RUSH: Resource-Aware Query Scheduling across Heterogeneous Cloud Compute Services

Wenbo Li
Tsinghua University
Beijing, China
lwb21@mails.tsinghua.edu.cn

Haoqiong Bian
Renmin University, China
Beijing, China
bianhq@ruc.edu.cn

Chao Zhang
Renmin University, China
Beijing, China
cycchao@ruc.edu.cn

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

ABSTRACT

Modern cloud platforms provide a wide range of compute services, including virtual machines, cloud functions, and Query-as-a-Service, each offering unique trade-offs in performance, elasticity, and cost. While these services are suitable for diverse types of OLAP workloads, existing systems typically rely on a single service type or apply static rule-based scheduling strategies, failing to harness the complementary strengths of heterogeneous services and adapt to the dynamic OLAP workloads. The key challenges of scheduling OLAP queries across heterogeneous services lie in (1) routing queries to suitable services, (2) minimizing resource contention among queries within each service, and (3) balancing load across services under dynamic workloads.

In this paper, we present RUSH, a resource-aware query scheduling framework designed to efficiently execute OLAP queries across heterogeneous cloud compute services. RUSH addresses these challenges through resource-aware scheduling. First, it predicts query resource demands at the pipeline level to guide routing decisions. Second, it employs a resource-aware intra-service scheduler to reduce contention within each service. Third, it introduces a load-aware inter-service scheduler that dynamically reallocates long-waiting queries across services with minimal performance overhead. By integrating these mechanisms, RUSH achieves substantial improvements in both performance and cost efficiency. Experiments demonstrate that RUSH outperforms state-of-the-art approaches, reducing query latency by up to 35% and cutting costs by up to 67%.

1 INTRODUCTION

Modern cloud platforms provide a range of compute services, each optimized for specific types of workloads. For example, virtual machines (VMs) [3, 5, 6] offer high and stable performance, rendering them well-suited for long-running queries with predictable resource usage. Cloud functions (CFs) [8, 10, 11] provide high elasticity and fine-grained billing, making them ideal for short-lived bursty workloads. Query-as-a-Service (QaaS) [4, 9, 13] platforms eliminate the need for infrastructure pre-provisioning, enabling on-demand execution of ad-hoc analytical queries. As summarized in Table 1, these services differ significantly in terms of startup time, billing granularity, and concurrency limits, making them suitable for different use cases.

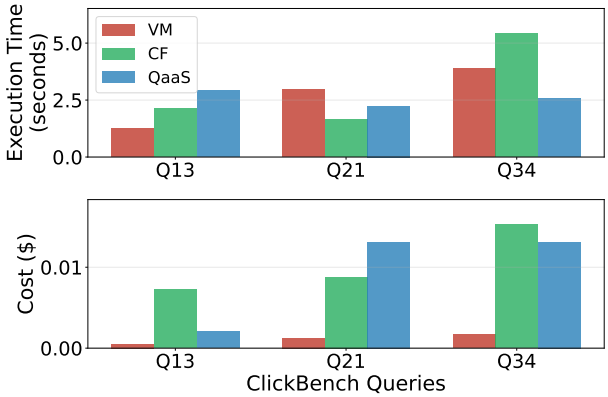


Figure 1: Execution time and cost of three representative ClickBench [14] queries across different compute services.

However, each service has inherent limitations that lead to severe performance degradation when handling mismatched workloads. Specifically, VMs suffer from high startup delays, which can lead to query backlogs during sudden workload spikes. CFs lack native support for inter-function communication, making them inefficient for complex queries involving large-scale data exchanges. Most QaaS platforms [4, 9] adopt a pay-per-data-scanned pricing model, incurring prohibitive costs for queries that process large datasets.

Existing systems often operate within a single type of service [27, 31] or employ simplistic scheduling strategies [18, 30], neglecting the compatibility between query characteristics and service capabilities. Consequently, queries are frequently executed on suboptimal services, resulting in poor performance and high costs.

