale factors 2, 5, 10 and 50. (3) JOB: the default Join Order Benchmark [30], with IMDB dataset of 7.2GB.

Workloads Generation. To build a workload, either for training or testing, we do the following process: let us build a TPCH training workload, for example. For each scale factor, we randomly select, without replacement, 50% of the benchmark queries (11 queries in the TPCH case) to be used for training, and leave the remaining queries for testing (we make sure that testing queries were never seen during training). This means that there is a total of 55 queries, from all scale factors, we can use for generating workloads. A workload of size x is generated by randomly selecting x times, with replacement, from these 55 queries. To generate workloads with different arrival times, we follow the process of generating continuous arrival queries in [58], where the spacing between queries arrival is sampled from an exponential distribution with expected value $1/\lambda$. Using this, we can control the expected arrival rate of λ queries per second. We follow the same process for generating training/testing workloads for TPCH/SSB/JOB. In case of JOB, we directly generate any workload of size x using the randomly sampled queries from the original benchmark (114 queries), whether training or testing, (i.e., no scale factor to enlarge the number of queries).

***LSched* Implementation.** *LSched* training is implemented using TensorFlow [51]. The physical plan features generation is implemented inside Quickstep in C++. During testing, the communication between Quickstep and *LSched* is done through an RPC interface.

Default Settings. The parameters of all competitors are configured according to the guidelines in their original papers. Pipelining is always enabled in all competitors. Unless otherwise stated, the maximum number of execution threads is set to 60. We used *LSched* to build a default scheduling model for TPCH workloads using 5000 training episodes, where each episode has a number of streaming queries that vary between 20 to 100, and arrive with rates (λ) that vary between 10 to 400. For each of SSB and JOB, we built another scheduling model using 3000 training episodes, where each episode has a number of streaming queries that vary between 10 to 200, and arrive with rates (λ) that vary between 10 to 400. The weights for *average* and *tail* latency optimization (Section 6) are 0.5. We use the cumulative distribution function (CDF) of average query duration in the system as the default evaluation metric.

7.2 Comparison with Other Competitors

We first study the performance of *LSched* against all scheduling competitors with three different benchmarks (TPCH, SSB and JOB), and with two different arrival patterns (streaming and batching). In batching, all queries arrive the system at the same time. This is a typical batch processing scenario that happens frequently with

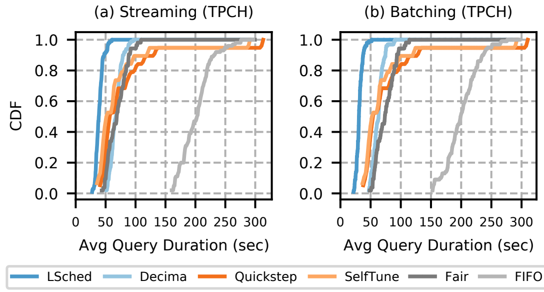

Figure 8: Comparing *LSched* with other scheduling baselines under streaming and batched (TPCH) queries.

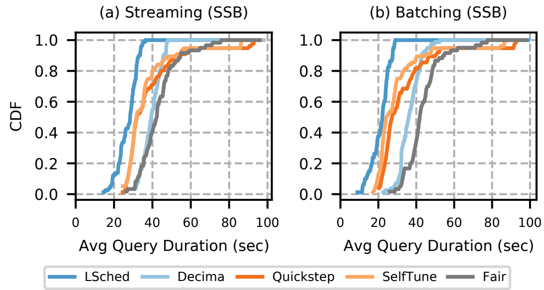

Figure 9: Comparing *LSched* with other scheduling baselines under streaming and batched (SSB) queries.

