Improving DBMS Scheduling Decisions with Accurate Performance Prediction on Concurrent Queries

Ziniu Wu
MIT CSAIL
Cambridge, MA, USA
ziniuw@mit.edu

Markos Markakis
MIT CSAIL
Cambridge, MA, USA
markakis@mit.edu

Chunwei Liu
MIT CSAIL
Cambridge, MA, USA
chunwei@csail.mit.edu

Peter Baile Chen
MIT CSAIL
Cambridge, MA, USA
peterbc@mit.edu

Balakrishnan Narayanaswamy
Amazon Web Services
Santa Clara, CA, USA
muralibn@amazon.com

Tim Kraska
Amazon Web Services,
MIT CSAIL
Cambridge, MA, USA
kraska@mit.edu

Samuel Madden
MIT CSAIL
Cambridge, MA, USA
madden@csail.mit.edu

ABSTRACT

Query scheduling is a critical task that directly impacts query performance in database management systems (DBMS). Deeply integrated schedulers, which require changes to DBMS internals, are usually customized for a specific engine and can take months to implement. In contrast, non-intrusive schedulers make coarse-grained decisions, such as controlling query admission and re-ordering query execution, without requiring modifications to DBMS internals. They require much less engineering effort and can be applied across a wide range of DBMS engines, offering immediate benefits to end users. However, most existing non-intrusive scheduling systems rely on simplified cost models and heuristics that cannot accurately model query interactions under concurrency and different system states, possibly leading to suboptimal scheduling decisions.

This work introduces *IconqSched*, a new, principled non-intrusive scheduler that optimizes the execution order and timing of queries to enhance total end-to-end runtime as experienced by the user — query queuing time plus system runtime. Unlike previous approaches, *IconqSched* features a novel predictor, *Iconq*, which treats the DBMS as a black box and accurately estimates the system runtime of concurrently executed queries under different system states. Using these predictions, *IconqSched* is able to capture system runtime variations across different query mixes and system loads. It then employs a greedy scheduling algorithm to effectively determine which queries to submit and when to submit them. We compare *IconqSched* to other schedulers in terms of end-to-end runtime using realistic workload traces. On Postgres, IconqSched reduces end-to-end runtime by up to 16.5% on average and 33.6% in the tail. Similarly, on Redshift, it reduces end-to-end runtime by up to 14.4% on average and 22.9% in the tail.

PVLDB Reference Format:
Ziniu Wu, Markos Markakis, Chunwei Liu, Peter Baile Chen, Balakrishnan Narayanaswamy, Tim Kraska, and Samuel Madden. Improving DBMS Scheduling Decisions with Accurate Performance Prediction on Concurrent Queries. PVLDB, 18(11): 4185 - 4198, 2025.
doi:10.14778/3749646.3749686

Proceedings of the VLDB Endowment, Vol. 18, No. 11 ISSN 2150-8097.
doi:10.14778/3749646.3749686

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/wuziniu/IconqSched.

1 INTRODUCTION

Query scheduling is a fundamental problem in database management systems (DBMSs) that directly impacts query performance. A scheduler needs to decide when to execute a query (or operators of a query) and how much of each resource (e.g., CPU, memory) to allocate. Query scheduling is challenging because making an optimal decision requires accurately predicting the cost/runtime of executing different operators with access to varying amounts of resources and under different system loads and concurrency levels [59]. Even with accurate predictions, scheduling itself is an NP-complete decision problem [66].

This work focuses on scheduling OLAP queries because they involve accessing large amounts of data and resources. Thus, optimally scheduling each OLAP query individually can lead to more performance gain than OLTP query. Several past efforts have built deeply integrated DBMS schedulers [10, 22, 23, 34, 41, 44, 54, 59, 60] for OLAP queries, which require modifications to the DBMS internals, such as changing the way the system allocates resources at the query operator level. This requirement limits applicability, because the schedulers are tailored to a specific DBMS, making generalization to other engines non-trivial. In addition, it can take months for engineers to implement and verify the performance of such schedulers before they can be used by available to end users. To address these shortcomings, *non-intrusive* schedulers [3, 9, 15, 27, 55, 79] make coarse-grained decisions, such as controlling query admission and reordering query execution, and as such, do not require modifications to DBMS internals. They require much less engineering effort and can potentially have an immediate impact on the end users of almost any DBMS engine [49, 80]. However, most of the existing solutions (e.g., first-in-first-out, shortest-query-first [60],

fair scheduling [21, 25]) are based on simplified cost/runtime models and heuristics that can easily lead to inaccurate predictions and suboptimal scheduling decisions.

In this work, we propose *IconqSched*, an improved non-intrusive scheduler made possible by *Iconq*, a new fine-grained predictor that is able to accurately predict the *system runtime* of concurrent queries (i.e. the time they take from submission to the DBMS until completion). As a result, *IconqSched* can accurately understand query performance under different concurrent query loads and levels of resource availability. Relying on this model, *IconqSched* can schedule the submission time of each query on the host DBMS to maximally improve overall *end-to-end runtime* over the workload, including both system runtime and query queuing time.

Building *IconqSched* poses a major technical challenge: **how to accurately predict the system runtimes of concurrently executing queries?** Building an accurate concurrent query system runtime predictor requires understanding complex query interactions and accounting for different system states (e.g., resource utilization and cache state), which significantly affect query performance. Most existing approaches use heuristics and simplifying assumptions that can lead to substantial estimation errors [1, 3, 4, 18, 19, 71]. A recent learned method [82] uses a graph neural network (GNN) to model data sharing and memory contention among queries. However, it does not model the system state changes nor the impact of the query submission time, often producing inaccurate estimations. Furthermore, it assumes perfect knowledge of future queries to model the query interaction, which is impractical.

To address this challenge, we develop a novel system runtime predictor, *Iconq* (Interaction of concurrent queries), which accurately models complex query interactions and system state changes. *Iconq* first encodes a stream or batch of queries as interaction feature vectors, which can capture various complex query interactions. Then, it uses a bi-directional LSTM model to ingest these interaction feature vectors and predict the system runtime of concurrently executed queries. The LSTM model's internal states implicitly encode the concurrent state (e.g., what queries are executing, what execution stage they are at) and the system state (e.g., resource utilization, data in cache). Crucially, the bi-directional LSTM architecture enables *Iconq* to separately understand the impact of queries submitted *before* and *after* a target query using the forward and backward passes. The forward pass understands the impact of *previous* queries, allowing *IconqSched* to predict the system runtime of an incoming query Q_{n+1}, given the queries $Q_1, ..., Q_n$ already running in the DBMS. In addition, for each running query Q_i, *IconqSched* needs to understand its system runtime change after submitting Q_{n+1}. This can be estimated with the backward pass sending information from Q_{n+1} back to each Q_i.

Our Approach. We develop *IconqSched* as a lightweight proxy layer that can be implemented outside a DBMS instance to help schedule users' queries effectively. In principle, *IconqSched* can be applied to any relational DBMS.

Even given *Iconq*'s accurate predictions, *IconqSched* still needs to make a non-trivial decision about which queries to execute and when. On one hand, we cannot use existing query schedulers [3, 25, 49, 60] as the of-the-shelf solutions because they do not consider fine-grained runtime prediction on concurrent queries. On the other hand, a brute-force approach will enumerate the entire search space of possible decisions, including an exponential number of combinations of candidate queries to submit concurrently and infinitely many candidate timestamps to submit them at. Thus, we develop an efficient greedy scheduling algorithm tailored to *Iconq* to approximate optimal scheduling decisions. At a high level, we develop a new scoring function based on *Iconq*'s predictions to judge the overall end-to-end runtime impact of submitting each candidate query. The scheduling algorithm submits the best query according to the scoring function. Immediately after, it re-evaluates the benefits of the remaining queued queries and iterates the above process to decide on the next best query, and so on. The iteration stops when no queued queries are more beneficial to submit at the current decision timestamp, compared to staying in the queue.

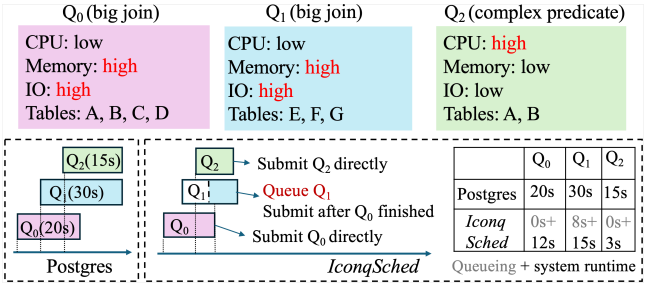


Figure 1: Illustration of *IconqSched* during an execution of the BRAD workload, described in Section 5.1.

In Figure 1, we illustrate *IconqSched* on a simple example with three queries. We summarize each query's resource requirements at the top of the figure, which are not directly observed but can be implicitly captured by *Iconq*. Left alone, the underlying DBMS (PostgreSQL [24]) will execute each query upon arrival, leading to high resource contention: executing Q_0 and Q_1 together results in high memory and I/O utilization and poor performance for both queries. *IconqSched* instead determines that delaying the submission of Q_1 until Q_0 is finished would be beneficial, as it would minimize resource contention. It is worth noting that existing non-intrusive schedulers cannot accurately compute the system runtime impact of deferring a query in order to make this decision. In the meantime, *IconqSched* decides to execute Q_2 as soon as it arrives because the system state created by the execution of Q_0 is beneficial: resource usage is optimized because Q_2 and Q_0 need different resource types, while data sharing is maximized because the two queries access overlapping table sets. As a result, *IconqSched* achieves a 42% reduction in the total end-to-end runtime.

