LASER: Buffer-Aware Learned Query Scheduling in Master-Standby Databases (Extended Technical Report)

Yuwei Huang
Tsinghua University, China
hyw22@mails.tsinghua.edu.cn

Guoliang Li
Tsinghua University, China
liguoliang@tsinghua.edu.cn

ABSTRACT

Master-standby database deployment is a commonly adopted database architecture in modern production environments, thanks to its fault tolerance and high availability. However, despite the architecture's widespread application in various online services, relatively few research efforts have been made to improve its overall query performance. When a sequence of queries arrive, existing methods of scheduling them across master and standby servers still rely on rules or heuristics, which may overlook some potential optimization directions such as buffer utilization. If we can efficiently reuse the database buffers resident in memory through intelligent query scheduling, the average response time of user queries can be significantly reduced as opposed to reading data from disk.

To address this issue, in this paper, we introduce a new buffer-aware query scheduling system named *LASER*. The system integrates a lightweight learned model that can directly map a query to the data blocks it accesses. Then, based on the predictions of the queries, we develop adaptive query scheduling algorithms to perform query allocation as well as query rearrangements, aiming to maximize the overall buffer hit rate while also maintaining load balance. The proposed system requires no pre-training, and can adjust to unseen workloads on the fly through constant model updates and query re-allocation. In our experiments, we observe a reduction of ~80% in query completion time compared to other traditional heuristic-based methods, with relatively low extra overhead added to the critical path of query execution.

1 INTRODUCTION

For commercial databases, high availability and reliability are the two vital properties that are generally required to fulfill its service-level agreement (SLA), and a common solution is the master-standby database replication architecture that usually consists of a master node, which is responsible for all write queries and some read queries, and multiple standby nodes, which are responsible for read-only queries. Such allocation of workloads brings great improvement to the database's throughput as it reduces the burden on each individual node in case of peak load. The data stored in the master and standby nodes are essentially the same, thus the standby nodes can also serve as backups for the master node to provide availability and reliability. For data updates on the master node, the standby nodes will be synchronized based on the redo logs. Master-standby replication architecture is well supported by databases like MySQL [19], PostgreSQL [23], and openGauss [15].

For query scheduling in master-standby replication databases, a critical optimization direction is buffer utilization. Consider an example where there is only one big table in a two-node database and the buffer pool on each node can only accommodate half of the table data blocks. A workload consists of 2 queries that scan the first half of data blocks followed by 2 queries that scan the second half. Then, with naive Round-Robin scheduling, each node would receive both types of queries, resulting in complete cache misses. However, if the scheduler takes buffer utilization into account and allocates the same type of queries to the same node, the overall execution time can be vastly reduced due to buffer hits.

To promote buffer utilization, existing literature mainly adopts heuristic-based methods [22, 25] or addresses specialized queries [6, 20, 31]. These studies, while effective under specific settings, may have problems in more general cases. For example, [22] requires Materialized Query Table (MQT) which is an auxiliary data structure that occupies extra disk space and presents difficulty for adaptation to write queries. It also assumes batched arrival of queries in order to perform joint allocation of MQTs and queries. [25] calculates query similarities solely based on SQL texts, which is inaccurate as query access patterns are not directly reflected by the predicates. It also overlooks load balancing which is another critical aspect in query scheduling. To summarize, these works have the following limitations. First, they require prior knowledge of the queries in the workload such as query patterns, which largely depends on the insights from domain experts. Second, they do not build explicit maps between queries and their access patterns (i.e., which data blocks will be accessed by the query), either calculating similarities solely with query texts or relying on the database to provide such information, which is not generally available. Third, they lack the ability to learn from and adapt to complex workloads that are constantly changing or contains write queries. Finally, they pay relatively little attention to load balancing, which is also an important metric in master-standby query scheduling.

Challenges. There are four main challenges for efficient buffer-aware query scheduling in master-standby databases. First, it is necessary to predict the access pattern of each incoming query before its execution so as to decide which part of the database is to be loaded into memory **(C1)**. Second, smart scheduling algorithms need to be developed to promote buffer utilization based on the predicted query access patterns, while also maintaining load balance among all master and standby nodes **(C2)**. Third, the scheduling system should be able to handle various queries and adapt to both workload and database shifts during runtime **(C3)**. Finally, the whole solution cannot add too much overhead to the critical path of query execution **(C4)**.

Our Approach. To tackle the above challenges, we propose *LASER*, a buffer-aware **L**e**A**rned query **S**ch**E**duling system for maste**R**-standby databases. Specifically, to address **C1**, the system integrates a light-weight query-to-block model (Section 4), which can directly map a query to the data blocks it accesses. To address **C2**, we propose a set of self-adaptive scheduling algorithms (Section 5) to promote both buffer utilization and load balance, which include

adaptive query allocation and adaptive query selection. To address **C3**, we design self-adjustable parameters in the scheduling algorithms as well as mechanisms to strengthen the self-adaptation ability of the system (Section 6), including a model trainer to update and synchronize the query model on the fly and a DB monitor to detect and handle database shifts caused by write queries. To address **C4**, we decouple the lightweight query model at the client side with the training process at the server side, and make most of the scheduling decisions in parallel to query execution. We show by extensive experiments (Section 7) that our system can significantly accelerate query response with relatively little overhead added to the critical path of query execution.

Contributions. We make the following contributions:

(1) We identify the need of efficient query scheduling in master-standby database scenarios and formulate this problem under both static and dynamic settings. We propose *Laser*, a buffer-aware learned query scheduling system that is able to tackle this problem.

(2) We propose a highly modular lightweight query model that can directly map a query to its access pattern.

(3) We design adaptive scheduling algorithms that can simultaneously promote database buffer utilization as well as load balance, with the ability to self-adjust on the fly.

(4) We present mechanisms that enable *Laser* to handle complex dynamic query workloads, including query re-allocation, online model training, and database monitoring.

(5) We conduct extensive experiments on 5 different datasets against various heuristic-based methods and achieve ~80% reduction in query completion time, with relatively low overhead added to the critical path of query execution.

2 PROBLEM STATEMENT

In this section, we define the problem of *Query Scheduling* in a master-standby replication database setting, with input of either a static query set or a dynamic query stream. We also discuss the design principle for this problem.

2.1 Static Query Scheduling

Consider a master-standby replication database with $n \geq 2$ nodes (1 master node and $n-1$ standby nodes), all running as backend servers for external application queries. Along with that, a proxy layer (or JDBC/ODBC) is also deployed to gather all the input user queries and make scheduling decisions for them. Specifically, the proxy layer determines which backend servers the queries should be sent to and in what order they should be processed. The allocated queries are then executed locally on the database nodes. Once a query is completed, the proxy layer collects the result and sends it back to the corresponding user.

Under the static setting, all user queries arrive at time $t_0 = 0$ and thus form a static query set Q (e.g., batch processing in OLAP). The problem of *Static Query Scheduling* is then defined as follows.

Definition 2.1 (Static Query Scheduling). Given n master-standby replication database nodes and a static query set Q with $|Q| = m$, we aim to find an allocation function $f : Q \to \{1, 2, \ldots, n\}$ and an ordering function $g : Q \to \{1, 2, \ldots, m\}$ such that the following objective is minimized:

$$\min_{\substack{f:Q\to\{1,2,\ldots,n\}\\ g:Q\to\{1,2,\ldots,m\}}} \sum_{q\in Q} \mathbf{ct}\big(q, f(q); g(q)\big), \tag{1}$$

where $\mathbf{ct}(q, k; i)$ denotes the completion time (i.e., the time at which the proxy layer collects the full result set) of query q when it is executed as the i-th query on node k. Sometimes we care more about the maximum completion time, and the corresponding optimization objective would change to:

$$\min_{\substack{f:Q\to\{1,2,\ldots,n\}\\ g:Q\to\{1,2,\ldots,m\}}} \max_{q\in Q} \mathbf{ct}\big(q, f(q); g(q)\big). \tag{2}$$

Both (1) and (2) are called the problem of *Static Query Scheduling*.

2.2 Dynamic Query Scheduling

Under the dynamic setting, each query q now comes with a timestamp t_q, and we can take the difference between completion time $\mathbf{ct}$ and the timestamp t_q as its actual query response time. Also, the desired allocation function f and ordering function g are now dependent on query arrival time, i.e., they take the extra parameter $t_q \in \mathbb{R}_{\geq 0}$ as input. Taking these factors into consideration, we state the *Dynamic Query Scheduling* problem as follows.

Definition 2.2 (Dynamic Query Scheduling). Consider the same problem setting as in Definition 2.1, except that each query q is now associated with a timestamp t_q, we aim to find an allocation function $f : Q \times \mathbb{R}_{\geq 0} \to \{1, 2, \ldots, n\}$ and an ordering function $g : Q \times \mathbb{R}_{\geq 0} \to \{1, 2, \ldots, m\}$ such that the following objective is minimized:

$$\min_{\substack{f:Q\times\mathbb{R}_{\geq 0}\to\{1,2,\ldots,n\}\\ g:Q\times\mathbb{R}_{\geq 0}\to\{1,2,\ldots,m\}}} \sum_{q\in Q} \Big[\mathbf{ct}\big(q, f(q, t_q); g(q, t_q)\big) - t_q\Big], \tag{3}$$

where $\mathbf{ct}(q, k; i)$ is the query completion time as described in Definition 2.1. Optimization of (3) is called the problem of *Dynamic Query Scheduling*.

Note that for dynamic query streams, maximum completion time does not make much sense as it is heavily influenced by the query timestamps. Therefore, we treat the sum of query completion time as the only optimization goal in this setting.