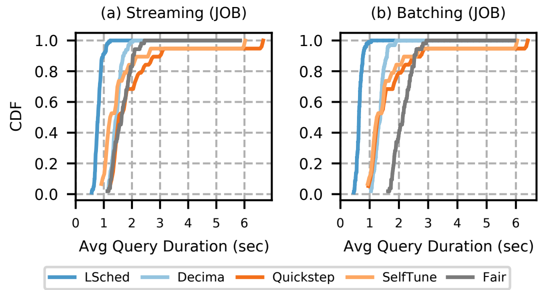

Figure 10: Comparing *LSched* with other scheduling baselines under streaming and batched (JOB) queries.

analytic workloads (e.g., the user provides a script with all queries that need to run in advance). In this mode, the database system becomes under high pressure, and all its resources are highly utilized for a considerable amount of time. On the other hand, in streaming, queries arrive at different time instants, and the system could be highly utilized at certain periods, and under utilized in others.

Figure 8 shows the CDF of average query duration in the system for two testing TPCH workloads that both have 80 queries, but arriving in a different way. Clearly, FIFO scheduling has the worst performance by far as it runs queries in the same order they arrive in and grants as many threads to each query as available, which stalls the execution of other queries and significantly increases their average query duration. Since FIFO is always the worst baseline, we removed it from all the remaining experiments.

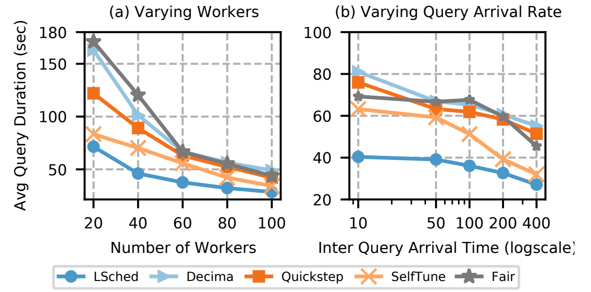

Figure 11: Studying the performance while varying (a) number of worker threads in the query execution engine and (b) inter-query arrival time.

As shown in Figure 8, *LSched* outperforms all baseline algorithms and improves the average query duration over Decima by at least 35% and 50% in both streaming and batching cases, respectively. This is because *LSched* prioritizes queries better, leverages the relational query structure better using efficient encoding techniques (e.g., tree convolution), and wisely controls the pipelining. As you can see, *LSched* has a much larger impact in batching than in streaming. This is because, in batching, the system faces severe high-load periods, which when good scheduling decisions have the most impact in reducing the workload peaks.

Figures 9 and 10 confirm the superiority of *LSched* over all baselines, but now under the SSB and JOB benchmarks, respectively. An interesting observation for the SSB benchmark is that the performance gap between *LSched* and the competitors decreases a bit compared to the TPCH benchmark. This is because, as we mentioned in the experimental setup, the highest scale factor used with the SSB benchmark is 50, which makes the worst query runtime in SSB is almost half of its counterpart in the TPCH workload (i.e., SSB has lighter queries), and in turn the system becomes more relieved. This confirms our previous conclusion about *LSched* being most effective when the system is extremely busy. For the JOB benchmark, we observe that *LSched* has a bit better performance gain over all other baselines (at least 38% and 59% average query duration improvement in both streaming and batching cases, respectively), compared to the TPCH and SSB benchmarks. This is because JOB benchmark has more challenging queries (some queries have more than 10 join operations) that require careful scheduling, and hence the superiority of *LSched* appears. In addition, it is clear that the variance of average duration times for *LSched*, in all figures, is much less than all other baselines. This is because of *LSched*'s support to reduce both average and tail latency (Section 6).

7.3 Environment and Query Scenarios

Figure 11(a) shows the performance of all schedulers while scaling up the number of execution threads from 20 to 100. In this experiment, we use the testing TPCH workload that contains 80 streaming queries and report the average query duration.