Summary of Results. We evaluate the performance of *IconqSched* as a scheduler for two widely-used DBMSes, PostgreSQL [24] and AWS Redshift [63]. Our target metric is end-to-end runtime—query queueing time plus system runtime. With a few hours of training on query execution history, we can generate improved, non-intrusive scheduling decisions. On PostgreSQL, which has a simpler native scheduler and executor, *IconqSched* has significant room to improve scheduling decisions: 16.2% – 16.5% in the mean and 23.8% – 33.6% in the tail (p-90) on realistic workload traces. On Redshift, which

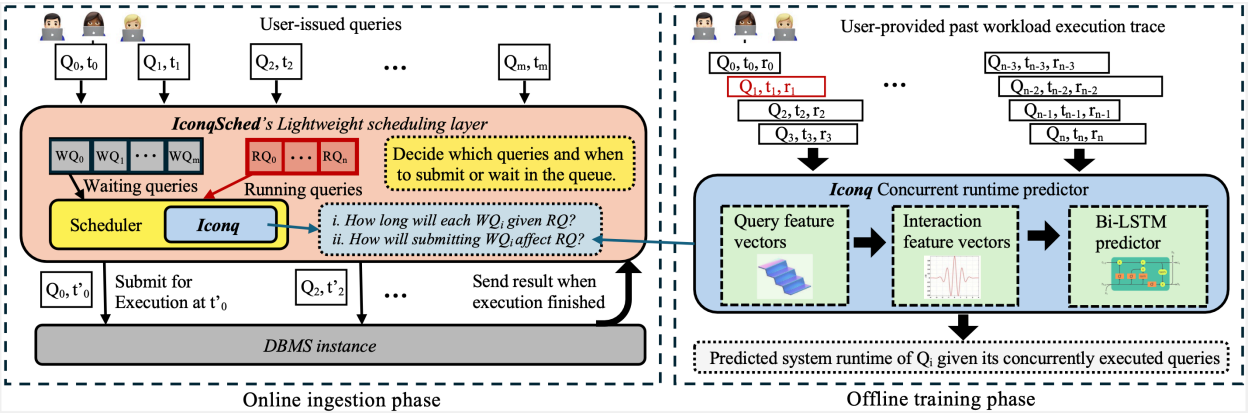


Figure 2: System overview of *IconqSched*.

has a more sophisticated native scheduler (arguably state-of-the-art among commercial DBMS) [50, 60, 73], it is very hard for a non-intrusive scheduler to improve performance further. However, *IconqSched* still improves performance by 10.3% – 14.4% in the mean and 14.9% – 22.9% in the tail.

In addition, we evaluated the performance of our *Iconq* predictor under various settings. *Iconq* achieved more accurate predictions than the baselines on real workload traces by 1.4 × –2.4× in the mean and 2.4 × –4.9× in the tail. We also show that *Iconq* is robust against changing workloads with varying numbers of concurrently executed queries and complex query templates.

Contributions. In summary, we make the following contributions:

- We propose *IconqSched*, a novel non-intrusive query scheduler that can improve DBMS performance (Section 2).
- We design *Iconq* that can accurately predict the system runtime of concurrently executed queries (Section 3). *Iconq* can also independently benefit a wide range of other downstream DBMS tasks that require accurate system runtime estimation, such as query optimization [58] and maintaining SLOs [11, 47].
- We develop a greedy algorithm that uses *Iconq*'s predictions to make effective scheduling decisions (Section 4).
- We conduct comprehensive experiments to evaluate the performance of both *IconqSched* and *Iconq* (Section 5).

2 SYSTEM OVERVIEW

In this work, we tackle the problem of non-intrusively ingesting and scheduling an online stream of OLAP queries $\{(Q_0, t_0), \ldots (Q_m, t_m)\}$, where t_i represents the arrival time of Q_i. We want to determine the optimal time to submit each query to the DBMS in order to minimize total end-to-end runtime as experienced by users, where end-to-end runtime is query queuing time plus system runtime, summed over all Q_i. Since most DBMSes do not preempt queries, this work focuses on non-preemptive scheduling. Our approach can also be naturally generalized to batch scheduling scenarios where all queries are ingested at the same time, or modified to minimize the median or tail query runtime.

Architecture. To address the non-intrusive scheduling problem effectively, *IconqSched* is deployed as a *lightweight scheduling layer* that sits outside the DBMS instance. This proxy layer design is commonly used for many systems [37, 56, 57, 62, 78] and can be generally applied to most DBMS [8]. Users can submit queries to this layer for better scheduling decisions, but we do not require every query to be submitted through *IconqSched*. Instead, we only require information about all currently running queries to be available to *IconqSched*. Many DBMS offer an interface to obtain a list of running queries, which can be used to populate this information.

IconqSched consists of two components, as shown in Figure 2: an **online scheduling algorithm** and a **concurrent query system runtime predictor**, called *Iconq*. *Iconq* is trained offline on the user's executed workload trace. In the online ingestion phase, the scheduling algorithm invokes *Iconq* to make effective decisions. This offline-online two-phase design allows *IconqSched* to offload a heavy amount of computation to the trained system runtime predictor for efficient online decision-making. In the following, we will explain the two components of *IconqSched* at a high level.

Online scheduling algorithm: As shown in Figure 2, when ingesting queries online, *IconqSched* first puts every arriving query in a queue and then uses *Iconq* to decide which queued queries to execute and when. Specifically, given n queries running in the system $RQ = \{RQ_1, \ldots, RQ_n\}$ and m queries waiting in the queue $WQ = \{WQ_1, \ldots, WQ_m\}$ at timestamp t, *IconqSched* needs to decide which queries (if any) to submit to the system. Making the optimal decision is non-trivial because choosing queries to submit is an NP-complete problem [66], and deciding when to submit a query involves considering infinite possible submission timestamps.

Therefore, we design a greedy scheduling algorithm to approximate the optimal solution efficiently. First, instead of considering infinite timestamps, our algorithm will only decide whether to submit a waiting query whenever a new query arrives or a running query finishes. Besides being efficient, this approach is also highly effective, because query arrival and completion highly impact the system and concurrent state, so we need to re-evaluate the benefits of submitting a waiting query or keeping it in the queue. At each decision timestamp, *IconqSched* uses a scoring function to evaluate these benefits accurately. This scoring function uses *Iconq* to predict *i) the system runtime for each* $WQ_i \in WQ$ *given* RQ and *ii) the total system runtime change across* RQ *if* WQ_i *is submitted for execution.* Accordingly, *IconqSched* identifies a set of candidate queries that are more beneficial to submit at the current decision timestamp,

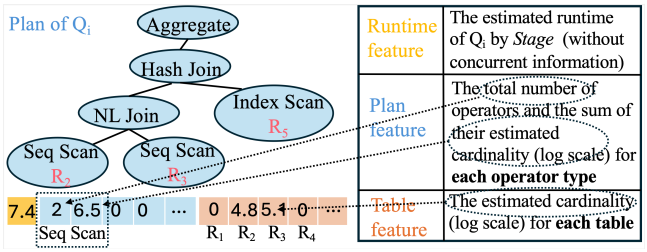


Figure 3: Query feature vector.

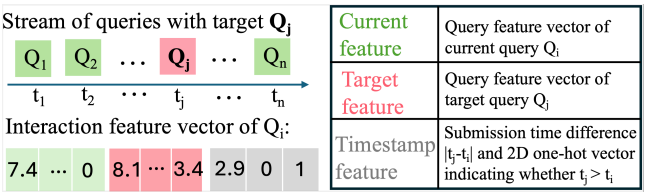


Figure 4: Interaction feature vector with concurrent query execution information.

instead of later. If this set is not empty, it submits the "best" query with the lowest score. Then, it iterates this process to decide the next "best" query until the candidate set is empty. This procedure avoids the exponential search space of candidate query sets. We describe the details of this algorithm in Section 4.

Concurrent system runtime predictor: To make accurate system runtime predictions for concurrent queries, *Iconq* first needs to observe a representative workload trace. In the offline training phase, we first analyze this trace to derive the concurrently executed queries for each query. For example, for Q_1 in Figure 2, $\mathcal{Q}_1 = \{Q_0, Q_1, Q_2, Q_3\}$ are its concurrently executed queries and t_i and r_i are its starting time and system runtime (label). Next, *Iconq* featurizes each query in $\mathcal{Q}_1$ into a *query feature vector* and derives *interaction feature vectors* accordingly. Next, *Iconq* constructs a bi-directional LSTM [29] to iteratively ingest these interaction feature vectors in the forward and backward directions. The LSTM model's *hidden state* vector implicitly encodes the DBMS instance's system and concurrent state. After ingesting all four interaction feature vectors, the final hidden state will be used to predict the system runtime of Q_1. We will discuss the details in Section 3.

After training on the past workload execution history, *Iconq* can accurately estimate the system runtime of an arbitrary query given any concurrent state. Unlike prior work [1, 3, 4, 18, 19, 71, 82], our design allows *Iconq* to capture complex query interactions and understand the changes in resource usage and data-sharing properties of the DBMS instance. Therefore, *Iconq* can generalize to more complex query templates and a larger number of concurrent queries, which are not present in the training dataset (details in Section 5.4). Moreover, as explained in Section 1, the bi-directional LSTM design allows *Iconq* to separately understand not only the impact of the already running queries RQ on the system runtime of WQ_i but also the impact of submitting WQ_i on the system runtime of each RQ_j, which no existing approach can accurately capture.

Iconq treats the underlying DBMS engine as a black box, which generally applies to any DBMS engine with fixed on-premised hardware. For serverless and multi-tenant instances, a user's allocated resources may vary significantly depending on other users' resource usage, which is not observable by our model and cannot be accurately predicted. Exploring the system runtime changes for different hardware types and serverless resources is an active field of research [7, 17, 43, 52] and is beyond the scope of this paper.

3 *ICONQ* RUNTIME PREDICTOR

In this section, we explain the design of *Iconq*. The goal of *Iconq* is to predict the system runtime of a target query Q, given a set of concurrently executing queries $RQ = \{RQ_1, \ldots, RQ_n\}$. At a high level, *Iconq* first featurizes Q and each RQ_i as interaction feature vectors and orders them according to their start timestamps. Then, it uses an LSTM model to process the interaction feature vectors one at a time. It understands the concurrent state and system state changes as the LSTM model's hidden state updates. Finally, the LSTM model is used to predict Q's system runtime. We explain the details of our query featurization in Section 3.1 and predictive model design in Section 3.2. In Section 3.3, we explain the training and inference pipeline to make *Iconq* function inside *IconqSched*.

3.1 Query featurization

We are given a target query Q_j and a set of concurrently executing queries $\mathcal{Q}_j$, which includes all the queries whose execution overlapped at least partially with Q_j. We featurize each query in $\mathcal{Q}_j$ into a *query feature vector* and then derive *interaction feature vectors* to use as inputs to the LSTM model.

Representing a query: Prior work has developed several featurization methods to represent a single query [28, 37, 48, 64, 78]. In this work, we adapt the approach used by *Stage* [73] because of its proven effectiveness and efficiency inside a commercial DBMS. Specifically, we first identify n_p physical operators that can significantly impact a query's system runtime (e.g., scan, merge join, hash). We then initialize 2 *plan features* for each of these operators in query feature vector: a count of how many times the operator appears in the query plan and the sum of estimated rows (i.e., cardinality) processed by this operator, as shown in Figure 3.