2.3 Insights

Before introducing our *Laser* framework, we make some analysis of the predefined problem. Clearly, directly modeling the completion time of a query is rather complicated, as it requires large amounts of information including (i) Network latency, (ii) Query complexity, (iii) Available OS resources, (iv) Database load condition, and (v) Impact of preceding queries. Moreover, under the dynamic setting, only part of the queries can be seen during a scheduling process, rendering it impossible to find the global optimum of Equation (3) even with all the information above. On the other hand, as the proxy layer only connects to the query interface of backend databases, it cannot access low-level OS information such as memory usage or CPU load. The aforementioned impracticality of a direct solution pushes us to find efficient workarounds.

We make the observation that, among the factors which influence query completion time, some are mostly hardware-dependent (network latency, OS resources, etc.), and some are workload-intrinsic

(query complexity). Therefore, we mainly focus on the impact of preceding queries on the current query's execution. Inspired by the intuition that queries accessing similar data blocks should be sent to the same node and executed consecutively, our *Laser* system features a query model that can map a query to the blocks it scans, and scheduling algorithms that group similar queries onto each node while keeping load balance. This design principle helps to improve the overall buffer hit rate and eventually contributes to earlier query completion. The implementation details of our proposed system will be discussed in the following sections.

3 SYSTEM OVERVIEW

In this section, we present an overview of our *Laser* system, illustrating its various components and how they interact with each other in a typical query workflow.

3.1 Laser Architecture

Figure 1 shows the architecture of our *Laser* System. The system serves as a proxy layer (or in JDBC/ODBC) between applications and backend database servers. To external users, the system functions exactly the same as a normal database connection interface, with all the implementation details and deployment configurations hidden behind the scene. The interface provides basic functionalities for handling user requests such as querying and result-retrieving. In the following paragraphs, we will provide a brief introduction to the various components inside the *Laser* system.

Query Model. The user queries from external applications are first processed by the query model. The model, with the assistance of auxiliary information (e.g., database schema), takes the queries as input and outputs vectorized query features indicating their access patterns. Essentially, the access pattern of a query refers to the data blocks it accesses. Such predictions serve as guidance information for the subsequent query allocation process, as they can be effectively used to measure the similarity between different queries. The model adopts a lightweight MLP structure, which incurs relatively low overhead, and can be run efficiently even without dedicated hardware like a Graphics Processing Unit (GPU). Under our experimental settings, with batched processing, the query model takes an average of 3ms to encode each query when running on CPU, which is relatively affordable.

Allocator. The allocator is the central component in our system which is responsible for allocating user queries among backend database servers. For example, a naive round-robin allocator may distribute the queries in circular order according to their arrival time. In our *Laser* system, the query allocation process is performed using an adaptive greedy allocation algorithm, which takes both query access patterns predicted by the query model and current workload conditions on each node into consideration. The weights associated with the two factors are dynamically adjusted during system operation based on query run-time statistics. More details about the allocator module will be discussed later in this paper.

Database Connectors. The DB Connectors are a group of worker threads which directly communicate with backend database servers. Each DB Connector maintains multiple connections to one database node, and contains a local query queue that stores the queries sent by the allocator. The queries in the local query queue are then fetched and executed by the corresponding DB Connector. Note that instead of processing the queries in the order they are received, we develop an adaptive greedy selection algorithm to choose the most appropriate queries each time for execution. This algorithm is able to make a trade-off between query cost and buffer friendliness and, similar to the adaptive algorithm in the allocator, can adjust its weights on the fly according to query feedbacks. Once a query is completed, the DB Connector retrieves the result and returns it back to the applications. At regular intervals, the pending queries in the local query queues are collected back for query re-allocation, which we will explain later in Section 6.

Model Trainer. The model trainer is a component running on the master node. It maintains a structural copy of the query model and trains it in the background. To generate training data, every certain period of time, the trainer takes a snapshot of the buffered blocks in each database node. When a query completes, the trainer finds the snapshots that match the query's life cycle, calculates their differences, and then transforms the results into processable training data. These converted training data reside in a training data queue and are used for background model training. Periodically, the weights of the trained model are synchronized to the front-end model to capture the characteristics of the newest workload as well as current database status.

DB Monitor. The DB monitor also runs on the master node. This component keeps track of useful database metadata while the system is serving application queries, including schema information and the number of blocks in each table. These metadata can be used as auxiliary input for the query model. Note that during system running, some of the metadata may be modified due to write queries. In case of this, the monitor constantly detects possible shifts and informs the model trainer to discard the expired data when a change occurs. The monitor can also collect execution statistics such as the buffer hit rate in each database.

As we can see, all the components in the *Laser* system closely cooperate with each other to perform efficient query scheduling. Next, we will demonstrate the entire life cycle of the queries by examining a typical query workflow.

3.2 Query Workflow

A typical query workflow in our *Laser* system is illustrated in Figure 2. First, the user queries from external applications are sent to the query model for access pattern prediction. After that, the queries along with their access patterns are collected by the allocator. The allocator then makes allocation decisions for each query based on the information provided the query model and the load condition inside each local query queue. This process aims to group similar queries together while preserving load balance. The allocated queries will be placed in the local query queues of the corresponding DB Connectors. When queries are available in the local query queue and there exist idle connections, the DB Connectors select the most suitable queries at the time and execute them through backend servers. Finally, after a query completes, the result is returned back to the applications.

In the real scenario, some queries may fail to execute due to various reasons (e.g., deadlock). When an execution failure occurs, the DB Connector automatically retries up to a preset limit. If the retry

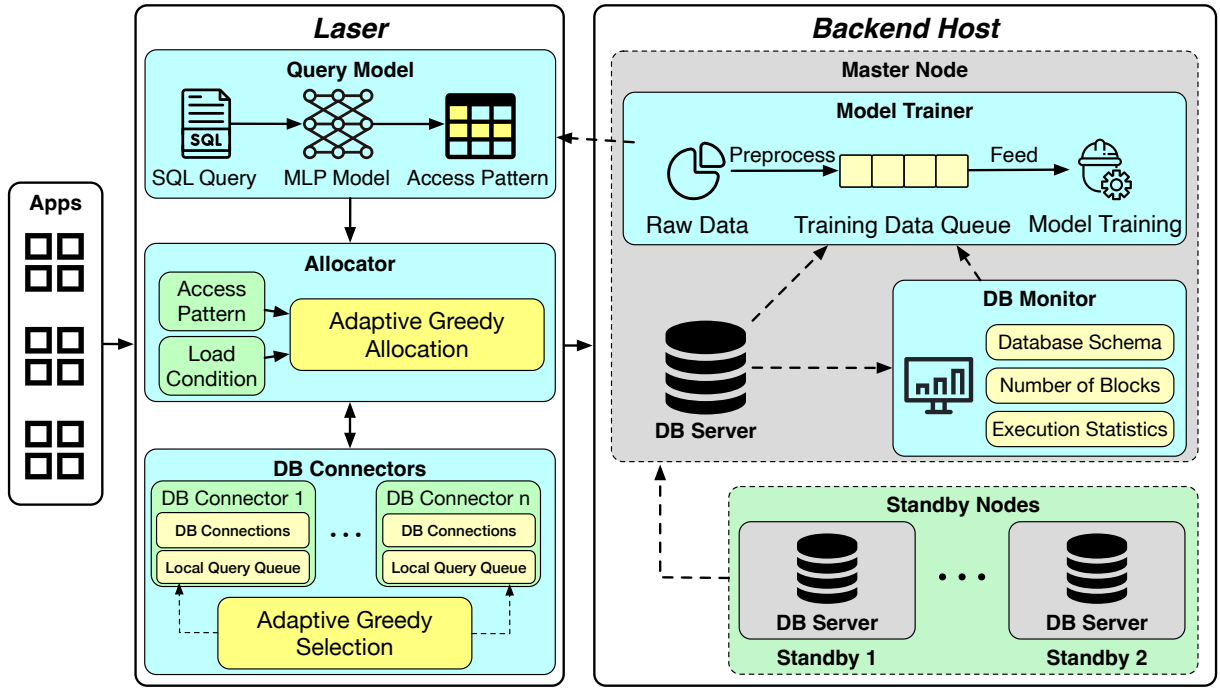

Figure 1: Architecture of the *Laser* system.

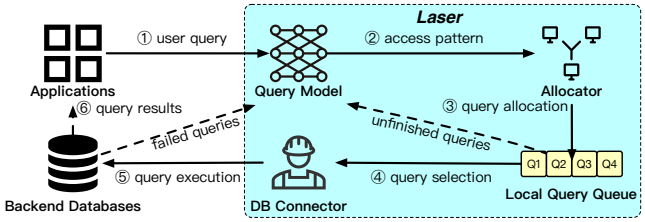

Figure 2: Query workflow in the *Laser* system.

limit is also reached, the failed query will be returned to the query model and put in a temporary queue. During system operation, the query model periodically triggers a re-allocation process for the queries in the temporary queue as well as the unfinished queries in each local query queue. Essentially, this process re-computes the access patterns of the queries and then re-allocates them with the allocator, as if the queries were newly arrived. By doing re-allocation, we can leverage the latest query model and weights in the adaptive algorithms to improve from previous allocation decisions. Note that query scheduling and query execution are performed following the producer-consumer paradigm, so they will not block each other and harm the overall query performance.

4 QUERY MODEL

The query model is one of the most important components in the *Laser* system. In this section, we provide a detailed description about how this component is designed to provide informative hints for the query scheduling process while also being lightweight. We first introduce how we encode the queries (§ 4.1) and model their access patterns (§ 4.2), and then highlight the concise modular structure of the query model which enables fast inference as well as quick response to database schema changes during runtime (§ 4.3).