Overall, all algorithms scale well when increasing the number of threads, although only fair scheduling achieves better scalability than others at very high number of threads. The main explanation for this good performance is that with this high number of threads (e.g., 100), which is even larger than the number of existing queries,

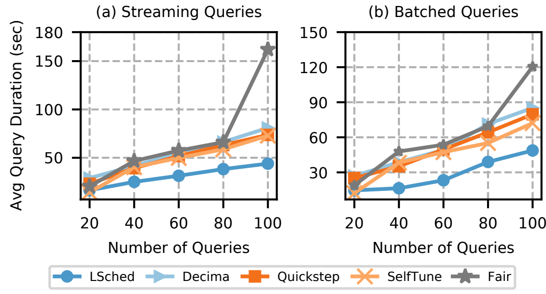

Figure 12: Studying the performance while varying number of (a) streaming and (b) batched queries.

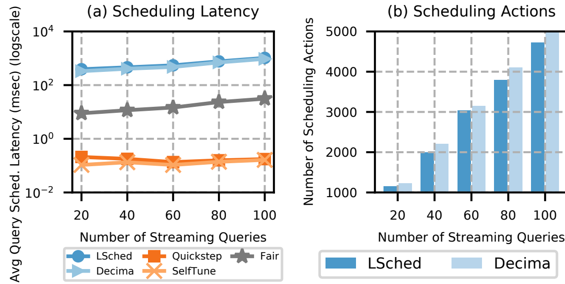

Figure 13: Analyzing the scheduling overhead in terms of (a) average latency per query, and (b) number of scheduling actions taken by the learned agent.

smart scheduling decisions have less effect, and fair sharing of resources still ends up with enough resources for each task to make a progress. Figure 11(b) shows the average query duration of all schedulers, under the testing TPCH workload of 80 streaming queries, but now, while varying the inter-query arrival rate from 10 to 400. As expected, and similar to the conclusion in Figure 11(a), the performance gap between *LSched* and others significantly decreases when the system almost runs a single query at a time. In this case, many schedulers tend to have the same scheduling decisions.

Figure 12 shows the sensitivity of all algorithms while varying the number of streaming and batching queries in the test TPCH workload. Here, we use the default number of threads which is 60. As shown, when the number of queries is very small, most schedulers become similar or slightly worse than each other. In contrast, when the number of queries becomes higher than execution threads, the performance of schedulers starts to degrade peacefully (except fair scheduling), with a clear advantage for *LSched*.

7.4 Scheduling Overhead

In this experiment, we measure the scheduling overhead of *LSched* compared to the other baselines. Figure 13(a) depicts the average scheduling latency per query for all schedulers while changing the number of queries in the test TPCH streaming workload from 20 to 100. We observe two things here. First, the scheduling overhead of learned approaches, both *LSched* and Decima, is significant. This is due to the expensive processing of convolution and neural network operations that happens per each scheduling decision in these learned approaches. In contrast, the remaining schedulers just

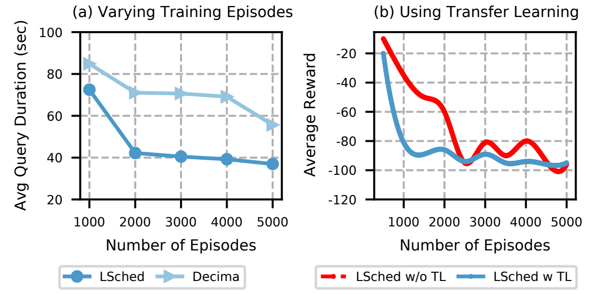

Figure 14: Effect of changing (a) number of training episodes and (b) using transfer learning on the *LSched* efficiency.

perform few function calls or run a simple heuristic algorithm on each scheduling event. *However, it is clear that this overhead is paid off by saving hundreds of seconds in the execution time* due to the effectiveness of the taken scheduling decisions (time saving is still 100x greater than this overhead on average). Second, the scheduling latency for the learned approaches increases a bit when increasing the number of queries in the system. This is also expected as the number of needed scheduling decisions increases. Figure 13(b) shows how the number of scheduling decisions increases for the same experiment in Figure 13(a).