One downside with the plan features derived from *Stage*'s method is that they do not contain information about the tables accessed by a query, which is important for understanding memory/buffer pool state and data sharing among concurrently executed queries. Therefore, we also add n_t *table features* to query feature vector, one for each of the n_t largest tables in the database. If a query touches one of these tables, we put an estimate of its cardinality for that table in the corresponding feature, as shown in Figure 3. Finally, we also add a *runtime feature* with the average system runtime of this query without considering concurrent query information, estimated by the *Stage* model, as described in [73]. As shown in Figure 3, the resulting query feature vector consists of the runtime feature, the $2 * n_p$ plan features and the n_t table features. The hyperparameters n_p and n_t are tunable for each workload and database. In our evaluation, we set $n_p = 15$ and $n_t = 20$.

Representing a stream of queries: Given a target query Q_j, for which we wish to estimate the system runtime, let the list $\mathcal{Q}_j = \{Q_1, \ldots, Q_j, \ldots, Q_n\}$ be the queries whose execution overlaps

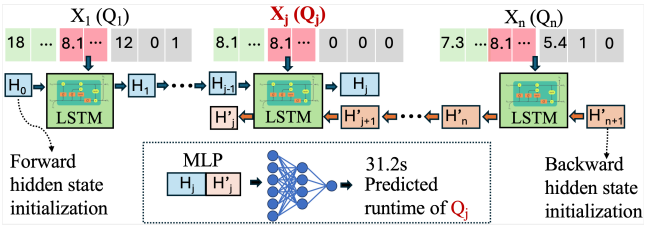


Figure 5: Concurrent query runtime predictor.

with that of Q_j, ordered by their submission times $t_1, \ldots, t_n$. As shown in Figure 4, for each query $Q_i \in \boldsymbol{Q}_j$ (including the target query Q_j), we derive an interaction feature vector with three parts: i) the query feature vector of Q_i, derived from the above approach, ii) the query feature vector of Q_j, and iii) three *timestamp features*. The three timestamp features are: i) the absolute difference $|t_i - t_j|$ between the submission times of Q_i and Q_j, ii) an indicator for whether $t_i < t_j$, and iii) an indicator for whether $t_j < t_i$.

Concatenating the features of Q_i and Q_j helps the model understand their interactions, while the model can use the timestamp features to determine the extent of query execution overlap and infer the execution stage each query is in. We denote the interaction feature vector for Q_i as x_i for each query in $Q_i \in \boldsymbol{Q}_j$ and derive a time series of interaction feature vectors $\{x_1, \ldots, x_n\}$.

3.2 Modeling concurrent query execution

Recall that the goal of *Iconq* is to accurately predict the system runtime of a target query Q_j when it is concurrently executed with queries $\boldsymbol{Q}_j = \{Q_1, \ldots, Q_n\}$, by understanding the impact of concurrency and system state changes (e.g., resource utilization). *Iconq* first uses the above approach to derive a time series of interaction feature vectors $\{x_1, \ldots, x_n\}$. Then, *Iconq* ingests these vectors into a bi-directional LSTM model, which updates its internal hidden state to model the DBMS instance's system state changes.

Long Short-Term Memory (LSTM) [29] models are a type of recurrent neural network (RNN) that are specifically designed to process and remember sequences of data and have been shown to achieve state-of-the-art performance in various fields [32, 40, 65]. We design *Iconq* based on a special type of LSTM, a bi-directional LSTM model [61], which processes the time series of inputs in two directions, forward and backward, and uses two hidden states to understand the dependencies from past and future inputs.

We describe how *Iconq* predicts the system runtime of a target query Q_i given interaction feature vectors of its concurrently executed queries in Figure 5. *Iconq* takes as input the time series $\{x_1, \ldots, x_n\}$ and the index j of the target query. It then traverses the time series in both the forward and backward directions. In the forward pass, *Iconq* initializes the forward hidden state H_0 as a zero matrix and ingests the time series in the forward direction. Upon ingesting each x_i, *Iconq* updates its previous forward hidden state H_{i-1} and produces H_i. After ingesting x_j, the forward hidden state H_j is saved, which captures the impact of the preceding queries $Q_1, \ldots, Q_{j-1}$ on Q_j. In the backward pass, the backward hidden state H'_{n+1} is similarly initialized, and then the interaction feature vectors $\{x_j, ..., x_n\}$ are ingested one at a time *in reverse order*, with the ingestion of x_i producing backward hidden state H'_i. After ingesting x_j, the backward hidden state H'_j is saved. This hidden state captures the impact of the subsequent queries $Q_{j+1}, \ldots, Q_n$ on Q_j. When both the forward and the backward passes are complete, *Iconq* updates its final hidden state as a concatenation of the forward hidden state H_j and the backward hidden state H'_j. It then passes the final hidden state through a multi-layer perception (MLP) to predict the system runtime of Q_j.

Analysis and discussion: The design of *Iconq* as a bidirectional LSTM is uniquely suited for the task of system runtime prediction in three core ways. Whereas, the state-of-the-art deep learning architectures, e.g. transformers [69], lack these properties.

First, **LSTMs explicitly update a hidden state**. This is important because concurrent queries do not impact each other's system runtimes directly; they do so by impacting the state of the DBMS (e.g., the memory/CPU usage or buffer pool state), which in turn can have a positive or negative effect on the system runtime of concurrent queries. Our choice of an LSTM model, where each x_i is submitted one at a time and updates the hidden state to H_i before processing x_{i+1}, naturally parallels this reality. This hidden state implicitly captures the state of the DBMS that it is able to learn from the training data; for example, if the training data includes concurrent queries with large joins that run very slowly, the hidden state can learn to reflect this interaction and model its effect on other concurrent queries. Because our featurization captures the high-level properties of the queries (e.g., cardinalities and plan structure) rather than data-specific features (e.g., column names), this learning is generalizable to unseen feature patterns, such as unseen query templates (see Section 5.4). In contrast, transformer architectures rely on pairwise attention between queries, making it difficult to maintain a compact representation of the evolving DBMS state. Moreover, transformer architectures are good at capturing long-range dependencies, but the time series of interaction feature vectors are much shorter in length (e.g., less than 10).

Second, **LSTMs selectively forget information over time**. This is desirable because parts of the DBMS state are affected by each query for different durations - evaluating a complex predicate will create a temporary demand for CPU cycles, but bringing a table into the buffer pool will have a longer-lasting impact. During its training, the LSTM learns *which* features to "remember" in its hidden state and *for how long*. Concretely, *Iconq* can learn that the features corresponding to some finished operators have little effect on other concurrent queries and discard them from its hidden states. This selective forgetfulness means that the hidden state remains informative even as the time series $\{x_1, ..., x_n\}$ grows longer, allowing *Iconq* to generalize to scenarios with a large number of concurrent queries (see Section 5.4). This observation is similar to the situation where an LSTM trained on short sentences can generalize to a longer corpus of text [29, 33].

Third, **a bidirectional LSTM understands the impact of the past and the future separately**. In a DBMS, not only is the system runtime of a newly-submitted query Q affected by the current state of the DMBS, created by already-running queries RQ (the "past"); Q also itself affects the state of the DBMS once submitted, possibly impacting the system runtime of each query in RQ (from the perspective of which, such impact comes from the "future").

For *IconqSched* to make good scheduling decisions, each of these effects must be explicitly modeled (see Section 3.3). In contrast, transformer architectures cannot explicitly separate these impacts.

3.3 Offline training and Online Inference

In the following, we describe how we train *Iconq* in the offline phase and how we use it to make inferences in the online phase.

Offline training: In this work, we assume that users provide us with a few days of query execution history inside the target DBMS. This is reasonable because we hypothesize that the users will deploy *IconqSched* on an existing workload on a particular DBMS instance. In the case of a new instance, *IconqSched* needs to collect data for a few days before making effective scheduling decisions. We plan to address this "cold-start" problem in future work.

The user-provided training trace needs to contain a list of queries along with their query plans, submission time, and system runtime. First, we train a *Stage* model from these queries to predict the average system runtime per query, without considering the impact of concurrent execution [73], as mentioned in Section 3.1. Then, for each query Q_j, we identify the list $\boldsymbol{Q}_j$ of overlapping queries and compute the interaction feature vectors. Each Q_j and its associated time series of interaction feature vectors are forwarded to the bi-directional LSTM model to predict its system runtime, per Section 3.2. The LSTM model is trained end-to-end using L1 loss and Q-loss [35]. We do assume that the underlying hardware the DBMS runs on does not change frequently – *Iconq* needs to be re-trained if the user switches to a new hardware type. Thus, the current version of *Iconq* is not eligible for serverless settings where virtualization hardware may have a high-performance variability.

Online inference: *Iconq* can be used for a wide range of downstream DBMS tasks that require accurate system runtime estimation, such as optimization [58] and maintaining SLA/SLOs [11, 47]. In *IconqSched*, it is used to support our online scheduler in deciding which queries from the waiting queue $WQ = \{WQ_1, \ldots, WQ_m\}$ to submit for execution and when to submit them (see Section 4). Specifically, *IconqSched* needs to know i) the runtime of WQ_i if it is submitted at timestamp t, given the currently running queries $RQ = \{RQ_1, \ldots, RQ_n\}$; and ii) how the runtime of each running query $RQ_j \in RQ$ will change if WQ_i is submitted.

To answer the first question, we consider WQ_i as the target query and the running queries RQ as its concurrently executed queries. We derive a time series of interaction feature vectors as per Section 3.1 and input them into *Iconq*. *IconqSched only needs to perform the forward pass* on this time series since no query has been submitted after the target WQ_i. Note that the estimated system runtime of WQ_i does not account for any future queries that may be submitted while WQ_i is running. If *IconqSched* decides to submit such queries, their impact on WQ_i will be considered at that time, at which WQ_i will be one of the "running queries". This design enables *IconqSched* to make robust decisions given imperfect information, i.e., without knowing what queries will arrive. For implementation efficiency, we only need to featurize each query once when ingesting it and cache its featurization for later use.

To answer the second question, for each query $RQ_j \in RQ$, we re-predict its system runtime to account for the impact of query WQ_i. Specifically, for each RQ_j, let $\{x_0, \ldots x_n\}$ be its time series of interaction feature vectors. *Iconq* appends another element to this series: the interaction feature vector x_{n+1} for the interaction of WQ_i with the target RQ_j. Then, *Iconq* will input $\{x_0, \ldots x_{n+1}\}$ into the LSTM model to predict the system runtime of RQ_j. WQ_i only affects the backward pass of each RQ_j's system runtime prediction because it is submitted after RQ_j. It is worth noticing that the existing runtime predictors cannot accurately predict this question because they do not have the bi-directional mechanism.