4.1 Query Encoding

Various query encoding methods have been proposed in the literature, ranging from SQL-based one-hot encoding [11] to plan-based tree encoding [26, 33, 37]. Different query encodings possess different levels of expressiveness and complexity, making them suitable in different scenarios. Under the problem setting of this paper, we aim to embed the *access features* of a query into its encoding. Clearly, SQL text alone is not sufficient for this purpose, as it may contain both pre-scan and post-scan filters which are hard to distinguish. *In fact, we find that only the pre-scan filters matter when it comes to determining which data blocks are to be accessed by the query.* Therefore, to obtain more accurate information about data access, we turn to the query plans generated by backend servers. Unlike most existing works which encode the entire plan tree of a query, we focus only on the operators that contain necessary query access features, i.e., the scan operators. By eliminating irrelevant non-scan operators, we reduce the computational overhead of query encoding, which aligns with our lightweight design principle.

Specifically, given a query to be encoded, we first obtain its execution plan tree through database functionalities (e.g., `EXPLAIN` command in PostgreSQL), and then searches for all the scan operators in the leaf nodes. For each scan operator, we further focus on the following four types of features:

- **Scan Type.** The type of the scan operator, such as Sequential Scan, Index Scan, and Index Only Scan.
- **Table Name.** The name of the base relation to be scanned.
- **Index Name.** The name of the index to be used. This can be none if no index is involved in the scan operator.
- **Index Condition.** The conditions that determine which indexes are to be visited by an index-related scan operator.

Our current design primarily focuses on row-format storage and the accessed columns are not included in the features. The method can be easily extended to support column-format storage by adding the column features into query encoding and modifying the corresponding dimension settings inside the query model. Note that post-scan filters are also present in the scan operator, which we ignore as they do not affect the scanning process. For the *Scan Type*, *Table Name*, and *Index Name* features, we encode them directly using one-hot encodings, with the database schema information known in advance. The *Index Condition* feature, however, consists of two major categories that should be handled differently. One category is join conditions, such as `"a.id = b.id"`, which may contain multiple columns from different tables. For these conditions, we first represent them as global join ids, and then convert the ids into one-hot encodings. The other category is range predicates, such as `"a.id < 10"`, which only contain a single index column. For these conditions, we encode the column name and the operator with one-hot encodings and attach the normalized value right behind. Note that more complicated conditions other than the two categories rarely appear in index scans, so we do not consider them for simplicity of model design. Finally, the encodings corresponding to those two categories are concatenated together to ensure a fixed encoding length. The whole query encoding process described above is demonstrated in Figure 3.

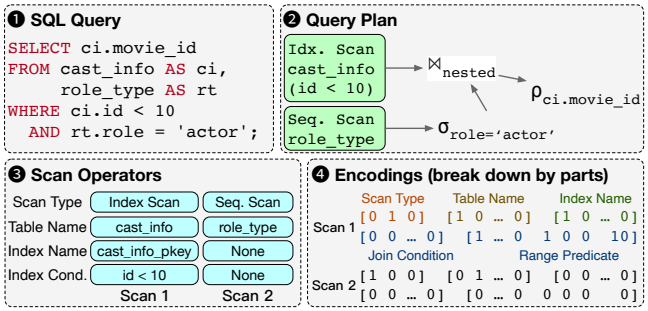

Figure 3: Query encoding process in *Laser*.

4.2 Access Pattern Modeling

As stated before, we want to build a mapping between the queries and their accessed blocks. To achieve this goal, besides encoding the queries, we also need to model the access pattern of each query into a mathematical form that is suitable for prediction and comparison. A naive approach would be to construct a block-access bitmap and use binary values to indicate which data blocks of a table are to be scanned. However, as the number of blocks in each table may be extremely large, it is beyond the capability of a lightweight model to predict for each data block individually. Suppose, for example, an 8GB table is stored as 8KB blocks on the disk, then for each query on this table, we need to make approximately 10^6 binary predictions using the bitmap representation, which is impractical.

To tackle the problem above, we have to reduce the dimensionality of the block-access bitmap. Inspired by prior work [26, 40], we divide the data blocks of a table into a fixed number of buckets. Specifically, for a table with B data blocks, we first choose an appropriate bucket number N, and then assign each bucket with $\lceil B/N \rceil$ blocks consecutively. Given a block-access bitmap, we calculate the percentage of marked bits (i.e., accessed blocks) inside each bucket and combine the resultant values in $[0, 1]$ to form a new vector. This vector can be viewed as a downsized version of the original bitmap. Below is an example of the process.

Example 4.1. Suppose a table with 12 data blocks is divided into 4 buckets, each corresponding to 3 blocks. A query on this table has block-access bitmap $[0, 0, 1, 1, 1, 1, 0, 1, 1, 1, 0, 0]$. Then, as shown in Figure 4, after the downsizing process, the query's access pattern is modeled as a vector $[0.33, 1.00, 0.67, 0.33]$.

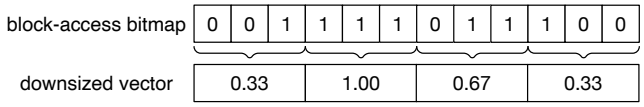

Figure 4: An example of the downsizing process.

Despite its lossy nature, the downsizing process is still favorable due to the following three reasons. First, the similarity information between queries are largely preserved. For different queries on the same table, the overlapping bits in their block-access bitmaps will contribute to the same buckets in the downsized vector. Conversely, if two queries take values x and y respectively in the same bucket, then it is guaranteed that even in the worst-case scenario, at least $\max(x + y - 1, 0)$ of data blocks in the bucket are accessed by both queries. Second, the representation length is independent of the actual size of the tables, thus we do not need to design models with different output dimensionality specifically for each table. Last, the length of the downsized vector is flexible, which provides a trade-off between representation granularity and computation complexity. As the vector length N increases, we can capture more fine-grained similarity information at the cost of higher model requirements and longer processing time. The impact of N on query scheduling performances will be studied later in the experimental Section.

It is worth mentioning that query access patterns have direct links with database buffer states. Specifically, the data blocks represented in the access pattern of a query are very likely to exist in the database buffer pool after its execution, unless the buffer pool is not large enough to accommodate all the accessed blocks. This is due to the fact that common buffer replacement algorithms like LRU tend to keep recently accessed blocks inside the buffer pool. To verify this correlation, we generate 100 index-only range scan queries whose actual access patterns are easily obtainable and collect the newly buffered blocks (i.e., database buffer state changes) after executing each of them. The collected blocks are represented and

downsized in the same way as query access patterns. We then plot the correlation coefficients of the 100 pairs of query access patterns and buffer state changes in Figure 5. As we can see, the query access patterns have a strong positive correlation with database buffer state changes, with over 75% of the correlation coefficients larger than 0.8. This correlation is critical since we will later simulate the changes inside database buffer pools with query access patterns (Section 5.3) and collect database buffer states as training data for the query model (Section 6.2).

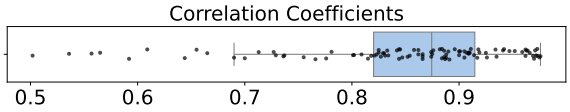

Figure 5: Box-plot of correlation between query access patterns and database buffer state changes.

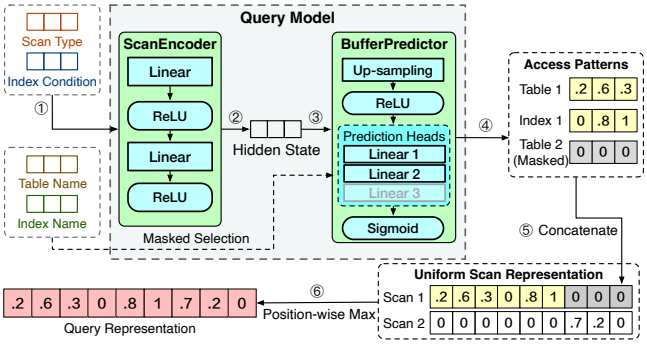

Figure 6: Architecture of the Query Model.

4.3 MLP Model Structure

Since our *Laser* system is designed as a client-side interface, it is important that the query model does not incur too much computational overhead so as not to harm the end-to-end query performance, especially when no dedicated hardware for running the model (e.g., a GPU) is available. Also, as the database schema may not be consistent during long-term operation, the structure of the query model need to be flexible in order to avoid retraining from scratch each time a table is added or removed. To address these issues, we employ a lightweight and modular MLP model design which is not only efficient in training and inference, but also able to quickly adapt to schema changes without completely altering its structure. Details of the query model are depicted in Figure 6.

Our query model functions at the granularity level of scan operators. Given an encoded query, we first use a two-layer *ScanEncoder* to convert each of its scan operators into a hidden state. The *ScanEncoder* takes the *Scan Type* and the *Index Condition* parts of the scan encoding as input, and adopts ReLU as its activation function. Output of this model is a fixed-size vector that embeds the semantic information of the input scan operator.

After obtaining the hidden states, we further utilize a two-layer *BufferPredictor* to calculate the final access patterns. Specifically, for each scan operator, we perform an up-sampling on its hidden state, and then apply a prediction head to get the desired access pattern. The up-sampling layer is equipped with ReLU activation, while the prediction head adopts Sigmoid to normalize model outputs into the $[0, 1]$ interval. We set different prediction heads for different tables or indexes, and use the *Table Name* and the *Index Name* parts of the scan encoding as masks to choose from them. Such modular design prevents different relations from intervening with each other during model training and makes parallel processing possible. Also, it is convenient to add or delete prediction heads in case of database schema changes without needing to restructure the model and discard all current model parameters (more detailed discussion in Section 6.3). Output of the prediction head is a vector of predefined length (i.e., number of buckets as described in Section 4.2).

Clearly, for tables and indexes not involved in the scan operator, the corresponding access patterns are zero vectors. To make model predictions of different inputs comparable, we concatenate the access patterns of all relations together to form a unified representation for each scan operator, which is of length `N_RELATIONS` $\times$ `N_BUCKETS`. When a query contains multiple scan operators, we merge the model predictions of all the operators using position-wise max function to derive the representation of the query itself.