7.5 Effect of Training

Figure 14(a) shows how increasing the number of training episodes boosts the accuracy of the scheduling decisions and hence decreases the average query duration. The figure compares both *LSched* and Decima while varying the number of episodes from 1000 to 5000. As we can see, *LSched* saturates much faster than Decima. *LSched* takes only 2000 episodes while Decima takes 5000. This is mainly because of the efficiency of the *LSched*'s neural network design and its underlying convolution techniques.

Figure 14(b) shows the effectiveness of using transfer learning to speed up the training process. In this experiment, we used the already-built scheduling model for the TPCH workloads to train a scheduling model for an SSB workload, and then compared that with building a new scheduling model from scratch for the same workload. The figure shows the number of training episodes completed on x-axis and the achieved average reward corresponding to these episodes on the y-axis. We can see that using transfer learning the number of episodes needed to reach to an effective average reward is reduced by 50%. Note that the reward is shown in a negative value because it represents latency penalties (Section 6).

7.6 Different Variations of *LSched*

Figure 15 shows that removing any one component from *LSched* results in worse average query duration. Here, the complete variation of *LSched* is trained with transfer learning (blue curve). As you can see, using tree convolution (TCN) and graph attention mechanism (GAT) (i.e., not weighing the importance of child operators and attached edges) has the greatest impact on *LSched*'s performance. Without TCN and GAT, the average query duration becomes at least 2X and 1.5X worse, respectively. Another interesting observation is the effect of transfer learning on the accuracy performance.

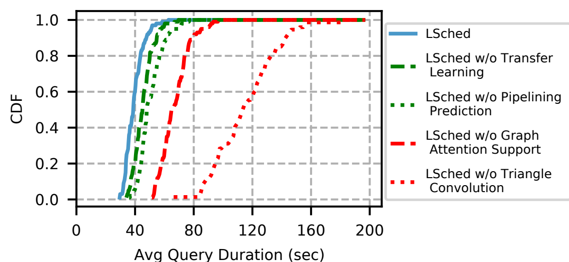

Figure 15: Studying the performance of different *LSched* variations (each variation represents *LSched* without a specific key contribution).

Transfer learning mainly aims to expedite the training phase as shown in Figure 14(b), and hence one expects that it has no effect on the accuracy. However, our results show that removing the transfer learning makes the average query duration 10% worse. This is because, using transfer learning, the neural network starts training from some meaningful embeddings (i.e., less random) and hence consumes more iterations in optimizing them, which in turn improves the scheduling performance. Lastly, as expected, ignoring the pipelining prediction during scheduling has its significant effect which increases the average query duration time with 25%.

8 RELATED WORK

Scheduling is a classical problem that has been studied for decades in different domains including databases (e.g., [16]), networking (e.g., [48]) and operating systems (e.g., [27, 59]). However, we focus here on the literature of query scheduling for database and distributed systems only.

Scheduling in Analytical Database Systems. We start with query scheduling for analytical workloads, which is the main focus of this paper. Introducing an explicit query scheduler for analytical workloads has been studied in few systems, such as Umbra [40], SAP HANA [44], HyPer [20] and Quickstep [43]. For example, in Umbra, a recent query scheduler, namely SelfTune [58], was proposed to self-tune the hyper-parameters of its fixed priority-based scheduling policy for each input workload. In Quickstep, an efficient query scheduler was proposed with tuned implementations for many heuristic policies (e.g., fair, highest priority first, and proportional priority) that can work on a very fine-grained level of tasks. However, the query schedulers in these systems are still heuristics-based, and are not optimal for the input workload (i.e., policies are not built from scratch to take the workload characteristics into account). Unlike these systems, *LSched* provides the ability to have fully-learned workload-aware scheduling policies.