4 SCHEDULING ALGORITHM

In this section we describe how we use *Iconq* to make scheduling decisions when ingesting an online stream of OLAP queries. Specifically, given n queries running on the system $RQ = \{RQ_1, \ldots, RQ_n\}$ and m queries waiting in the queue $WQ = \{WQ_1, \ldots, WQ_m\}$, *IconqSched* needs to decide which queries (if any) in WQ to submit for execution to minimize the total query end-to-end runtime (*e2e-time*), i.e., the sum of queuing time plus system runtime for all queries in RQ and WQ.

Briefly, *IconqSched* invokes the scheduler whenever a query arrives, is submitted, or finishes. At each invocation, *IconqSched* uses *Iconq* to select a set of *candidate queries* from WQ, each of which the model determines would benefit from being submitted now. *IconqSched* then scores each candidate query using a scoring function and submits the one with the highest score. In the following, we will provide details on when to invoke the scheduler, how to select candidate queries, how to score them, and how to optimize the submission of short-running queries. Due to space limitations, we provide the pseudocode of the overall *IconqSched* scheduling algorithm in our technical report [74].

Invoking the scheduler: At each invocation, *IconqSched* decides the best waiting query to submit right now, if any. This leaves the problem of when exactly each invocation of *IconqSched* should occur. A naive approach would invoke the scheduler at equally spaced times (e.g., every $t_{sched} = 5$ seconds). However, this approach has drawbacks. First, it may lead to long queuing times for queries with a system runtime much smaller than t_{sched}. Second, this approach would be computationally wasteful during periods of no significant system state changes To avoid these drawbacks, we instead invoke *IconqSched* based on *events* rather than periodically. In particular, we invoke *IconqSched* i) each time a new query *arrives* in the queue and ii) each time a running query *finishes*.

Selecting candidate queries: At each invocation, for each queued query $WQ_i \in WQ$, *IconqSched* computes the benefit of submitting WQ_i for execution *now*, against submitting it *at one of L future timestamps*, where L (for "lookahead") is a hyperparameter. As described above, *IconqSched* can only submit a query whenever a query arrives or finishes in the future. Recall that *IconqSched* does not assume any knowledge about future *query arrivals*, the prediction of which is an orthogonal field of research [30, 42]. Therefore, we only compute the benefit of submitting WQ_i for execution *now*, against submitting it at one of L future invocations of *IconqSched because of completed query execution*.

We use *Iconq* to predict when each running query in RQ will finish, sort the finish times in increasing order, and take the L soonest ones, $t_1, \ldots, t_L$, where t_l corresponds to the l-th soonest time at which a running query is expected to finish. Thus, *IconqSched*

needs to compare the benefit of submitting WQ_i now (at t) or in the future at any t_l. This benefit has two components: i) $\delta_1(t_l)$, measuring the end-to-end runtime difference of WQ_i submitted at t v.s. t_l, and ii) $\delta_2(t_l)$, measuring the end-to-end runtime difference of each query in RQ will be if WQ_i is submitted at t instead of at t_l.

$$\delta_1(t_l) = \textbf{Iconq}(WQ_i, RQ, t) - (\textbf{Iconq}(WQ_i, RQ', t_l) + t_l - t) \quad (1)$$

$$\delta_2(t_l) = \sum_j \textbf{Iconq}(RQ_j, \{WQ_i\} \cup RQ, t) - \textbf{Iconq}(RQ_j, \{WQ_i\} \cup RQ', t_l) \quad (2)$$

As shown in Equation 1, $\delta_1(t_l)$ is the difference in the predicted end-to-end runtime of WQ_i if submitted at t concurrently with RQ (i.e. system runtime $\textbf{Iconq}(WQ_i, RQ, t)$ plus no queuing time), or at t_l concurrently with RQ' (i.e., system runtime $\textbf{Iconq}(WQ_i, RQ, t_l)$, plus extra queuing time $t_l - t$), where RQ' excludes the queries in RQ that will have finished by t_l. Similarly, $\delta_2(t_l)$ is the sum over all RQ of the differences in their end-to-end runtimes if WQ_i is submitted at t or at t_l. Therefore, $\delta_1(t_l) + \delta_2(t_l) > 0$ for some t_l suggests that submitting WQ_i at t_l is more beneficial than submitting now, so we should keep WQ_i in the waiting queue. Otherwise, $\delta_1(t_l) + \delta_2(t_l) \leq 0$ for all t_l justifies submitting WQ_i now. In this case, we will consider WQ_i as a candidate query for submission. *IconqSched* will iterate the above process for all waiting queries in WQ and derive a selected set of candidate queries.

Our selection criterion is inspired by works using cost-based scheduling (CBS) for maintaining SLOs [9, 55], where they define a criterion to compute the expected benefits of delaying a query. We extend their approach to account for query interactions.

Scoring candidate queries: Submitting all candidate queries together is not optimal because we have not accounted for their interactions *with each other*. For example, submitting any of WQ_1 or WQ_2 now may be beneficial, but after submitting WQ_1, *IconqSched* may find that deferring WQ_2 is better because of their negative interference. Since evaluating an exponential number of query interactions in the candidate query set is expensive, *IconqSched* uses a greedy algorithm to approximate the optimal solution. At a high level, *IconqSched* scores the candidate queries, ranks them, and submits *one query*. Submitting this query will immediately trigger a new invocation of *IconqSched*, at which point the candidate queries will be re-derived and the scoring function re-evaluated.

We design the scoring function of *IconqSched* to minimize the total end-to-end runtime for all RQ and WQ. For each candidate query WQ_i, the score is a sum of three components: i) $\Delta_1(WQ_i)$, measuring how promising the current system state is for WQ_i's execution, ii) $\Delta_2(WQ_i)$, measuring how badly the running queries RQ will be affected by WQ_i, and iii) how long WQ_i has been in the queue for. Smaller scores are better.

$$\Delta_1(WQ_i) = \textbf{Iconq}(WQ_i, RQ, t) - \textbf{Stage}(WQ_i) \quad (3)$$

$$\Delta_2(WQ_i) = \sum_j \textbf{Iconq}(RQ_j, \{WQ_i\} \cup RQ, t) - \textbf{Iconq}(RQ_j, RQ, t) \quad (4)$$

$$score = \Delta_1 + \Delta_2 - \lambda * Queueing\_time(WQ_i) \quad (5)$$

As shown in Equation 3, $\Delta_1(WQ_i)$ is the difference between i) WQ_i's predicted system runtime if submitted at time t with queries RQ also running and ii) WQ_i's average runtime, as predicted by *Stage* (see Section 3.1). We use WQ_i's average runtime to approximate its runtime when submitted at an arbitrary future point. A negative value of $\Delta_1(WQ_i)$ suggests that the current system state has a positive impact on WQ_i that makes its system runtime shorter than average. In Equation 4, $\Delta_2(WQ_i)$ is the sum over all RQ of the differences in their end-to-end runtimes if WQ_i is submitted now (at t) or not at all. Thus, $\Delta_1(WQ_i) + \Delta_2(WQ_i)$ corresponds to the total query end-to-end runtime impact if we submit WQ_i now. Finally, as shown in Equation 5, the overall scoring function also incorporates a penalty for queries that have been queued for too long, controlled by a a hyperparameter λ, in order to prevent a query from starving in the waiting queue. It is possible to modify the scoring function to optimize the median or tail runtime of all queries [6, 36]. We leave this exploration as future work.

The candidate query with the smallest score is submitted for execution, and then we immediately trigger another invocation of *IconqSched*, at which point the candidate queries will be re-derived, the scoring function re-evaluated, and the next best query submitted. This greedy algorithm will iteratively select one query to submit per invocation until some invocation derives an empty candidate set (resulting in at most $|WQ|$ invocations).

Short query optimization: *IconqSched* uses its trained *Stage* predictor to predict the average runtime of a query upon its arrival in the queue. *IconqSched* will directly submit this query if its average runtime is less than τ seconds, before even deriving the candidate set. We do this because scheduling has a limited impact on short-running queries–the runtime of short-running queries is very stable under different system loads and concurrent states because they use very limited resources (prior work made similar observations [2, 3]). Another reason is that *IconqSched* needs to invoke *Iconq* to make scheduling decisions, incurring an additional $10 - 100ms$ of overhead, which can be significant for short-running queries. In our evaluation, we set τ to 5 seconds.

Analysis: Each invocation of *IconqSched* calls *Iconq* $O(L * |RQ| * |WQ|)$ times in a batch. The batch inference significantly accelerates the inference process. Specifically, for each t_l, $l \in [1, L]$, we need to predict the impact of each $WQ_i \in WQ$ on each of the running queries RQ. For most of the OLAP workloads [67, 70], the number of concurrently running queries $|RQ|$ is relatively small (e.g., < 10). Compared to the heuristic-based schedulers [3, 25, 49, 60], *IconqSched* is more principled because it designs a scoring function to optimize the overall end-to-end query runtime. It is worth noticing that we do not consider our scheduling algorithm as a general-purpose standalone component, but rather it should be used in conjunction with an accurate runtime predictor capable of modeling fine-grained interference among concurrent queries.

5 EVALUATION

In this section, we first describe our experimental setup in Section 5.1. Then, we address the following questions:

- How much does our scheduler improve the query performance of existing DBMSs on a practical workload (Section 5.2)?
- How does our scheduler behave with a varied number of clients hitting the system (Section 5.3)?

• What are the accuracy and overhead of our runtime estimator? How robust is it under a changing workload (Section 5.4)?

5.1 Experimental Setup

In this section, we explain our experimental environment, the baselines we compare with, and the workloads.

Environment: We note that *IconqSched* is not tailored to a particular DBMS but can be applied to a wide range of systems. We conduct experiments on two DBMSs: Postgres [24] and Redshift [63].

We chose Postgres because it is one of the most widely used open-source DBMSs with countless applications built upon it [14]. However, Postgres has a very simple scheduling approach that uses the multi-programming level (MPL) to control the maximum number of concurrent queries in an instance and executes queries in a first-in-first-out fashion. Thus, Postgres leaves us room to improve its scheduling decisions in a non-intrusive way. Our experiments use AWS "db.m5.xlarge" Postgres 16.2 instances with 4 vCPUs, 16GB RAM, 1000GB storage, and 2000 provisioned IOPS.

We chose Redshift because it is one of the most widely used commercial data warehouses dedicated to OLAP workloads [14, 60], which are the focus of this work. Redshift contains a state-of-the-art workload manager that can make effective decisions on admission control, scheduling, and resource allocation [50, 60, 73]. Thus, it is very challenging for non-intrusive schedulers to improve Redshift. Our experiment uses AWS "dc2.large" Redshift instances with 1 node, 2 vCPUs, 7.5 GB of RAM, and 160 GB of SSD. We tuned the configurations (e.g., MPL) of both the Postgres and Redshift instances on the workload traces. Our scheduler and all baselines are trained and served on a Linux machine with 40 Intel(R) Xeon(R) Gold 6230 CPUs [31] and 128GB RAM. We implemented *Iconq* in Pytorch [53]. Our bi-LSTM model has 2 layers, a hidden size of 256, an embedding size of 128 and <10 MB storage overhead.