Model Bypass. In fact, for some scan operators, we can optimize their model prediction process with prior knowledge about how the database actually performs them. For example, an index-only scan will not access any data block of the base table, while a sequential scan will access every data block of the base table and ignore the index blocks. Therefore, if we know in advance that Scan 1 in Figure 6 is an index-only scan, we may manually set the first 3 digits of its representation corresponding to table 1 as $(0, 0, 0)$. Similarly, if Scan 2 is a sequential scan, we may set the last 3 digits of its representation corresponding to table 2 as $(1, 1, 1)$. Such process brings three benefits. First, we can save computational overhead for model inference, which is crucial as it lies in the critical path of query response. Second, the injected prior knowledge is more accurate than that learned by the query model, which can help the allocator to better measure the query similarities. Third, the process provides a good start point for model predictions when the system initializes. In this way, *Laser* can achieve higher quality and lower latency even without pre-training, as later verified in Section 7.8.

5 SCHEDULING ALGORITHM

The scheduling algorithms are another critical part in our *Laser* system, which includes adaptive greedy allocation in the allocator and adaptive greedy selection in the DB Connectors. In this section, we first explain our optimization goals (§ 5.1), and then introduce the implementation details of each of these algorithms (§ 5.2, § 5.3).

5.1 Optimization Goals

The ultimate goal of the *Laser* system is to reduce the end-to-end completion time of a given query set or a time-dependent query stream, as formally defined in Section 2. Here we focus on the dynamic setting as it is a more general case. We first show that the problem in Eq. (3) is NP-hard. In fact, consider a special case of our

problem by ignoring the ordering function g, then optimization for f would be a clustering problem that groups queries onto each node. Consider another special case by ignoring the allocation function f, then optimization for g would be a Travelling Salesman Problem (TSP) that needs to take into account the impact of consecutive queries. Another major challenge in solving Eq. (3) is that the completion time **ct** is an abstract function which is difficult to compute. To this end, we propose 3 optimization strategies that are easier to handle and can help reduce **ct**: (i) group similar queries together for better buffer utilization and faster execution, (ii) promote load balance to alleviate the long-tail problem of query completion, (iii) prioritize short queries as a query's execution time is counted in the completion time of all its successors.

Inspired by the analysis above, we break the problem into 2 stages with smaller scales and more manageable objectives. Stage 1 of our scheduling framework is a constrained clustering problem. Recall that after going through the query model, each query now is associated with its representation, which is referred to as an *access pattern*. With these access patterns, we can quantify the distance between different queries. Specifically, given two queries q_x, q_y with access patterns x, y of length L, we define their distance as

$$d(q_x, q_y) := \frac{1}{L}\sum_{i=1}^{L}|x_i - y_i|. \tag{4}$$

Here we use the L_1 distance to directly reflect the difference between two access patterns and then normalize it into $[0, 1]$. Besides query distance, another critical factor to take into account is query complexity. To this end, we directly utilize the estimated execution cost included in the query plans as the complexity measurement, denoted as $\text{cost}(q)$. With these two functions defined, we can finally formulate the Stage 1 Problem. Specifically, given n database nodes and m current visible queries, the Stage 1 Problem is defined as

$$\begin{aligned} &\min_{\bigcup_{i=1}^{n} Q_i=\{1,2,\ldots,m\}} \max_{i\in\{1,2,\ldots,n\}} \max_{q,q'\in Q_i} d(q,q'), \\ &\text{s.t.} \qquad \sum_{q\in Q_i}\text{cost}(q) < \theta, \quad i \in \{1,2,\ldots,n\}, \end{aligned} \tag{5}$$

where Q_i denotes the set of queries allocated to node i and θ is a cost threshold. We express the clustering criteria in Eq. (5) as minimization of the maximal cluster diameter. Essentially, the Stage 1 Problem is a combination of strategies (i) and (ii) which group similar queries together under load balance constraints.

Stage 2 of our scheduling framework is performed after queries have been allocated to each node. The purpose of this stage is to define an optimal query execution order within each node, and the corresponding problem is a TSP. Specifically, consider a node i with allocated query set Q_i. Suppose $r = |Q_i|$ and P is the set of all permutations of Q_i, then the Stage 2 Problem is defined as

$$\begin{aligned} &\min_{(q_1,q_2,\ldots,q_r)\in P}\sum_{i=1}^{r-1} d(q_i, q_{i+1}), \\ &\text{s.t.} \qquad \left|\left\{(i,j) : i < j, \text{cost}(q_i) > \text{cost}(q_j)\right\}\right| < \tau, \end{aligned} \tag{6}$$

where $d(\cdot,\cdot)$ and $\text{cost}(\cdot)$ are defined as in Eq. (5) and τ is a threshold indicating the priority of short queries. The constraint in Eq. (6) aims to control the number of inverse execution time pairs, i.e., short queries executed after long queries. Essentially, the Stage 2 Problem is a combination of strategies (i) and (iii) which prefer similar queries to execute consecutively for better buffer utilization while also taking into account the short query priorities.

5.2 Adaptive Greedy Allocation

To address the Stage 1 Problem, a straightforward idea is to adopt common clustering algorithms such as K-Means. However, in dynamic scenarios, these algorithms are not suitable since queries often arrive one by one and, more importantly, they are very time-consuming. To satisfy the demand of allocating queries individually and quickly, we develop the adaptive greedy allocation algorithm which allocates queries according to adjustable scores. Specifically, we first define the load factor of a node i in a particular state as

$$l_i := \frac{\sum_{q\in Q_i}\text{cost}(q)}{\sum_{j=1}^{n}\sum_{q\in Q_j}\text{cost}(q)}, \tag{7}$$

where n is the total number of database nodes and Q_j is the set of queries having been allocated to node j. Taking values within $[0, 1]$, the load factor l_i reflects the relative load condition of node i compared to other nodes. We also keep track of the current cluster center c_i of each node i, which is initialized as $\mathbf{0}$, during system running. Essentially, c_i is the average of the access patterns of the queries in Q_i. When a new query arrives, the algorithm first determines whether it is a write query, and if so, the query is directly sent to the DB Connector of the master node. For a read query q that can be executed on all nodes, the allocator calculates a weighted score s_i for each node i as

$$s_i := d(q, c_i) + w_\text{a} * l_i, \tag{8}$$

and selects the node with the lowest score to be the query's destination. Intuitively, a low score s_i suggests that the incoming query is similar to those queries already allocated to node i in terms of access pattern, and that node i is currently underloading. This criteria is similar to the one used in the K-Means algorithm, but with the constraint in Eq. (5) added as a penalty term. After the query is sent to the DB Connector of the selected node, we update its center and the load factors of all nodes to prepare for future allocation.

The weight w_a in Equation (8) is an adjustable parameter. If we want to focus more on load balancing, then we can simply increase the value of w_a, and vice versa. When the system starts, w_a is initialized to 0 to emphasize buffer hit rate. Every time a node runs out of query, we check the local query queues in other DB Connectors, and if there are still pending queries, we decide that the current load is not balanced and increase w_a by a fixed step. Conversely, if the above situation does not occur for a certain period of time, we decrease w_a by the same step to redirect some focus back to buffer friendliness. This tuning process is fully automatic and does not require human intervention. The pseudo-code of the adaptive greedy allocation algorithm is listed in Algorithm 1.

5.3 Adaptive Greedy Selection

After allocation, the queries are placed in the local query queues of the corresponding DB Connectors. The DB Connectors then determine the order in which the queries are to be executed, which is referred to as the Stage 2 Problem. As described in Section 5.1, the Stage 2 Problem is a constrained TSP, thus there is currently no polynomial-time algorithm for an optimal solution. On the other

Algorithm 1: Adaptive Greedy Allocation

Global Variables: number of database nodes n,
adaptive weight w_a (initialized as 0),
centers of each node c (initialized as $\mathbf{0}$).

function UpdateAllocWeight($step$)
  if load imbalance is detected **then**
    $w_a \leftarrow w_a + step$;
  else if no imbalance is detected recently **then**
    $w_a \leftarrow \max(w_a - step, 0)$;
function QueryAllocation($queries$)
  for every q in $queries$ **do**
    if q is a write query **then**
      allocate q to the master node;
    else
      $l \leftarrow$ list of current load factors;
      for i from 1 to n **do**
        $score[i] \leftarrow d(q, c[i]) + w_a * l[i]$;
      allocate q to the node with minimal score;
    update the center of the target node;
    update the load factors of all nodes;

hand, the queries in each local query queue do not form a static set, thus even we can solve the TSP very efficiently, the optimal solution may change when new queries arrive. Therefore, instead of infeasibly searching for the global optimum of query execution order, we focus on the *best* query to be executed each time, which is both fast and stable in performance. Specifically, each DB Connector maintains an estimation of the buffer state in the corresponding database node, which is represented in the same form as access patterns. The estimated vector, denoted as e, is initialized as $\mathbf{0}$ and updated using the Exponential Moving Average (EMA) mechanism:

$$e \leftarrow (1 - \alpha) * e + \alpha * x. \tag{9}$$

Here α is a hyper-parameter within $[0, 1]$. The update in Equation 9 takes place whenever a query with access pattern x is sent for execution, thus e is dynamically adjusted to simulate the database buffer pool. With this estimation, we can calculate the distance of each access pattern to the current database buffer state as in Equation (4). In addition, we also consider the relative cost of each query q with respect to all the queries in the local query queue Q:

$$r(q) := \frac{\text{cost}(q)}{\sum_{q' \in Q} \text{cost}(q')}. \tag{10}$$

When selecting a query to be executed, the DB Connector computes the weighted score $S(q)$ of each query q as

$$S(q) := d(q, e) + w_s * r(q), \tag{11}$$

and regards the query with the lowest $S(q)$ as the *best*. Similar to w_a in Equation (8), the weight w_s is also an adaptive parameter that balances between buffer hit rate and short query priority. Hence by minimizing $S(q)$, we can simultaneously achieve the two optimization goals described earlier in this section.