Scheduling in Computing Clusters. There has been an extensive research effort to provide efficient task scheduling in large clusters. For example, Google developed Borg [57], that can schedule hundreds of thousands of jobs, from many thousands of different applications, across thousands of machines. Yarn [55] and Mesos [17] are other examples on efficient open-source resource managers for sharing clusters of commodity hardware, such as Hadoop, in a fine-grained manner. In general, these resource managers try to achieve instantaneous fairness among different jobs. In contrast, [11] studied how jobs can yield fractions of their current allocated resources to guarantee long-term fairness. Decima [34], which uses RL to fully-learn a jobs scheduler on large clusters, is the closest to *LSched* in the approach, but different in the scheduling objective. Decima aims to schedule tasks among large cluster nodes, while *LSched* focuses on scheduling queries among threads on a single node database system. This means that the neural networks design and features needed for both systems are quite different.

ML for Databases. During the last few years, machine learning started to have a profound impact on automating the core database functionality and design decisions [24]. For example, a corpus of works studied the idea of replacing traditional indexes with learned models that predict the location of a key in a dataset including single-dimension (e.g., [7, 22, 25]), multi-dimensional (e.g., [4, 39]), updatable (e.g., [5]), and spatial (e.g., [31, 42, 45]) indexes. Query optimization is also another area that has several works on using machine learning to either entirely build an optimizer from scratch (e.g., [37]), or provide an advisor that improves the performance of existing optimizers (e.g., [36, 50, 54]). Moreover, there exists several optimizations, enabled by machine learning, in other database operations e.g., cardinality estimation [21], data partitioning [63], sorting [26], and multi-query execution [49]. However, to the best of our knowledge, there is no existing work on using the machine learning to revisit the query scheduling for analytical database systems. Our proposed work in this paper fills this gap.

9 CONCLUSION

In this paper, we introduce *LSched*, a fully learned workload-aware query scheduler for in-memory analytical database systems. *LSched* provides an efficient *inter-query* and *intra-query* scheduling for dynamic analytical workloads. *LSched* supports efficient and accurate query encoding, flexible scheduling decisions, balancing between average and tail latency, and effective training time. We integrated *LSched* with an efficient in-memory analytical database system, namely Quickstep, and evaluated it with TPCH, SSB and JOB benchmarks. Our evaluation shows that *LSched* improves over the performance of existing state-of-the-art query schedulers and heuristic-based ones by at least 35% and 50% in both streaming and batching query workloads.

ACKNOWLEDGMENTS

This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, and is partially supported by the NSF, USA, under grants IIS-1900933, and CCF-2030859 (an award to the Computing Research Association for the CIFellows Project). This research was also sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] Peter Battaglia, Jessica Blake Chandler Hamrick, Victor Bapst, Alvaro Sanchez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, Caglar Gulcehre, Francis Song, Andy Ballard, Justin Gilmer, George E. Dahl, Ashish Vaswani, Kelsey Allen, Charles Nash, Victoria Jayne Langston, Chris Dyer, Nicolas Heess, Daan Wierstra, Pushmeet Kohli, Matt Botvinick, Oriol Vinyals, Yujia Li, and Razvan Pascanu. Relational inductive biases, deep learning, and graph networks. *arXiv*, 2018.

[2] Yoshua Bengio. Deep Learning of Representations for Unsupervised and Transfer Learning. In *Proceedings of ICML Workshop on Unsupervised and Transfer Learning*, 2012.

[3] Decima open-source. https://github.com/hongzimao/decima-sim.

[4] Jialin Ding et al. Tsunami: A Learned Multi-Dimensional Index for Correlated Data and Skewed Workloads. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2020.

[5] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. ALEX: An Updatable Adaptive Learned Index. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2020.

[6] Siying Dong, Mark Callaghan, Leonidas Galanis, Dhruba Borthakur, Tony Savor, and Michael Strum. Optimizing Space Amplification in RocksDB. In *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2017.

[7] Paolo Ferragina and Giorgio Vinciguerra. The PGM-Index: A Fully-Dynamic Compressed Learned Index with Provable Worst-Case Bounds. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2020.

[8] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation, NSDI*, 2011.