Scheduling Baselines: We compared *IconqSched* to the following non-intrusive query scheduler baselines:

• *Postgres/Redshift*: the original DBMS with tuned knobs.

• *PGM*: the PGM scheduler [49] that estimates the memory usage for each query. The PGM scheduler uses the estimated memory as admission control to keep the total memory usage of concurrently executed queries under the total memory of the instance. Whenever admitting a query would exceed this total memory, it adds the query to a queue and considers the query with the largest predicted memory from the queue as the next query to submit.

• *Qshuffler*: the Qshuffler scheduler [3] uses a simple heuristic-based model to understand query interactions and schedule queries. Specifically, it first clusters all queries into k types and assumes the queries from the same type have the same characteristics. Then, it represents the system state as a vector, counting the number of running queries of each type. At last, it scores all queued queries based on this vector and submits the one with the best score.

We did not compare with intrusive scheduling algorithms [21–23, 34], including RL-based learned methods [41, 44, 59], because they require changes to the execution engine of the underlying DBMS that are impractical to implement in Postgres and Redshift.

Runtime Prediction Baselines: We compared *Iconq* with the following baselines on concurrent query runtime prediction accuracy:

• *Qshuffler*: the runtime predictor used by *Qshuffler* [3].

• *GPredictor*: An ML-based state-of-the-art concurrent runtime predictor [82]. Specifically, it represents the operators in the target query and its concurrently running queries as nodes and their interactions as edges. Then, it uses a graph neural network to propagate this graph and estimate each operator's runtime.

• *Stage*: the staged runtime predictor [73] mentioned in Section 3.1. It only predicts the average runtime of a query without considering any concurrent query information.

• *Function*: expert-designed analytic functions [1, 4, 19, 71], commonly used to predict runtime as a function of I/O, CPU, and memory usage. We combine the merits of existing approaches to derive a comprehensive analytic function. Specifically, this function jointly considers a wide range of query/system features (such as the estimated resource usage of queries, data sharing, and system resource capacities), which are independently modeled in prior works. It also contains several parameters to adjust for estimation errors and weight impacts on different resource types, optimized using multi-dimensional regression on the training data. The details can be found in our technical report [74].

We tuned the hyperparameters of all baselines and our scheduler on a held-out validation trace. We ran experiments three times to evaluate performance and took the average performance.

Workloads: The most straightforward approach to evaluate our scheduler and the baselines is to compare their end-to-end performance improvement over widely-used DBMSs on realistic workloads. We conduct our experiments on the following workloads:

• *CAB* [68] is a well-established cloud analytic benchmark for comparing data warehouse performance on OLAP query workloads. It uses TPC-H benchmark [13] to match the execution characteristics of Snowset [70], which contains real customer's query traces from Snowflake data warehouse but does not include the relevant tables or SQL statements. Specifically, CAB contains a large pool of queries, generated from 22 TPC-H query templates plus one additional insert/delete template. CAB issues queries from the pool to match the typical query submission time patterns in Snowset traces. Then, CAB uses different scales of TPC-H dataset to match the query execution time, CPU time, concurrency level, and data size of Snowset. One limitation of CAB is that the TPC-H dataset is artificially generated and may not contain complex data distribution and correlations like real-world datasets. Therefore, we additionally conduct our experiments on the BRAD workload.

• *BRAD* [78] proposed an analytical workload based on real-world data and realistic queries. Specifically, BRAD scales the IMDB dataset [38] to $160GB$ in total size and generates a set of 1,000 OLAP queries with diverse query templates that resemble the IMDB JOB queries [38]. Afterward, we generate query submission times based on the Snowset trace to derive a practical workload trace that closely matches Snowset's query submission times, completion times, and concurrency levels.

We provide more details, such as query runtime and arrival rate distribution, on these two workloads in our technical report [74]. For each workload, we obtained an execution trace spanning 7 days, with an average of 5,000 queries per day. In our experiments, we use the first 5 days of both traces as the training data to train our models and other baselines, the second-to-last day as the validation trace to tune their hyperparameters, and the last day to evaluate their

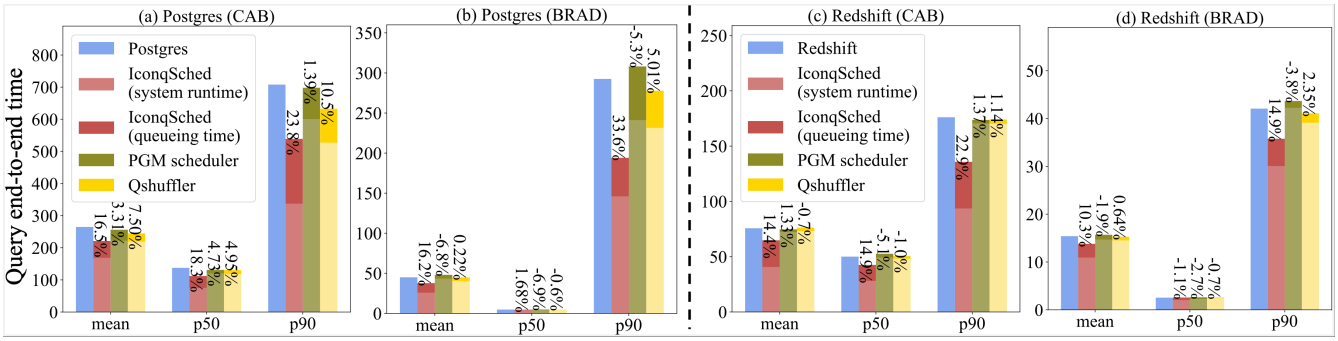


Figure 6: End-to-end query performance of the non-intrusive schedulers over one day of the workload traces. The percentage improvements for the schedulers over the Postgres/Redshift are shown at the top of the bars.

performance. We train one model per workload for both *IconqSched* and the baselines. In addition to these realistic traces, we conduct more adaptivity experiments on simulated workload traces, whose details are described in Section 5.3.

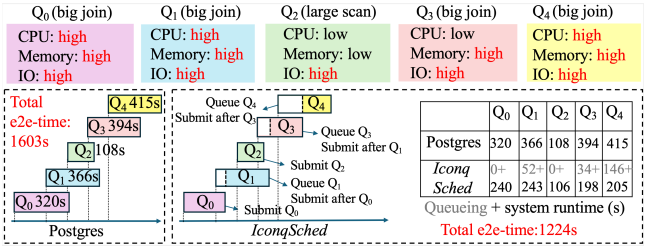
5.2 End-to-end performance

To evaluate *IconqSched* and the baseline systems, we first present their overall performance for executing the *CAB* and *BRAD* workloads in both Postgres and Redshift. Subsequently, we look at specific examples to understand how *IconqSched* enhances performance compared to Postgres. For a fair assessment, we report the end-to-end runtime (*e2e-time*), which includes the scheduler's queueing time plus the system runtime for a query.

5.2.1 Performance on Postgres. The overall performance of *IconqSched* and other baselines executing one day of *CAB* and *BRAD* workload traces are shown in Figure 6. We report the average, median, and tail *e2e-time* of all queries.

Performance for CAB workload: Figure 6-(a) shows the performance of all schedulers when executing the *CAB* workload. *IconqSched* achieves 16.2% average improvement ((*Postgres* − *IconqSched*)/*Postgres*), 18.5% median improvement, and 23.8% tail improvement over Postgres. *IconqSched* has a significant performance gain compared with the other baselines because of our accurate runtime predictor. The *CAB* workload only contains 22 unique TPC-H query templates plus one additional insert/delete query. Queries from the same template exhibit similar characteristics (e.g., scanning the same tables and having the same query plans), making it easier to understand query interactions. Thus, we observe that the non-intrusive scheduler baselines can improve the performance of Postgres. In particular, *Qshuffler* scheduler can cluster queries based on their corresponding query templates and capture interactions between query templates. *Qshuffler* achieves up to 10.5% improvement over Postgres.

Performance for BRAD workload: Figure 6-(b) shows the performance of all schedulers when executing the *BRAD* workload. *IconqSched* achieves a 16.2% average improvement and 33.6% on all queries. Different from the CAB workload, other baselines barely outperform Postgres because the complicated query interactions in the workload are hard to model with heuristics. On the p-50 metric of all queries, roughly all baselines achieve the same performance because more than 50% of queries are short-running, which use a small amount of resources, and are not sensitive to the different concurrent states of the system. Thus, no scheduler is likely to improve the performance of these queries. Recent literature has made similar observations [2, 3]. In contrast, when considering just long-running queries (those with p-90 runtime), *IconqSched* can achieve a 33.6% improvement over Postgres because of its accurate modeling of concurrent query performance and effective scheduling decisions. Overall, these results show that *IconqSched* can re-arrange query execution order to significantly speed up the overall workload execution. We provide detailed examples of how *IconqSched* improves Postgres performance in Section 5.2.3.



	Q0	Q1	Q2	Q3	Q4
Postgres	320	366	108	394	415
Iconq Sched	0+ 240	52+ 243	0+ 106	34+ 198	146+ 205

Figure 7: (Example 1) Delaying the execution of some queries can improve query performance.

5.2.2 Performance on Redshift. We executed these two workloads in Redshift. On *CAB* workload (Figure 6-(c)), *IconqSched* achieves a 14.4% average and 22.9% tail improvement over Redshift on all queries. However, the other baselines do not improve over Redshift because Redshift already employs a state-of-the-art scheduler [50, 60, 73], making it challenging to improve its performance with a non-intrusive scheduler.

We observe similar results on the *BRAD* workload. As shown in Figure 6-(d), *IconqSched* achieves a 10.3% average, and 14.9% tail improvement over Redshift. This improvement is slightly lower than that on *CAB* workload because *BRAD* has a lighter query load with more short-running queries. As we explained earlier, short-running queries are hard for *IconqSched* and other baselines to improve, which also explains why *IconqSched* does not improve the median *BRAD* query *e2e-time* in Figure 6-(d).

5.2.3 Example Scenarios. In the following, we provide two examples derived from real executions of the BRAD workload in Postgres

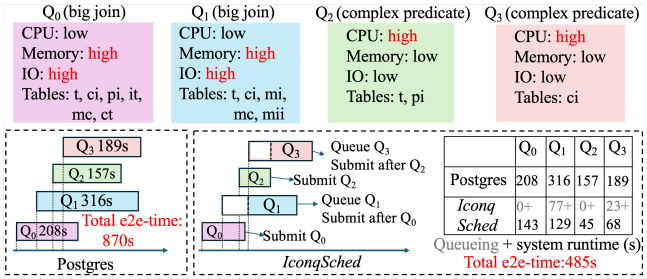


Figure 8: (Example 2) Optimizing resource usage and data sharing can improve query performance.