To update w_s on the fly, we collect the execution time of all queries over fixed time windows and count the number of inverse pairs in each window, i.e., the constraint as formulated in Eq. (6). For a random sequence of length N, the expected number of inverse pairs is $N(N-1)/4$. Therefore, if the counted number is large compared to this expectation, we decide that w_s needs to be increased to assign more priority to short queries. Conversely, if the counted number is relatively small, we can decrease w_s to concentrate more on buffer friendliness. Essentially, we move the constraint in Eq. (6) to the objective function as a penalty term and use adaptive w_s to balance the two. The pseudo-code of the adaptive greedy selection algorithm is listed in Algorithm 2.

Algorithm 2: Adaptive Greedy Selection

Global Variables: local query queue Q,
adaptive weight w_s (initialized as 0),
buffer state estimation e (initialized as $\mathbf{0}$).

function UpdateEstimation(x, α)
  $e \leftarrow (1 - \alpha) * e + \alpha * x$;
function UpdateSelectWeight($step, exectime$)
  if number of inverse pairs in $exectime$ is large **then**
    $w_s \leftarrow w_s + step$;
  else if inverse pairs in $exectime$ is few **then**
    $w_s \leftarrow \max(w_s - step, 0)$;
function QuerySelection($queries$)
  while there are idle connections **do**
    $r \leftarrow$ list of relative costs of the queries in Q;
    `# here we write q as index for simplicity`
    for every q in Q **do**
      $score[q] \leftarrow d(q, e) + w_s * r[q]$;
    $qbest \leftarrow$ the query with minimal score;
    update e with the access vector of $qbest$;
    remove $qbest$ from Q and execute $qbest$;

6 DYNAMIC ADAPTATION

Besides the query model and scheduling algorithms, our *Laser* system also features capabilities to adapt to complex dynamic query workloads. In this section, we introduce how we revise query allocation decisions based on newly acquired information (§ 6.1), how we collect training data and update the query model in parallel to serving user queries (§ 6.2), and how we react to possible database shifts during runtime (§ 6.3).

6.1 Query Re-allocation

The algorithms described in Section 5 only consider the information currently visible, which is very limited under dynamic settings. For example, suppose there are two database nodes in the system, and a workload consists of two types of queries Q1 and Q2, where 10 queries of type Q1 arrive before 10 queries of type Q2. Then, upon seeing the Q1 queries, the algorithms will assign each of the nodes 5 queries to promote load balance. However, the global optimum of query allocation in this scenario is to assign queries of the same

type to the same node, which the algorithms fail to accomplish. To address this issue for similar dynamic workloads, we propose the query re-allocation mechanism, as described below.

In essence, query re-allocation can be regarded as a revision process, which aims to combine the information of newly arrived queries into the previous allocation decision. Specifically, every certain period of time, we collect all the pending queries in the local query queues, re-compute their access patterns, and then re-allocate them with the allocator. This process is done as if the queries were newly arrived at the time. Since the number of queries to re-allocate may be large, instead of re-allocating one by one, we perform a cost-bounded K-Means clustering which directly addresses the Stage 1 Problem, as shown in Algorithm 3. Note that we allow a certain over-allocation ratio on each node (i.e., the sum of allocated query costs is above average) as the query costs cannot be perfectly evenly distributed. By doing re-allocation, we are essentially utilizing all the visible information to revise the allocation decisions for previous queries. In the example above, Q1 and Q2 would certainly belong to different clusters, thus we can derive a closer-to-optimum solution through query re-allocation. The re-allocation process can also benefit from the latest query model and adaptive algorithm parameters. When load imbalance is detected (see Section 5.2), we use query re-allocation to assign some queries from busy DB Connectors to the idle DB Connector.

Algorithm 3: Cost-bounded K-Means

Global Variables : number of database nodes n,
list of centers of each node c.

function CostBoundedKmeans(*queries*)
  $centers, iter, eps \leftarrow c, 200, 1e\text{-}5$;
  $tot \leftarrow$ total cost of the queries;
  # allow 50% of over-allocation
  $ub \leftarrow 1.5 * tot/n$;
  for i from 1 to $iter$ **do**
    for every q in *queries* **do**
      if q is a write query **then**
        assign q to the master cluster;
      else
        $ac \leftarrow$ accumulated cost of each cluster;
        $cl \leftarrow$ clusters whose ac is below ub;
        assign q to the nearest cluster in cl;
    update *centers* with the new cluster centers;
    if change in *centers* is less than *eps* **then** break;

6.2 Online Model Trainer

As our query model is deployed without pre-training, it is necessary to collect training data and tune the model during runtime to improve accuracy and adapt to the newest workload pattern. This is done through the online model trainer, which is a background-running component at the backend host. Ideally, the trainer would track the accessed blocks of each query and directly transform them into a training pair. Nevertheless, such tracking could incur additional query execution overhead and not all database engines have built-in support for it. To overcome this issue, we propose a more efficient workaround. Instead of focusing on the access pattern of each query, we keep track of the buffer state changes inside the databases. Specifically, the model trainer periodically takes snapshots of the current buffer states inside each node (see Figure 7) and arrange them into a time-ordered list. It also records the start time and finish time of each query. When a query completes, the trainer finds the two closest snapshots in the corresponding node that covers the entire life cycle of the query and calculates their differences by examining (1) blocks that newly appear in the later snapshot and (2) blocks whose access count has increased in the later snapshot. After that, we filter out the blocks which belong to relations irrelevant to the query and divide the remaining blocks into buckets to form an access pattern. The query and the access pattern are then paired and put into the training data queue. Note that training data derived in this way may be inaccurate, which is a compromise since we do not want to intervene in the actual query execution process.

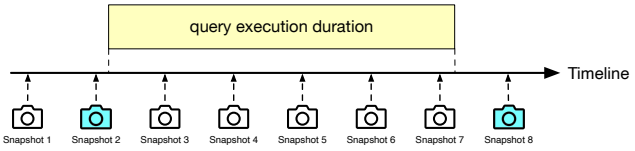

Figure 7: Finding matched snapshots (2 and 8) for a query.

The training data queue is a sliding window of fixed size. When a new training pair arrives, we evict the oldest item from the queue to ensure data freshness. A global expiration time is set to avoid overfitting the model on some of the data. Every certain period of time, the trainer iterates through all the training data in the queue and trains the query model for an epoch, using the common MSE loss and Adam [10] optimizer. The trained model parameters are synchronized to the frontend model in a regular manner to improve its performance on the fly.

6.3 DB Monitor

As described in Section 4, the query encoding process and query model structure highly rely on database metadata, such as schema information and the number of blocks in each relation. To better manage these metadata during runtime, we set up the DB monitor, which also runs in the background at the backend host. Essentially, the monitor can be viewed as a metadata server that is deployed to alleviate the burdens of database nodes. It maintains a copy of the needed information, answers metadata queries from the trainer, and constantly updates in case database shifts occur due to complex workloads that may contain write queries.

We mainly pay attention to two types of shifts that can be caused by user queries. The first type comes from Data Manipulation Language (DML), including `INSERT`, `UPDATE`, and `DELETE` queries. These queries may change the number of blocks in the target relation as well as the data distribution inside those blocks. In consideration of this, when a DML query is detected, the monitor updates the block number of the target relation and informs the trainer to discard any training data that contains this relation. The second type

comes from Data Definition Language (DDL), which may directly modify the database schema. In rare cases when a DDL statement is detected, the monitor updates the corresponding metadata used in query encoding phase and notifies the trainer to add/remove prediction heads for new/deleted relations. Any incompatible training pair after the schema modification is also erased from the training data queue. Note that we do not discard the model itself in both scenarios, as it still preserves useful information about other unchanged relations.

7 EXPERIMENTS

In this section, we evaluate our proposed system and compare with baseline methods. We first introduce our experimental settings (§ 7.1) and present the results for both static query scheduling (§ 7.2) and dynamic query scheduling (§ 7.3). We will also make an analysis of the overhead incurred by the *Laser* system (§ 7.4). Effects of different query arrival rates (§ 7.5), number of concurrent connections (§ 7.6), algorithm parameters (§ 7.7), percentage of pre-trained data (§ 7.8), and access pattern sizes (§ 7.9) will be discussed as well. In Section 7.10 and 7.11, we present ablation studies on scan features and applied techniques in *Laser*. At the end of this chapter (§ 7.12), we explore the performance of *Laser* on SSDs.

7.1 Experimental Settings

Environments. We deploy a master-standby replication database with 3 nodes (1 master node and 2 standby nodes) on 3 identical Ubuntu 22.04.3 LTS servers, each equipped with dual 2.20GHz Intel(R) Xeon(R) E5-2630 v4 CPUs and 128 GB of DDR4 RAM. All these servers use HDDs as their permanent storage devices, which have a rotational speed of 7200 RPM and a typical sequential read speed of 200 MB/s. We use PostgreSQL v15.3 as the database engine and configure physical replication slots to synchronize data between master and standby nodes. We utilize Linux Control Groups to constrain the maximal memory usage of the database processes according to different dataset sizes and set `shared_buffers` to be 25% of available memory across all our experiments.

Implementation. Our *Laser* system is written with Python 3.10. The frontend layer implements various functionalities related to query scheduling and wraps the Psycopg2 [24] adapter to communicate with backend database nodes. The query model is constructed and trained using PyTorch [21] API without CUDA acceleration. We use 1024 buckets for access pattern modeling as it generally produces good query performances in all experiments. The whole system runs on the same server as the master node.