[9] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep Sparse Rectifier Neural Networks. In *Proceedings of the International Conference on Artificial Intelligence and Statistics*, 2011.

[10] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. Multi-Resource Packing for Cluster Schedulers. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM*, 2014.

[11] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic Scheduling in Multi-Resource Clusters. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation, OSDI*, 2016.

[12] Martin Grund, Jens Krüger, Hasso Plattner, Alexander Zeier, Philippe Cudre-Mauroux, and Samuel Madden. HYRISE: A Main Memory Hybrid Storage Engine. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2010.

[13] Chetan Gupta, Abhay Mehta, Song Wang, and Umesh Dayal. Fair, Effective, Efficient and Differentiated Scheduling in an Enterprise Data Warehouse. In *Proceedings of the International Conference on Extending Database Technology, EDBT*, 2009.

[14] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *Proceedings of the Annual Conference on Neural Information Processing Systems, NIPS*, 2017.

[15] W. K. Hastings. Monte Carlo Sampling Methods Using Markov Chains and Their Applications. *Biometrika*, 57(1):97–109, 1970.

[16] Joseph M. Hellerstein, Michael Stonebraker, and James Hamilton. Architecture of a Database System. *Found. Trends Databases*, 1(2):141–259, 2007.

[17] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation, NSDI*, 2011.

[18] Stratos Idreos, Fabian Groffen, Niels Nes, Stefan Manegold, K. Sjoerd Mullender, and Martin L. Kersten. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Engineering Bulletin*, 35(1):40–45, 2012.

[19] James E. Kelley and Morgan R. Walker. Critical-Path Planning and Scheduling. In *Eastern Joint IRE-AIEE-ACM Computer Conference*, 1959.

[20] Alfons Kemper and Thomas Neumann. HyPer: A Hybrid OLTP and OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *Proceedings of the International Conference on Data Engineering, ICDE*, 2011.

[21] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2019.

[22] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. RadixSpline: A Single-Pass Learned Index. In *Proceedings of the aiDM workshop @SIGMOD*, 2020.

[23] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations, ICLR*, 2017.

[24] Tim Kraska. Towards instance-optimized data systems. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2021.

[25] Tim Kraska, Alex Beutel, Ed H. Chi, Jeffrey Dean, and Neoklis Polyzotis. The Case for Learned Index Structures. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, page 489–504, 2018.

[26] Ani Kristo, Kapil Vaidya, Ugur Çetintemel, Sanchit Misra, and Tim Kraska. The Case for a Learned Sorting Algorithm. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2020.

[27] Butler W. Lampson and Howard E. Sturgis. Reflections on an Operating System Design. *Communications of ACM*, 19(5), 1976.

[28] Chiang Lee, Chi-Sheng Shih, and Yaw huei Chen. Optimizing Large Join Queries Using A Graph-Based Approach. *IEEE Transactions on Knowledge and Data Engineering, TKDE*, 13, 2001.

[29] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2014.

[30] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. How Good Are Query Optimizers, Really? In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2015.

[31] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. LISA: A Learned Index Structure for Spatial Data. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2020.

[32] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E. Alsaadi. A Survey of Deep Neural Network Architectures and their Applications. *Neurocomputing*, 234, 2017.

[33] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, 2015.

[34] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication, SIGCOMM*, 2019.

[35] Gary Marcus. Innateness, AlphaZero, and Artificial Intelligence. *ArXiv*, abs/1801.05667, 2018.

[36] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. Bao: Making Learned Query Optimization Practical. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2021.

[37] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. Neo: A Learned Query Optimizer. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2019.

[38] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *Proceedings of the AAAI Conference on Artificial Intelligence, AAAI*, 2016.

[39] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska. Learning Multi-Dimensional Indexes. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2020.

[40] Thomas Neumann and Michael Freitag. Umbra: A Disk-Based System with In-Memory Performance. In *Proceedings of the International Conference on Innovative Data Systems Research, CIDR*, 2020.