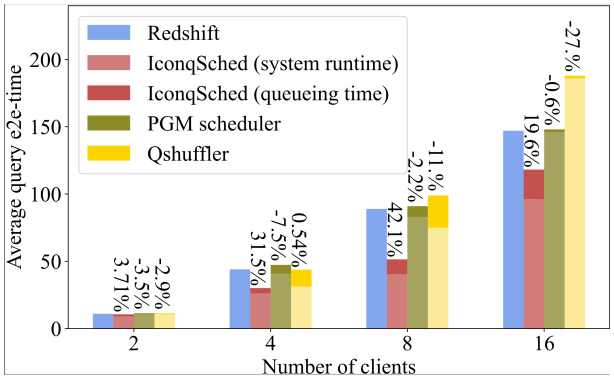


Figure 9: Performance of different schedulers with varied number of clients on executing BRAD queries in Postgres.

to explain the performance gain, demonstrating *IconqSched*'s ability to avoid "bad" and seek "good" concurrent executions.

First, sometimes executing queries sequentially can be more efficient than executing them concurrently. For instance, executing two memory-intensive queries simultaneously can cause significant memory contention and even lead to disk spills, which greatly slows down the joins in both queries. Figure 7 provides a concrete example involving five queries, $Q_0, \ldots, Q_4$, executed within our workload. The legend at the top outlines each query's resource consumption characteristics with different colors. These characteristics are manually derived for the ease of understanding the example but they are not observable by *IconqSched* or other baselines. The left panel shows their default execution in Postgres, where these queries are run concurrently, resulting in high resource contention and a long total *e2e-time* of 1603*s*.

The right panel of Figure 7 shows *IconqSched*'s scheduling and execution decisions. *Iconq* accurately predicts the runtime of these queries both in sequential and concurrent executions, identifying the benefits of delaying the execution of certain queries. As a result, when *IconqSched* processes these queries, it decides to queue Q_1 and submit it after Q_0 is completed, queue Q_3 and submit it after Q_1, and queue Q_4 to be submitted after Q_3 finishes. This approach minimizes resource contention throughout the execution, thereby reducing the total *e2e-time* to 1224*s*.

It's important to note that other scheduling methods, such as PGM and Qshuffler, lack the precise and detailed runtime prediction capabilities necessary to differentiate between sequential and concurrent execution of these queries.

Second, optimizing different resource usage and data sharing can improve query performance. An on-premises DBMS cluster contains a fixed number of different resource types (e.g., memory, CPU, I/O, bandwidth), and a query may use different amounts of each resource. For example, executing two queries with large joins may use up most of the memory, but the system may still efficiently process CPU-intensive queries (e.g., with complex filter predicates). As in previous example, Figure 8 shows a example of four queries $Q_0, \ldots, Q_3$ in our workload executed on Postgres. Q_0 and Q_1 contain I/O- and memory-intensive big joins while Q_2 and Q_3 are CPU-intensive queries with complex filter predicates. Postgres executes Q_0, Q_1, Q_2, and Q_3 concurrently, leading to high resource contention and a long total *e2e-time* of 870*s*.

In contrast, *IconqSched* can accurately model the interactions of these four queries and understand that executing Q_0 and Q_1 together is inefficient. Thus, *IconqSched* decides to queue Q_1 until Q_0 finishes to avoid memory contention. When Q_2 arrives, *IconqSched* identifies the benefits of running Q_2 concurrently with Q_0, since they use different resources but access the same set of tables. Thus, *IconqSched* decides to submit it directly. Similarly, *IconqSched* understands that running Q_2 and Q_3 together is inefficient and decides to queue Q_3 and submit it after Q_2 finishes. Therefore, by using *Iconq*'s fine-grained concurrent runtime predictions, *IconqSched* can implicitly optimize different types of resource usage and leverage data sharing. It is worth noting that instead of directly estimating each query's CPU or memory usage, *Iconq* implicitly captures these resource demands by accurately predicting the query runtime under different system states.

5.3 Adaptivity Analysis

To better understand the robustness of *IconqSched* in various situations, we conducted adaptivity experiments with a varied number of clients. Our experimental setting is similar to many prior works [25, 49] in that we simulate k clients in parallel. Each client continuously issues queries in a closed loop to the same DBMS (Postgres or Redshift) instance. Specifically, each client randomly draws a query from the CAB/BRAD query pool, submits it to the DBMS instance, waits for it to finish, and immediately submits the next query. We chose the number of clients to be $k = 2, 4, 8, 16$, corresponding to a light load, an average load, a heavy load, and an extremely overloaded system state of the practical workload trace. We ran each experiment for 10 hours. Figure 9 shows the average query runtime for different numbers of clients running BRAD queries on Postgres. Due to space limitations, we put similar results on CAB queries and Redshift in our technical report [74]. For $k = 2$, *IconqSched* performs similarly to Postgres because there are not enough queries in the waiting queue for *IconqSched* to select a query to submit that positively interacts with the running queries.

IconqSched's performance improvement over Postgres increased significantly from two to eight clients because, with more clients, more queries are waiting in the queue, and our scheduler has a better opportunity to submit optimal queries at the optimal time. Moreover, with a median and heavy system load, cleverly managing system resource usage can lead to significant improvement in execution time. The performance improvement for 16 clients

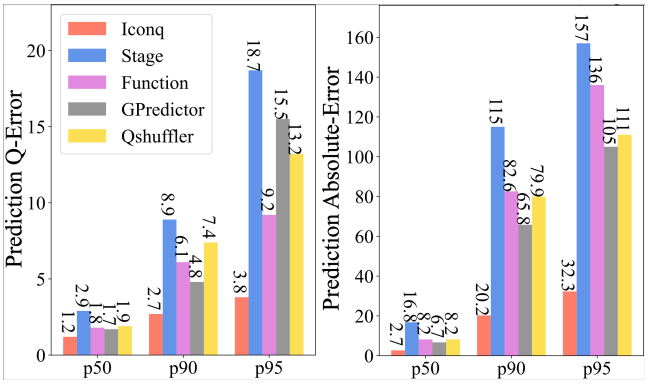


Figure 10: Runtime predictors' performance on concurrent queries of BRAD workload in Postgres.

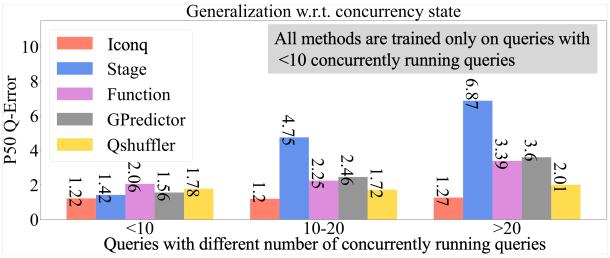


Figure 11: Runtime predictors' generalizability to queries executed under heavier concurrency state.

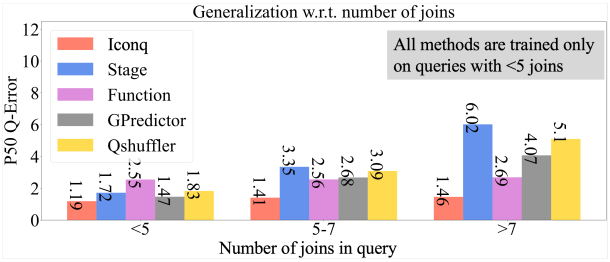


Figure 12: Runtime predictors' generalizability to queries with more complex templates.

(19.6%) is not as significant as for four or eight clients. Simulating with 16 concurrent clients severely overloaded the Postgres instance, with more than 10% of the queries timing out (> 1000s) in the original Postgres execution. Because of the extremely heavy load, *IconqSched* frequently queues queries to offload the instance. As a result, many queries are queued for a long time and have to be submitted due to the starvation penalty (Equation 5). These queries are not necessarily the optimal queries to submit at the optimal time. Therefore, we observe a decrease in performance gain for *IconqSched* in the case of 16 clients. We plan to explore how to address this issue in future work. We did not experiment with more than 16 clients because $k = 16$ already clearly exceeded the capacity of the Postgres instance.

Instead, they can sometimes increase the queueing time. As a result, these two baselines frequently make suboptimal decisions and sometimes perform worse than Postgres.

5.4 Performance of Runtime Predictors

In this section, we evaluate *Iconq* against the baselines mentioned in Section 5.1. We first report the overall performance on the BRAD workload executed in Postgres and then evaluate the generalizability of each method. Due to space limitation, we do not report similar performance trends on the CAB workload and Redshift.

Overall performance: Following the convention of prior works [26, 51, 73, 75, 76], we judge the prediction accuracy of each method at p50, p90, and p95, using two well-recognized metrics: i) *absolute-error*, which measures the absolute difference between predicted runtime and true runtime: $|predicted - true|$; and ii) *Q-error*, which measures their relative/fractional difference: $max\{predicted/true, true/predicted\}$. On both metrics, lower is better, with 0 and 1 being optimal, respectively.

Figure 10 shows the performance of *Iconq* and the baselines on all concurrent queries of the testing workload (~ 5000 queries). *Iconq* achieves significantly better performance on all metrics. We observe a $1.5x - 2.4x$ and $2.4x - 4.9x$ difference on *Q-error* for median and tail (p-90), respectively. Similarly, on *absolute-error*, we observe a $2.5x - 6.2x$ improvement on the median and $3.3x - 4.9x$ improvement on the tail. *Stage* performs the worst because it only estimates the average query runtime without considering concurrent query information. We also compared other aspects of runtime predictors, such as the training time, model size, and inference speed, which can be found in our technical report [74].

Robustness and generalizability: In addition, we conducted two generalizability experiments to demonstrate the robustness of *Iconq*.

First, we evaluate these runtime predictors' generalizability to heavier concurrency states, i.e., queries with more concurrently running queries. Specifically, we train these predictors on queries with < 10 concurrently running queries and test on queries with $10 - 20$ and > 20 concurrently running queries. Figure 11 shows that *Iconq* remains very accurate even for unseen heavier concurrency states. This is because *Iconq* uses the hidden state of a bi-directional LSTM to represent concurrent and system state changes that can better extrapolate to an unseen concurrency state. We provide more detailed reasons for *Iconq*'s robustness in Section 3.2. We also find *Qshuffler*'s performance relatively stable because its heuristics are robust against changing concurrent states. However, other baselines experience a significant decrease in accuracy when tested on heavier concurrent states.