Datasets. We use the following datasets in our experiments.

- **TPC-H** [2] is a widely adopted OLAP benchmark consisting of 8 relations and 22 query templates. We use a scale factor of 10 (about 11 GB of data) and generate 1000 queries out of these templates with distinct random seeds.
- **TPC-DS** [1] is also a popular OLAP benchmark with 24 relations and 99 query templates. We use a scale factor of 10 (about 12 GB of data) and generate 1000 queries with distinct random seeds.
- **JOB** [13] is an OLAP benchmark built upon the IMDB dataset, which contains 21 relations and about 3.7 GB of raw data. We sample from the 113 provided benchmark queries with replacement to form a workload of 1000 queries.
- **Sysbench** [29] is a customizable database benchmark tool. We use this tool to generate a dataset with 10 relations, each containing about 1 GB of data. We also design an OLTP workload with 1000 queries, of which 20% are update queries, 40% are single-table select queries, and 40% are multi-table select queries.
- **HTAP** is a hybrid workload generated by Swarm64 Toolkit [28], which contains 200 TPCH-like OLAP queries and 800 TPCC-like OLTP transactions. The underlying database contains 12 relations and we use a scale factor of 10 (about 20 GB of data).

For TPC-H, TPC-DS, and Sysbench datasets, we constrain each database process to use no more than 8 GB of memory and hence set `shared_buffers` to be 2 GB. For JOB dataset, we limit database memory usage as 1 GB and set `shared_buffers` to be 256 MB. For HTAP dataset, we only allow 4 GB memory budget and 1 GB `shared_buffers` to simulate more buffer-intense scenarios. We avoid complex queries in the workload that take extremely long time (more than 1 hour) to execute individually under our memory constraints for experiment practicability. All queries are shuffled randomly before being scheduled.

Baselines. We compare the *Laser* system with the following baselines in our experiments.

- **Round-Robin (RR)** is a naive strategy that allocates the incoming queries in cyclic order to each database node.
- **Fixed-Table (FIX)** is a heuristic-based method that assigns each node with a pre-defined set of relations and allocates incoming queries on these relations to the assigned node. When a query involves multiple relations, the strategy considers the first one.
- **Minimal-Waiting (MIN)** is a heuristic-based method that allocates each incoming query to the database node with minimal number of waiting queries at that time.
- **Quickest-Queueing (QCK)** is a heuristic-based method that tracks the accumulated queueing time on each node and allocates the incoming queries to the node with the quickest queueing.
- **CAS** [25] is a heuristic-based method that calculates query similarities based on signatures to promote buffer utilization.

For all these baselines, we implement both the basic version and the version with query stealing, that is, a node with idle connections would steal queries from another busy node. Query stealing can be very beneficial when the workload distribution among database nodes is highly imbalanced.

Metrics. We use the following metrics to measure the performance of each scheduling method in our experiments.

- **Average Execution Time** is the mean of all queries' actual execution time, i.e., not including the time spent on query scheduling and queuing. This metric directly reflects the query optimization ability of each method.
- **Average Completion Time** is the mean of all queries' completion time, i.e., including the time spent on query scheduling and queuing. This metric reflects the effectiveness of each method in accelerating query response.
- **Maximal Completion Time** is the maximum of all queries' completion time, i.e., the time when the last query is completed. This metric can reflect the load balancing capability of each method. We only use this metric in static query scheduling experiments as mentioned in Section 2.

- **Shared Buffers Hit Rate** is the hit rate in database shared buffers. This metric can serve as an indicator for buffer friendliness. We do not include statistics from OS buffer cache as they are hard to obtain from inside a database.

7.2 Static Query Scheduling

Under the static setting, all queries are visible at time 0 and thus ready for scheduling. We maintain 8 DB connections for each node and use the same table to warm up database buffer before conducting experiments on different scheduling strategies. Figure 8 shows the performance of each method in terms of the aforementioned metrics, and we make the following observations.

First, our *Laser* system outperforms all the baselines by a large margin in both execution time and completion time, which verifies the effectiveness of the proposed method in accelerating query execution and response. Such performance improvement can be attributed to the fact that *Laser* always achieves the highest in-database buffer hit rate across all experiments. As the complexity of database schema increases, the advantage of our method becomes more obvious thanks to its ability to capture complicated query access patterns. For example, on the TPC-DS dataset, *Laser* reduces 74.7% of average execution time, 82.4% of average completion time, and 76.1% of maximal completion time compared to the best-performing baseline FIX with query stealing.

Second, among all the baselines, FIX generally produces better results. This finding further emphasizes the significance of buffer friendliness in master-standby database query scheduling, as FIX is one of the only 2 baselines to take this factor into consideration. The other baseline CAS that also considers buffer utilization, however, performs not that good on complex datasets such as TPC-DS. This is probably due to its over-simplified encodings of join predicates. Although FIX does not distinguish itself from other baselines in terms of in-database buffer hit rate, its decent performance can be due to more hits in the underlying OS buffer cache.

Third, MIN and QCK do not exhibit significantly superior performance compared to the naive RR strategy. This is because both methods require runtime statistics to adjust their scheduling decisions, while, under the static setting, no such information is available when queries arrive.

Finally, query stealing generally improves the performance of each baseline, especially in reducing maximal completion time. This is due to the mechanism's ability to promote load balance among database nodes and eliminate long-tail distribution of query completion time. Such capability is crucial to strategies like FIX where query load distribution can be highly imbalanced, e.g., on JOB and Sysbench datasets. Our method, on the other hand, naturally supports query stealing through re-allocation, thus no additional mechanism is needed to ensure load balance.

In conclusion, our *Laser* system can greatly improve query performance and promote load balance through better buffer utilization and query allocation, which are of great importance when it comes to query scheduling in master-standby database scenarios.

7.3 Dynamic Query Scheduling

Under the dynamic setting, each query is associated with an arrival timestamp, thus only queries arriving before t are visible at time t. We utilize exponential distribution to generate these timestamps. Specifically, given a desired query arrival rate λ, we independently sample arrival intervals $s_1, s_2, \ldots, s_n$ from $\text{Exp}(\lambda)$ and set the arrival timestamp of query k as $t_k = \sum_{i=1}^{k} s_i$. Essentially, we model the events of query arrival as a Poisson process.

Similar to the static setting, we maintain 8 DB connections for each node and warm up database buffer states with the same relation for different methods. As query stealing generally improves performance of the baselines, we omit experiments using their basic versions. For TPC-H, TPC-DS, and JOB datasets, we set query arrival rate $\lambda = 0.5/s$, while for the Sysbench and HTAP dataset we choose $\lambda = 1/s$. The same random seed is used before generating arrival intervals for each experiment run. Figure 9 illustrates the experiment results of all scheduling strategies.

As we can see, under the dynamic setting where information is rather limited compared to the static one, our *Laser* system still surpasses all the baselines in terms of the 3 metrics. Also, FIX remains the best-performing baseline as before, which again proves the importance of buffer friendliness in query scheduling. It is worth noting that with query stealing, RR does not fall behind MIN and QCK, which are heuristics primarily designed for dynamic scenarios. Such a simple mechanism turns out to be effective in balancing loads among database nodes. The effect of query arrival rate λ on the overall performance will be discussed in Section 7.5.

7.4 Overhead Analysis

Besides buffer utilization, another major concern of our work is to add as little extra overhead as possible to the query execution process, especially in the critical path. The extra overhead here includes time spent on retrieving query plans, predicting access patterns, adaptive query allocation, adaptive query selection, and possible re-allocation. Table 1 lists the time consumed by each of these processes during experimentation on the JOB dataset with respect to different number of queries. Note that the query selection time in the table is the time to choose one best query from a query queue of given length. As we can see, except query selection, other processes take only 1 ms ~ 30 ms per query, which is quite negligible compared to the actual query execution time. The query selection time, on the other hand, largely depends on the length of the query queue as it requires a linear pass through all candidate queries. Nevertheless, even with all 1000 queries in the same queue, this process still takes less than 1 second to complete, which is also relatively affordable. In fact, all these extra overheads combined only account for less than 1% of average query execution time. More importantly, the above processes are usually performed while the database nodes are busy executing other queries, which means that the consequent overhead would not be counted into the critical path of query execution. In cases where these overheads do matter, e.g., when scheduling very short-running queries, we suggest using a smaller access pattern size to further accelerate the scheduling processes as later discussed in Section 7.9. But generally speaking, it can be concluded that our *Laser* system is a light-weight solution towards query scheduling in master-standby databases.

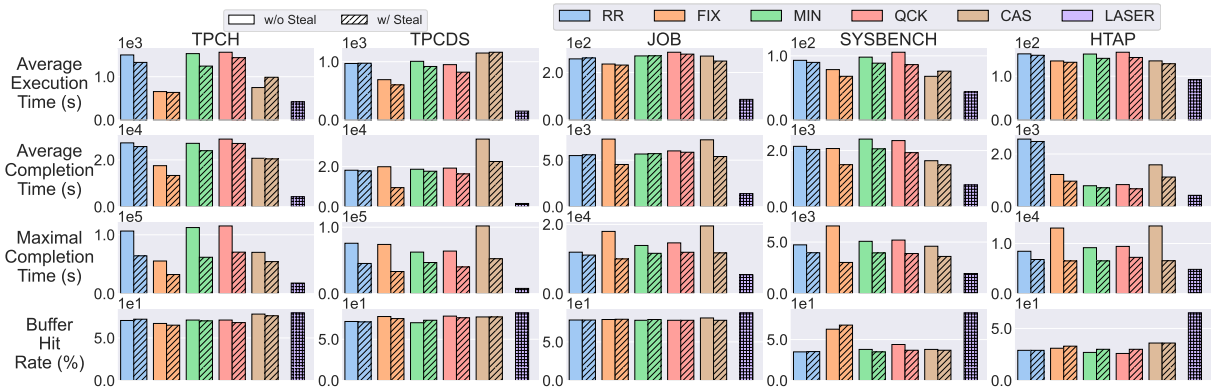

Figure 8: Experiment results under the static setting.