[41] P. O'Neil, E. O'Neil, and X. Chen. The star schema benchmark. http://www.cs.umb.edu/~poneil/StarSchemaB.pdf,Jan2007.

[42] Varun Pandey, Alexander van Renen, Andreas Kipf, Ibrahim Sabek, Jialin Ding, and Alfons Kemper. The Case for Learned Spatial Indexes. In *Proceedings of the AIDB Workshop @VLDB*, 2020.

[43] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. Quickstep: A Data Platform Based on the Scaling-up Approach. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2018.

[44] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. Scaling up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-Aware Data and Task Placement. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2015.

[45] Jianzhong Qi, Guanli Liu, Christian S. Jensen, and Lars Kulik. Effectively Learning Spatial Indices. In *Proceedings of the International Conference on Very Large Data Bases, VLDB*, 2020.

[46] Quickstep open-source. https://github.com/apache/incubator-quickstep.

[47] David B. Shmoys, Clifford Stein, and Joel Wein. Improved Approximation Algorithms for Shop Scheduling Problems. *SIAM Journal on Computing*, 23(3), 1994.

[48] M. Shreedhar and G. Varghese. Efficient Fair Queuing Using Deficit Round-Robin. *IEEE/ACM Transactions on Networking*, 4(3), 1996.

[49] Panagiotis Sioulas and Anastasia Ailamaki. *Scalable Multi-Query Execution Using Reinforcement Learning*. 2021.

[50] Michael Stillger, Guy M. Lohman, Volker Markl, and Mokhtar Kandil. LEO - DB2's LEarning Optimizer. In *Proceedings of the International Conference on Very*

Large Data Bases, VLDB, pages 19 – 28, 2001.

[51] TensorFlow - homepage. https://www.tensorflow.org/.

[52] TPC-H - homepage. http://www.tpc.org/tpch/.

[53] J.D. Ullman. NP-complete Scheduling Problems. *Journal of Computer and System Sciences*, 10(3):384–393, 1975.

[54] Gary Valentin, Michael Zuliani, Daniel C. Zilio, Guy M. Lohman, and Alan Skelley. DB2 Advisor: An Optimizer Smart Enough to Recommend its Own Indexes. In *Proceedings of the International Conference on Data Engineering, ICDE*, pages 101 – 110, 2000.

[55] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache Hadoop YARN: yet another resource negotiator. In *ACM Symposium on Cloud Computing, SOCC*, 2013.

[56] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *International Conference on Learning Representations, ICLR*, 2018.

[57] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the European Conference on Computer Systems*, 2015.

[58] Benjamin Wagner, André Kohn, and Thomas Neumann. Self-Tuning Query Scheduling for Analytical Workloads. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2021.

[59] Carl A. Waldspurger and William E. Weihl. Lottery Scheduling: Flexible Proportional-Share Resource Management. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation, OSDI*, 1994.

[60] Yu Wang and Tyler Derr. Tree Decomposed Graph Neural Network. In *Proceedings of the International Conference on Information and Knowledge Management, CIKM*, 2021.

[61] Lex Weaver and Nigel Tao. The Optimal Reward Baseline for Gradient-Based Reinforcement Learning. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 2001.

[62] Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Learning*, 8(3–4), 1992.

[63] Zongheng Yang, Badrish Chandramouli, Chi Wang, Johannes Gehrke, Yinan Li, Umar Farooq Minhas, Per Åke Larson, Donald Kossmann, and Rajeev Acharya. Qd-Tree: Learning Data Layouts for Big Data Analytics. In *Proceedings of the ACM International Conference on Management of Data, SIGMOD*, 2020.

[64] Chi Zhang, Ryan C. Marcus, Anat Kleiman, and Olga Papaemmanouil. Buffer Pool Aware Query Scheduling via Deep Reinforcement Learning. In *Proceedings of the AIDB Workshop @VLDB*, 2020.