Second, we evaluate these methods' generalizability to more complex query templates (e.g., more table joins). We train them on queries with < 5 table joins and test on queries with $5 - 7$ and > 7 table joins. Figure 11 shows that *Iconq* remains very accurate even for harder query templates because the model feature captures the inherent properties of the queries (e.g., average runtime, cardinalities, and plan structure) that can be better extrapolated to unseen query templates. We also find *Function*'s performance relatively stable because this simple method does not encode any information about the query template. However, other baselines experience a significant decrease in accuracy when tested on harder templates. *GPredictor* featurizes the query plan as a graph. For queries with more joins, their corresponding graphs get much larger. We hypothesize that the trained GCN model may overfit small graphs and

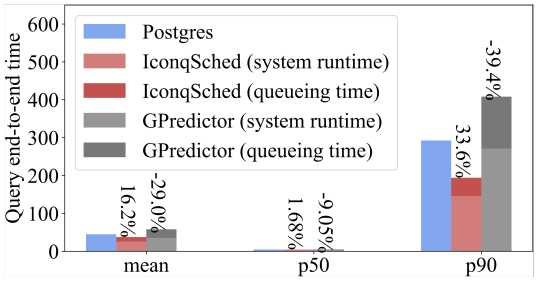


Figure 13: Ablation study: end-to-end query performance of BRAD workload on Postgres for different predictors.

extrapolate poorly to larger ones. *Qshuffler* uses a simple clustering model to group similar queries into clusters. For more complex queries, the trained model cannot easily cluster them and will lead to very poor performance.

Ablation study: To further demonstrate the superiority of *Iconq*, we conduct an ablation study on the *BRAD* workload. We compare *IconqSched*'s end-to-end performance against a baseline where we keep our scheduling algorithm, but swap the runtime predictor to use *GPredictor* rather than using *Iconq*. Figure 13 shows that this baseline leads to much worse end-to-end query performance, even worse than Postgres. As discussed earlier, this is because the prediction accuracy of *Iconq* is much higher than that of *GPredictor*. Moreover, our scheduling algorithm needs predictions on *how submitting a query impacts the execution time of existing queries*, which *Iconq* can provide through its bi-directional message passing design. *GPredictor* does not have such a mechanism to make a reasonable prediction, leading to poor end-to-end performance.

6 RELATED WORK

To highlight this work's novelty, we review the related work in runtime prediction and scheduling for OLAP queries.

Query runtime prediction: Much of the prior work focuses on predicting the runtime of a single query without considering its concurrent state. Traditional methods typically rely on manually-derived heuristics and statistical models to analyze relational operators [5, 20, 39, 72]. On the other hand, machine learning models can predict the runtime of a query with greater accuracy than traditional approaches, but they experience high inference latency [28, 45, 46, 48, 64, 73, 77, 81].

Predicting the runtime of a target query given its concurrent state (i.e., the system state and the concurrently executed queries) is a much harder problem than single query prediction. Most of the existing methods are based on simple heuristics. Analytic functions/models [1, 4, 19, 71] are commonly used to predict concurrent runtime as a function of I/O, CPU, and memory usage. BAL [18] uses buffer access latency to capture the effect of disk I/O. Qshuffler [3] clusters queries into different types and approximates query interactions between these types. Such simple approaches cannot accurately predict the runtime of queries in practical workloads with complex query interactions and system state changes. A recent work [82] proposed using a graph neural network (GNN) to understand complex interactions among concurrently executing queries. It has been shown to be significantly more accurate than the traditional methods. However, this GNN-based approach is not well suited to our work. First, it assumes perfect knowledge of queries that arrive after the target query, which is impractical in our setting. Second, it cannot understand system state changes at different timestamps, which can produce inaccurate predictions.

Query scheduling: Traditional, deeply-integrated schedulers use simple heuristics or analytical models to estimate the needs of each query/operator and allocate appropriate resources [21–23, 34, 60]. Recent approaches [41, 44, 59] propose representing query plans as graphs and using deep reinforcement learning (RL) to make fine-grained scheduling decisions. They can cleverly allocate appropriate resources to different query operators, which significantly improves over traditional methods. However, these RL-based methods require extensive training and exploration inside the DBMS instance before achieving satisfactory performance, which may not be affordable for many users. Many non-intrusive approaches also use simple rules or heuristics to schedule queries and control query admission, such as first-in-first-out, shortest-query-first [60], and fair scheduling [21, 25]. Others use analytic models and regression models to estimate query cost/runtime and resource consumption in order to control the multi-programming level (MPL) and perform query admission [9, 12, 15, 16, 27, 49, 55, 79]. QShuffler [2, 3] uses heuristics to approximate query interactions and leverages them to schedule queries that can minimize the overall runtime. These heuristic-based non-intrusive scheduling algorithms can easily make sub-optimal decisions. In contrast, *IconqSched* leverages its fine-grained runtime predictions to accurately understand query performance under different concurrent query loads to decide which queries to execute and when to execute them.

7 CONCLUSION

This work introduces *IconqSched*, a non-intrusive scheduler designed to enhance the performance of a DBMS by effectively rearranging the execution order and submission time of queries. *IconqSched* employs a novel fine-grained predictor, *Iconq*, which treats the DBMS instance as a black box and predicts the system runtime of concurrently executed queries on that instance. Experiments demonstrate that *IconqSched* can significantly improve the performance of Postgres and Redshift on a realistic OLAP workload. Although not a focus of this work, we believe *Iconq* has significant additional benefits in other DBMS tasks, such as query optimization and maintaining service level objectives (SLOs).

ACKNOWLEDGMENTS

This work was supported by the DARPA ASKEM program (award HR00112220042), the ARPA-H Biomedical Data Fabric project, the MIT DSAIL Project, and grants from Liberty Mutual and Google. This research was also sponsored by the United States Air Force Research Laboratory and the Department of the 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 Department of the 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] Mumtaz Ahmad, Ashraf Aboulnaga, and Shivnath Babu. 2009. Query interactions in database workloads. In *Proceedings of the Second International Workshop on Testing Database Systems*. 1–6.

[2] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. 2008. Modeling and exploiting query interactions in database systems. In *Proceedings of the 17th ACM conference on Information and knowledge management*. 183–192.

[3] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. 2011. Interaction-aware scheduling of report-generation workloads. *The VLDB Journal* 20 (2011), 589–615.

[4] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. 2011. Predicting completion times of batch query workloads using interaction-aware models and simulation. In *Proceedings of the 14th International Conference on Extending Database Technology*. 449–460.

[5] Mert Akdere, Ugur Çetintemel, Matteo Riondato, Eli Upfal, and Stanley B. Zdonik. 2012. Learning-based Query Performance Modeling and Prediction. In *IEEE 28th International Conference on Data Engineering (ICDE 2012), Washington, DC, USA (Arlington, Virginia), 1-5 April, 2012*, Anastasios Kementsietsidis and Marcos Antonio Vaz Salles (Eds.). IEEE Computer Society, 390–401. https://doi.org/10.1109/ICDE.2012.64

[6] Aharon Ben-Tal, Laurent El Ghaoui, and Arkadi Nemirovski. 2009. *Robust optimization*. Vol. 28. Princeton university press.

[7] Christian Böhm. 2000. A cost model for query processing in high dimensional data spaces. *ACM Transactions on Database Systems (TODS)* 25, 2 (2000), 129–178.

[8] Matthew Butrovich, Karthik Ramanathan, John Rollinson, Wan Shen Lim, William Zhang, Justine Sherry, and Andrew Pavlo. 2023. Tigger: A database proxy that bounces with user-bypass. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3335–3348.

[9] Yun Chi, Hakan Hacígümüş, Wang-Pin Hsiung, and Jeffrey F Naughton. 2013. Distribution-based query scheduling. *Proceedings of the VLDB Endowment* 6, 9 (2013), 673–684.

[10] Yun Chi, Hyun Jin Moon, and Hakan Hacigümüş. 2011. iCBS: incremental cost-based scheduling under piecewise linear SLAs. *Proc. VLDB Endow.* 4, 9 (Jun 2011), 563–574. https://doi.org/10.14778/2002938.2002942

[11] Yun Chi, Hyun Jin Moon, Hakan Hacigümüs, and Jun'ichi Tatemura. 2011. SLA-tree: a framework for efficiently supporting SLA-based decisions in cloud computing. In *EDBT 2011, 14th International Conference on Extending Database Technology, Uppsala, Sweden, March 21-24, 2011, Proceedings*, Anastasia Ailamaki, Sihem Amer-Yahia, Jignesh M. Patel, Tore Risch, Pierre Senellart, and Julia Stoyanovich (Eds.). ACM, 129–140. https://doi.org/10.1145/1951365.1951383

[12] Microsoft Corp. 2017. Managing SQL server workloads with resource governor. (2017).

[13] Transaction Processing Performance Council. 2014. *TPC Benchmark™ H (Decision Support) Specification Revision 2.17.3.* Retrieved July 12, 2025 from http://www.tpc.org/tpch/

[14] DB-Engines. 2024. DB-Engines Ranking - popularity ranking of database management systems. Retrieved July 12, 2025 from https://db-engines.com/en/ranking

[15] Peter J. Denning. 1980. Working sets past and present. *IEEE Transactions on Software engineering* 1 (1980), 64–84.

[16] Peter J Denning, Kevin C Kahn, Jacques Leroudier, Dominique Potier, and Rajan Suri. 1976. Optimal multiprogramming. *Acta Informatica* 7 (1976), 197–216.

[17] Weimin Du, Ravi Krishnamurthy, and Ming-Chien Shan. 1992. Query optimization in a heterogeneous dbms. In *VLDB*, Vol. 92. 277–291.

[18] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*. 337–348.

[19] Jennie Duggan, Olga Papaemmanouil, Ugur Cetintemel, and Eli Upfal. 2014. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction.. In *EDBT*. 109–120.

[20] Jennie Duggan, Olga Papaemmanouil, Ugur Çetintemel, and Eli Upfal. 2014. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy (Eds.). OpenProceedings.org, 109–120. https://doi.org/10.5441/002/EDBT.2014.11

[21] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX symposium on networked systems design and implementation (NSDI 11)*.

[22] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. *ACM SIGCOMM Computer Communication Review* 44, 4 (2014), 455–466.

[23] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. {GRAPHENE}: Packing and {Dependency-Aware} scheduling for {Data-Parallel} clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 81–97.

[24] PostgreSQL Global Development Group. 2024. *PostgreSQL*. Retrieved July 12, 2025 from https://www.postgresql.org/docs/release/16.1/ Database management system.