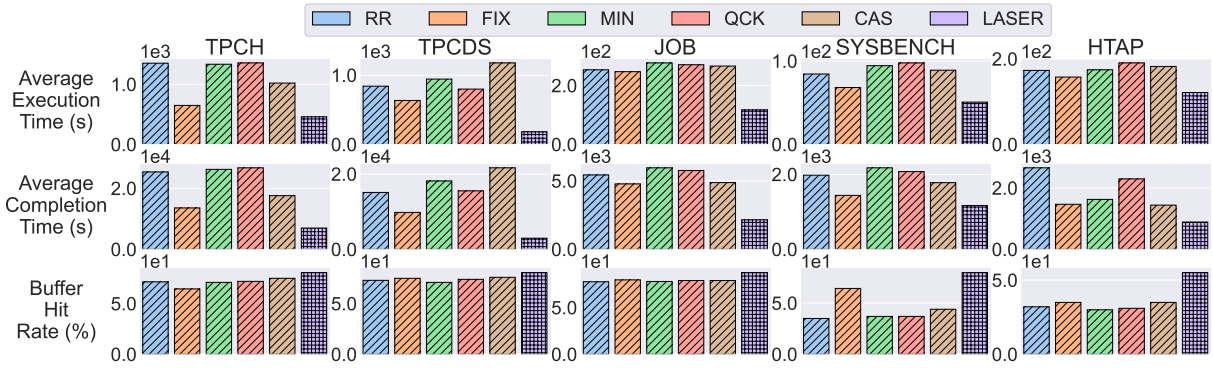

Figure 9: Experiment results under the dynamic setting.

Table 1: Time consumed by *Laser* processes. Numbers in bold are averaged consumptions.

#Queries	Plan Retrieval		Model Prediction		Query Allocation		Query Selection	Re-allocation	
	Sum.	Avg.	Sum.	Avg.	Sum.	Avg.	Avg.	Sum.	Avg.
1000	28.2381s	**0.0282s**	2.7484s	**0.0027s**	2.1029s	**0.0021s**	**0.7995s**	6.1459s	**0.0061s**
100	1.9022s	**0.0190s**	0.2486s	**0.0025s**	0.1867s	**0.0019s**	**0.0786s**	0.4357s	**0.0044s**
10	0.0867s	**0.0087s**	0.0193s	**0.0019s**	0.0185s	**0.0019s**	**0.0079s**	0.0505s	**0.0050s**
1	0.0109s	**0.0109s**	0.0024s	**0.0024s**	0.0023s	**0.0023s**	**0.0020s**	0.0151s	**0.0151s**

7.5 Experiment on Query Arrival Rates

To better understand the differences between static and dynamic scenarios, we conduct experiments with varying query arrival rates. Specifically, we choose the JOB dataset as benchmark and use query arrival rates ranging from 0.1 to 5. We also include the static scenario where $\lambda = \infty$. For performance reference, we compare our *Laser* system with two representative baselines RR and FIX. We keep other settings the same with previous dynamic query scheduling experiments. Figure 10 demonstrates the experiment results.

From the figures, we can find that as the query arrival rate increases, both FIX and *Laser* have a performance gain, while RR roughly stays the same. This is because with higher arrival rates, more connection slots tend to be occupied concurrently, and buffer-friendly strategies may benefit more from parallel execution due

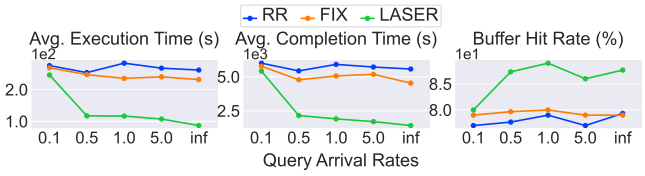

Figure 10: Varying query arrival rates on JOB.

to less data conflicts. Also, for our *Laser* system, a higher arrival rate means more available workload information when making allocation decisions. Note that *Laser* has a performance drop when $\lambda = 0.1$. This is because in such extreme scenarios, queries are

inevitably processed in a near-serialized manner, and many of our key components (adaptive query selection, re-allocation, etc.) can not be fully utilized. However, *Laser* still manages to beat the other 2 baselines with more accurate access pattern predictions.

7.6 Experiment on Degree of Parallelism

In this subsection, we further examine the impact of different degrees of parallelism on the scheduling performance. Specifically, we conduct experiments on the JOB dataset with the number of connections on each database node ranging from 1 to 16. We also use RR and FIX as performance references like before. Other settings are kept the same as in the static query scheduling experiments. Figure 11 demonstrates the experiment results.

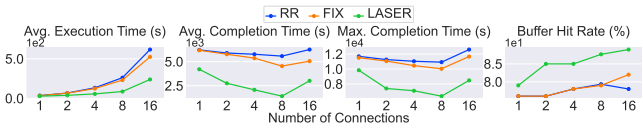

Figure 11: Varying the degree of parallelism on JOB.

Generally speaking, increasing the degree of parallelism may slow down the execution of individual queries due to data and computation conflicts, while also improve the overall throughput with more sufficient utilization of OS resources. This is verified by the first graph where the average execution time of each query grows with the increase of number of connections. When the number of connections is below 8, the advantage of resource utilization is more dominant in that both average completion time and maximal completion time decrease with more connection slots. However, when the number of connections exceeds 8, conflicts between queries become more severe and the performances of all strategies drop. Thanks to better buffer state modeling capabilities, our method still stands out when the other 2 baselines are struggling with data conflicts and resultant buffer misses.

7.7 Experiment on Algorithm Parameters

In the following sections, we conduct experiments on JOB under the same settings as in Section 7.2 unless otherwise specified. To explore the sensitivity of *Laser* to the algorithm parameters, we test its performance by varying the EMA weight α in Equation 9 and the over-allocation ratio in Section 6.1. Figure 12a and 12b illustrate the results. From the figures, we can see that varying both parameters does not severely impact the performance of *Laser*. With a smaller α, the estimation e in Equation 9 focuses more on the current buffer state, while a larger α indicates stronger emphases on recent queries. As it turns out, both strategies work well in our experiments. The over-allocation ratio, on the other hand, is originally designed as a relaxation parameter since the query costs cannot be distributed in a perfectly even manner. According to our observation, this ratio only affects the last few allocation decisions when some nodes have reached their cost upper-bound, thus its impact on the overall performance is relatively negligible. To conclude, the performance of *Laser* is robust under a wide range of parameter settings.

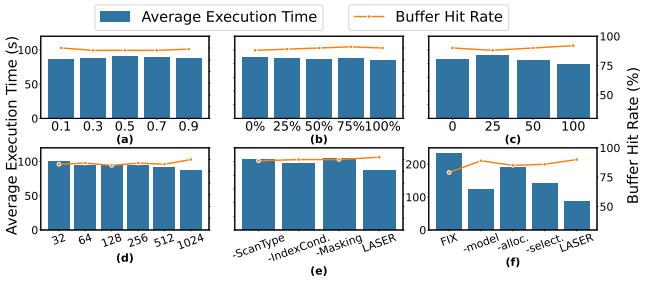

Figure 12: Experiment results on JOB w.r.t. (a) EMA parameter α, (b) Over-allocation ratio, (c) Percentage of pre-trained templates, (d) Access pattern sizes, (e) Ablation study on features, (f) Ablation study on *Laser* components.

7.8 Experiment on Model Pre-training

To verify the effectiveness of *Laser* without pre-training, we compare its performance with 3 variants whose Query Model is pre-trained on 25%, 50%, and 100% of query templates. Figure 12c shows the results on average execution time and shared buffers hit rate. As we can see, both metrics of *Laser* are comparable to those of the variants with pre-training (only 10% longer execution time and 2% lower hit rate compared to fully pre-trained). This is because the model bypass process gives the model a good start point for estimating query access patterns even with randomly initialized parameters. Note that the variant pre-trained with 25% of query templates performs slightly worse, this is probably due to the model's excessive focus on the biased training data. Overall, *Laser*'s performance is still remarkable without model pre-training.

7.9 Experiment on Access Pattern Size

In this subsection, we study the impact of access pattern sizes on query scheduling performances. Specifically, we adopt different number of buckets ranging from 32 to 1024 for query access patterns. Figure 12d illustrates the trends in average execution time and shared buffers hit rate. As we can see, the influence of access pattern sizes on query performances is relatively insignificant. With 32× more buckets (from 32 to 1024), the average execution time only reduces by 13%. This is probably due to the fact that queries in the benchmark are generated by a limited number of templates, and a small access pattern size is sufficient to identify the differences between those templates, thus producing a decent performance gain. With larger access pattern sizes, the intra-template query similarity information are better captured, and the query performances continue growing. However, considering the relatively small benefits brought by larger access patterns, it is advisable to use a small number of buckets when computational overhead for *Laser* become non-negligible compared to the actual query execution time.

7.10 Ablation Study for Scan Features

In this subsection, we study the importance of the scan features including (i) Scan Type, (ii) Index Condition, (iii) Table Name and Index Name for masking. Specifically, we replace each of these features with zero vectors and measure the scheduling performance of

Laser. We disable model bypass here for more direct comparison. Figure 12e demonstrates the experiment results. As we can see, removing any of these features would result in a performance drop in terms of both execution time and buffer hit rate. This agrees with our intuition that with more features, the query model is able to learn more accurate access information. The decrease in scheduling performance is, however, not that significant as in Section 7.9. The reason behind this is also similar, i.e., the query model is able to roughly identify different query templates using a subset of features.