[25] Chetan Gupta, Abhay Mehta, Song Wang, and Umesh Dayal. 2009. Fair, effective, efficient and differentiated scheduling in an enterprise data warehouse. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 696–707.

[26] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, et al. 2022. Cardinality estimation in dbms: A comprehensive benchmark evaluation. *VLDB* (2022).

[27] Jayant R Haritsa, Michael J Canrey, and Miron Livny. 1993. Value-based scheduling in real-time database systems. *The VLDB Journal 2* (1993), 117–152.

[28] Benjamin Hilprecht and Carsten Binnig. 2022. Zero-Shot Cost Models for Out-of-the-box Learned Cost Prediction. *Proc. VLDB Endow.* 15, 11 (2022), 2361–2374. https://www.vldb.org/pvldb/vol15/p2361-hilprecht.pdf

[29] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[30] Hanxian Huang, Tarique Siddiqui, Rana Alotaibi, Carlo Curino, Jyoti Leeka, Alekh Jindal, Jishen Zhao, Jesús Camacho-Rodríguez, and Yuanyuan Tian. 2024. Sibyl: Forecasting Time-Evolving Query Workloads. *Proc. ACM Manag. Data* 2, 1, Article 53 (Mar 2024), 27 pages. https://doi.org/10.1145/3639308

[31] Intel Corporation. 2019. *Intel Xeon Gold 6230 CPU*. Retrieved July 12, 2025 from https://ark.intel.com/content/www/us/en/ark/products/192437/intel-xeon-gold-6230-processor-27-5m-cache-2-10-ghz.html

[32] Rafal Jozefowicz, Wojciech Zaremba, and Ilya Sutskever. 2015. An empirical exploration of recurrent network architectures. In *International conference on machine learning*. PMLR, 2342–2350.

[33] Andrej Karpathy, Justin Johnson, and Li Fei-Fei. 2015. Visualizing and understanding recurrent networks. *arXiv preprint arXiv:1506.02078* (2015).

[34] James E Kelley Jr and Morgan R Walker. 1959. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*. 160–173.

[35] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned cardinalities: Estimating correlated joins with deep learning. *arXiv preprint arXiv:1809.00677* (2018).

[36] Roger Koenker. 2005. *Quantile regression*. Vol. 38. Cambridge university press.

[37] Tim Kraska, Tianyu Li, Samuel Madden, Markos Markakis, Amadou Ngom, Ziniu Wu, and Geoffrey X Yu. 2023. Check out the big brain on BRAD: simplifying cloud data processing with learned automated data meshes. *Proceedings of the VLDB Endowment* 16, 11 (2023), 3293–3301.

[38] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[39] Jiexing Li, Arnd Christian König, Vivek R. Narasayya, and Surajit Chaudhuri. 2012. Robust Estimation of Resource Consumption for SQL Queries using Statistical Techniques. *Proc. VLDB Endow.* 5, 11 (2012), 1555–1566. https://doi.org/10.14778/2350229.2350269

[40] Zachary C Lipton, John Berkowitz, and Charles Elkan. 2015. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019* (2015).

[41] Chenghao Lyu, Qi Fan, Fei Song, Arnab Sinha, Yanlei Diao, Wei Chen, Li Ma, Yihui Feng, Yaliang Li, Kai Zeng, et al. 2022. Fine-grained modeling and optimization for intelligent resource management in big data processing. *arXiv preprint arXiv:2207.02026* (2022).

[42] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J. Gordon. 2018. Query-based Workload Forecasting for Self-Driving Database Management Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 631–645. https://doi.org/10.1145/3183713.3196908

[43] Michael V Mannino, Paicheng Chu, and Thomas Sager. 1988. Statistical profile estimation in database systems. *ACM Computing Surveys (CSUR)* 20, 3 (1988), 191–221.

[44] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM special interest group on data communication*. 270–288.

[45] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. China. https://doi.org/10.1145/3448016.3452838 Award: 'best paper award'.

[46] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *PVLDB* 12, 11 (2019), 1705–1718.

[47] Ryan Marcus and Olga Papaemmanouil. 2016. WiSeDB: A Learning-based Workload Management Advisor for Cloud Databases. *Proc. VLDB Endow.* 9, 10 (2016), 780–791. https://doi.org/10.14778/2977797.2977804

[48] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746. https://doi.org/10.14778/3342263.3342646

[49] Abhay Mehta, Chetan Gupta, and Umeshwar Dayal. 2008. BI batch manager: a system for managing batch workloads on enterprise data-warehouses. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. 640–651.

[50] Vikram Nathan, Vikramank Singh, Zhengchun Liu, Mohammad Rahman, Andreas Kipf, Dominik Horn, Davide Pagano, Gaurav Saxena, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Intelligent Scaling in Amazon Redshift. In *Companion of the 2024 International Conference on Management of Data*. 269–279.

[51] Parimarjan Negi, Ziniu Wu, Andreas Kipf, Nesime Tatbul, Ryan Marcus, Sam Madden, Tim Kraska, and Mohammad Alizadeh. 2023. Robust query driven cardinality estimation under changing workloads. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1520–1533.

[52] Parimarjan Negi, Ziniu Wu, Arash Nasr-Esfahany, Harsha Sharma, Mohammad Alizadeh, Tim Kraska, and Sam Madden. 2024. OS Pre-trained Transformer: Predicting Query Latencies across Changing System Contexts.

[53] Adam Paszke, Sam Gross, Francesco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Srinath Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems*, Vol. 32. https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf

[54] J.M. Peha and F.A. Tobagi. 1996. Cost-based scheduling and dropping algorithms to support integrated services. *IEEE Transactions on Communications* 44, 2 (1996), 192–202. https://doi.org/10.1109/26.486612

[55] Jon Michael Peha. 1991. *Scheduling and dropping algorithms to support integrated services in packet-switched networks*. Stanford University.

[56] PgBouncer. [n.d.]. *PgBouncer - lightweight connection pooler for PostgreSQL*. Retrieved July 12, 2025 from https://www.pgbouncer.org Database management system.

[57] ProxySQL. [n.d.]. *ProxySQL - A High Performance Open Source MySQL Proxy*. Retrieved July 12, 2025 from https://proxysql.com Database management system.

[58] Prasan Roy, Srinivasan Seshadri, S Sudarshan, and Siddhesh Bhobe. 2000. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*. 249–260.

[59] 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*. 1228–1242.

[60] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan (Murali) Narayanaswamy. 2023. Auto-WLM: Machine Learning Enhanced Workload Management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*. ACM, 225–237. https://doi.org/10.1145/3555041.3589677

[61] Mike Schuster and Kuldip K Paliwal. 1997. Bidirectional recurrent neural networks. *IEEE transactions on Signal Processing* 45, 11 (1997), 2673–2681.

[62] Amazon Web Services. [n.d.]. *Amazon RDS Proxy | Highly Available Database Proxy*. Retrieved July 12, 2025 from https://aws.amazon.com/rds/proxy/ Database management system.

[63] Amazon Web Services. 2024. *Amazon Redshift*. Retrieved July 12, 2025 from https://aws.amazon.com/redshift/

[64] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319. https://doi.org/10.14778/3368289.3368296

[65] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[66] Jeffrey D. Ullman. 1975. NP-complete scheduling problems. *Journal of Computer and System sciences* 10, 3 (1975), 384–393.

[67] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil Eknath Vaidya, Wenjian Dong, Murali Narayanaswamy, Zhengchun Liu, Gaurav Saxena, Andreas Kipf, and Tim Kraska. 2024. Why TPC is not enough: An analysis of the Amazon Redshift fleet. (2024).

[68] Alexander Van Renen and Viktor Leis. 2023. Cloud analytics benchmark. *Proceedings of the VLDB Endowment* 16, 6 (2023), 1413–1425.

[69] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[70] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI '20)*. 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati

[71] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F Naughton. 2013. Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment* 6, 10 (2013), 925–936.

[72] Wentao Wu, Yun Chi, Shenghuo Zhu, Jun'ichi Tatemura, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Predicting query execution time: Are optimizer cost models really unusable?. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 1081–1092. https://doi.org/10.1109/ICDE.2013.6544899

[73] Ziniu Wu, Ryan Marcus, Zhengchun Liu, Parimarjan Negi, Vikram Nathan, Pascal Pfeil, Gaurav Saxena, Mohammad Rahman, Balakrishnan Narayanaswamy, and Tim Kraska. 2024. Stage: Query Execution Time Prediction in Amazon Redshift. In *SIGMOD*. 280–294.

[74] Ziniu Wu, Markos Markakis, Chunwei Liu, Peter Baile Chen, Balakrishnan Narayanaswamy, Tim Kraska, and Samuel Madden. 2025. Improving DBMS Scheduling Decisions with Fine-grained Performance Prediction on Concurrent Queries – Extended. *arXiv preprint arXiv:2501.16256* (2025).

[75] Ziniu Wu, Parimarjan Negi, Mohammad Alizadeh, Tim Kraska, and Samuel Madden. 2023. FactorJoin: a new cardinality estimation framework for join queries. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.

[76] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. 2020. Bayescard: Revitilizing bayesian frameworks for cardinality estimation. *arXiv preprint arXiv:2012.14743* (2020).

[77] Zongheng Yang, Wei-Lin Chiang, Sifei Luan, Gautam Mittal, Michael Luo, and Ion Stoica. 2022. Balsa: Learning a Query Optimizer Without Expert Demonstrations. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 931–944. https://doi.org/10.1145/3514221.3517885

[78] Geoffrey X Yu, Ziniu Wu, Ferdi Kossmann, Tianyu Li, Markos Markakis, Amadou Ngom, Samuel Madden, and Tim Kraska. 2024. Blueprinting the Cloud: Unifying and Automatically Optimizing Cloud Data Infrastructures with BRAD. *Proceedings of the VLDB Endowment* 17, 11 (2024), 3629–3643.

[79] Mingyi Zhang, Patrick Martin, Wendy Powley, Paul Bird, and Keith McDonald. 2012. Discovering indicators for congestion in DBMSs. In *2012 IEEE 28th International Conference on Data Engineering Workshops*. IEEE, 263–268.

[80] Mingyi Zhang, Patrick Martin, Wendy Powley, and Jianjun Chen. 2017. Workload management in database management systems: A taxonomy. *IEEE transactions on knowledge and data engineering* 30, 7 (2017), 1386–1402.

[81] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. Queryformer: A tree transformer model for query plan representation. *Proceedings of the VLDB Endowment* 15, 8 (2022), 1658–1670.

[82] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query performance prediction for concurrent queries using graph embedding. *Proceedings of the VLDB Endowment* 13, 9 (2020), 1416–1428.