7.11 Ablation Study for Applied Techniques

To better investigate how much each applied technique in *Laser* contributes to performance improvements, we make an ablation study. Specifically, we measure the performance of 3 variants of *Laser*: (i) w/o Query Model (use a table bitmap as each query's access pattern instead), (ii) w/o Adaptive Allocation (use RR instead), (iii) w/o Adaptive Selection (use FIFO instead). We also include the best-performing baseline FIX as a reference for comparison. Figure 12f shows the experiment results on average execution time and shared buffers hit rate. As we can see, without the above 3 components, the average execution time of queries increases by 41.7%, 117.3%, and 63.5% respectively. The corresponding shared buffers hit rate also drops by 1% to 5%. Therefore, we can conclude that all techniques in *Laser* are important for performance improvements. It is also worth noting that all 3 variants are still superior to FIX.

7.12 Experiment on SSDs

In this subsection, we study the impact of faster storage devices (SSDs) on the overall performance trend. Specifically, we conduct the same experiments as in Section 7.2 on TPC-H using a SATA SSD whose typical sequential read speed is 500 MB/s. Figure 13 shows the experiment results. As we can see, our proposed method Laser still outperforms all of the baselines in the 4 metrics, which verifies the effectiveness of *Laser* on faster storage devices. The relative performance differences between different methods, however, become less significant. This is due to the fact that with SSDs, the time penalty caused by buffer misses is smaller than using HDDs. Also, with faster devices, the proportion of query execution time spent on reading data and dealing with conflicts vastly decreases.

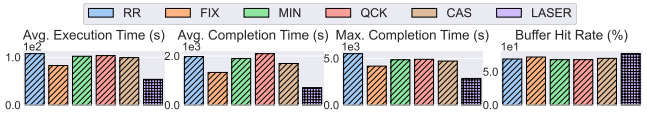

Figure 13: Experiment results on TPC-H using SSDs.

8 RELATED WORK

Learned Database components. Learning-based database components have drawn massive attention and shown outstanding performances [42]. Common research interests in this direction include learned knob tuning [14, 30, 41], learned indexes [4, 12, 16], and learned cardinality estimation [11, 34, 36]. Regarding learned query scheduling, Zhang et al. [40] and Sabek et al. [26] both come up with solutions based on Reinforcement Learning, addressing the issue of single-server scheduling with different levels of granularity. Amazon Auto-WLM [27] focuses on concurrency scaling and short query acceleration in database clusters with the help of learned query latencies. To the best of our knowledge, there is currently no existing work that applies machine learning techniques for buffer-aware query scheduling in master-standby database scenarios.

Database Query Scheduling. Query scheduling is a classic problem in the database field, whose scopes range from single-server scheduling [26, 32, 40] to multi-server scheduling [6, 9, 18, 20], operator-level scheduling [7, 8, 17] to query-level scheduling [3, 5, 22], and static query set scheduling [40] to dynamic query stream scheduling [26, 27]. Our work focuses on dynamic query-level scheduling in master-standby replication database scenarios. In this regard, prior works mainly apply heuristic-based strategies. For example, Waas et al. [31] address a special type of workload where access patterns of queries are easily accessible, and make scheduling decisions based on both query costs and server-query distances. Rohm et al. [25] propose to extract query signatures from SQL texts, based on which they calculate query similarities and make scheduling decisions according to cache benefits. Phan et al. [22] utilize Materialized Query Table (MQT) to reduce query execution time on each DB server and propose searching heuristics to assign different servers with different MQTs and queries. The above solutions are promising under their respective settings, however, they more or less require prior knowledge about the workload and thus cannot quickly learn from or adapt to unseen load patterns. Also, these works mainly consider read-only workloads while write queries are also common in master-standby databases, which may cause database shifts during runtime.

Locality-aware Query Processing. Locality awareness is an important factor in multi-server query processing, as locally cached data can be accessed much faster than from disk or through network connection. Existing literature has explored approaches to improve data locality from various aspects, including data partitioning [39], query scheduling [38], and query evaluation [35]. Our *Laser* system differs from this line of work in that we consider master-standby database scenarios where each node, despite holding a full copy of the database, may have non-static local data that is constantly changing in database buffer.

9 CONCLUSION

In this paper, we study the problem of efficient query scheduling in master-standby database scenarios. We formulate this problem under both static and dynamic settings and make analysis of some possible optimization directions. Based on the insights, we propose our *Laser* system, which can maximize buffer utilization while also maintaining load balance. The system incorporates a lightweight query model that can map a query to its access patterns, a set of self-adaptive algorithms that can make wise and timely scheduling decisions, and also mechanisms to train the model and monitor database shifts on the fly. Experiments on different datasets and workloads verify the effectiveness of our system against traditional heuristic-based scheduling strategies.

REFERENCES

[1] TPC-DS Benchmark. 2021. https://www.tpc.org/tpcds/. [Accessed 2023-05].

[2] TPC-H Benchmark. 2022. https://www.tpc.org/tpch/. [Accessed 2023-05].

[3] 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.

[4] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. 2020. ALEX: an updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 969–984.

[5] AA Diwan, S Sudarshan, and Dilys Thomas. 2006. Scheduling and caching in multi-query optimization. In *International Conference on Management of Data COMAD, Delhi, India*.

[6] Youngmoon Eom, Jinwoong Kim, and Beomseok Nam. 2015. Multi-dimensional multiple query scheduling with distributed semantic caching framework. *Cluster Computing* 18 (2015), 1141–1156.

[7] Minos N Garofalakis and Yannis E Ioannidis. 1996. Multi-dimensional resource scheduling for parallel queries. *ACM SIGMOD Record* 25, 2 (1996), 365–376.

[8] Jana Giceva, Gustavo Alonso, Timothy Roscoe, and Tim Harris. 2014. Deployment of query plans on multicores. *Proceedings of the VLDB Endowment* 8, 3 (2014), 233–244.

[9] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. 2009. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. 261–276.

[10] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[11] 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).

[12] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. 2020. RadixSpline: a single-pass learned index. In *Proceedings of the third international workshop on exploiting artificial intelligence techniques for data management*. 1–5.

[13] 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.

[14] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[15] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. opengauss: An autonomous database system. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3028–3042.

[16] Pengfei Li, Hua Lu, Qian Zheng, Long Yang, and Gang Pan. 2020. LISA: A learned index structure for spatial data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 2119–2133.

[17] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, et al. 2021. Greenplum: a hybrid database for transactional and analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 2530–2542.

[18] 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.

[19] MySQL. 1995. https://www.mysql.com/. [Accessed 2023-05].

[20] Beomseok Nam, Minho Shin, Henrique Andrade, and Alan Sussman. 2010. Multiple query scheduling for distributed semantic caches. *J. Parallel and Distrib. Comput.* 70, 5 (2010), 598–611.

[21] Adam Paszke, Sam Gross, Francisco 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, Sasank 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 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[22] Thomas Phan and Wen-Syan Li. 2008. Load distribution of analytical query workloads for database cluster architectures. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. 169–180.

[23] PostgreSQL. 1996. https://www.postgresql.org/. [Accessed 2023-05].

[24] Psycopg2. 2010. https://www.psycopg.org/. [Accessed 2023-05].

[25] Uwe Rohm, Klemens Bohm, and H-J Schek. 2001. Cache-aware query routing in a cluster of databases. In *Proceedings 17th International Conference on Data Engineering*. IEEE, 641–650.

[26] 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.

[27] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan Narayanaswamy. 2023. Auto-WLM: Machine learning enhanced workload management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data*. 225–237.

[28] Swarm64. 2020. https://github.com/swarm64/s64da-benchmark-toolkit. [Accessed 2024-08].

[29] Sysbench. 2020. https://github.com/akopytov/sysbench. [Accessed 2023-05].

[30] Dana Van Aken, Andrew Pavlo, Geoffrey J Gordon, and Bohan Zhang. 2017. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM international conference on management of data*. 1009–1024.

[31] Florian Waas and Martin L Kersten. 2000. *Memory aware query scheduling in a database cluster*. CWI (Centre for Mathematics and Computer Science).

[32] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-tuning query scheduling for analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 1879–1891.

[33] Fang Wang, Xiao Yan, Man Lung Yiu, Shuai LI, Zunyao Mao, and Bo Tang. 2023. Speeding Up End-to-end Query Execution via Learning-based Progressive Cardinality Estimation. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–25.

[34] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A normalizing flow based cardinality estimator. *Proceedings of the VLDB Endowment* 15, 1 (2021), 72–84.

[35] Qiufen Xia, Weifa Liang, and Zichuan Xu. 2014. Data locality-aware query evaluation for big data analytics in distributed clouds. In *2014 Second International Conference on Advanced Cloud and Big Data*. IEEE, 1–8.

[36] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep unsupervised cardinality estimation. *arXiv preprint arXiv:1905.04278* (2019).

[37] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement learning with tree-lstm for join order selection. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1297–1308.

[38] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. 2010. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In *Proceedings of the 5th European conference on Computer systems*. 265–278.

[39] Erfan Zamanian, Carsten Binnig, and Abdallah Salama. 2015. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 17–30.

[40] Chi Zhang, Ryan Marcus, Anat Kleiman, and Olga Papaemmanouil. 2020. Buffer pool aware query scheduling via deep reinforcement learning. *arXiv preprint arXiv:2007.10568* (2020).

[41] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 international conference on management of data*. 415–432.

[42] Xuanhe Zhou, Chengliang Chai, Guoliang Li, and Ji Sun. 2020. Database meets artificial intelligence: A survey. *IEEE Transactions on Knowledge and Data Engineering* 34, 3 (2020), 1096–1116.