This mismatch highlights the need for a more adaptive scheduling framework that leverages the complementary strengths of different compute services while mitigating their weaknesses. However, designing such a system presents several key challenges:

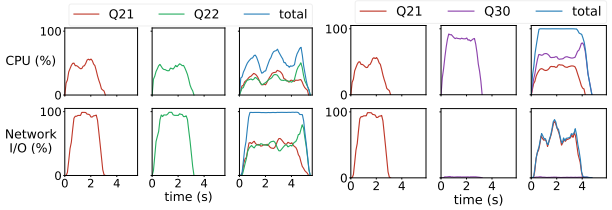
C1. How to route queries with complex resource demands to the most suitable service? OLAP queries exhibit complex and varying resource requirements, from memory-intensive joins to I/O-heavy scans. Meanwhile, cloud compute services offer different performance characteristics, pricing models, and capacities. As shown in Figure 1, these differences result in large variations in performance and cost for the same query across different services. Effective routing must consider both fine-grained query resource demands and the capabilities of available services, going beyond simple rules or heuristics that fail to capture their interaction.

C2. How to minimize resource contention within each service given their different capacity models? Concurrent queries

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Table 1: Comparison of cloud compute services, illustrated on AWS

Dimension	Attribute	VM [3, 5, 6]	CF [8, 10, 11]	QaaS [4, 9, 13]
Elasticity	Startup time	Long (tens of seconds)	Short (several milliseconds)	Medium (sub-second)
Cost	Billing granularity	Per second	Per millisecond	Per scanned data volume
Capacity	Concurrency limit	Limited resources	Limited function concurrency	Limited query concurrency
Performance	Best use case	Long-running queries	Bursty short queries	Ad-hoc analytical queries



(a) Q21 and Q22 (b) Q21 and Q30

Figure 2: CPU and network I/O usage of two ClickBench query pairs executed individually and concurrently on a VM. For each pair, the left and middle columns show the resource usage when queries run individually, and the right column shows total and per-query resource usage when they run concurrently.

on the same service compete for shared resources such as CPU, memory, and I/O bandwidth, leading to increased latency. As shown in Figure 2, an I/O-intensive query (Q21) nearly doubles its latency when executed concurrently with another I/O-heavy query (Q22), but experiences only minor slowdown when paired with a CPU-bound query (Q30). This demonstrates that interference is highly dependent on resource usage patterns. However, different services have different resource capacity: VMs face limited physical resources, CFs are bound by function concurrency limits, and QaaS platforms impose query concurrency limits. Designing an effective intra-service scheduler requires tailoring policies to each service's specific resource capacity, making it challenging to develop a unified scheduling framework that works across heterogeneous services.

C3. How to balance load across services while minimizing efficiency loss? Even with effective routing and intra-service scheduling, sudden traffic spikes can overwhelm specific services, leading to growing query backlogs. As shown in Figure 3, relying solely on auto-scaling VMs results in long provisioning delays and query accumulation. In contrast, when inter-service scheduling is enabled, long-waiting queries are dynamically reallocated to underutilized services, which handle the load spike more efficiently. However, such reallocation must be done carefully, as a service with available capacity may be poorly suited to the query's resource demands. For example, moving a scan-heavy query to QaaS or a complex query with multiple join operations to CF can lead to higher execution time and cost. Therefore, effective inter-service scheduling must consider not only system utilization but also whether the target service can efficiently execute the query, which makes it challenging to balance load while minimizing efficiency loss.

To tackle these challenges, we propose RUSH, a **R**esource-aware q**U**ery **S**cheduling framework for **H**eterogeneous cloud compute services. To address **C1**, we decompose each query into execution pipelines and predict fine-grained resource usage of each pipeline stage to guide routing decisions (Section 4). To address **C2**, we model query resource demands in a way that accounts for the specific capacity of each service, and design a resource-aware intra-service scheduling algorithm that co-schedules queries with complementary resource demands to reduce contention (Section 5). To address **C3**, we introduce a load-aware inter-service scheduler that leverages query resource characteristics to identify beneficial query migrations, enabling efficient load balancing while minimizing performance loss (Section 6).

In summary, this paper makes the following contributions.

- We present RUSH, a resource-aware query scheduling framework on heterogeneous cloud compute services that addresses the challenges of service mismatch, intra-service interference, and dynamic load imbalance through resource-aware scheduling.
- We develop a fine-grained, pipeline-level resource modeling approach that predicts resource demands for each query stage, enabling accurate and lightweight service selection without full execution.
- We design a unified intra-service scheduling algorithm that leverages service-adaptive resource modeling to co-schedule queries with complementary resource demands, minimizing interference within each compute service.
- We propose a load-aware inter-service scheduler that leverages runtime system pressure and lightweight suitability estimation to migrate long-waiting queries across services, mitigating performance degradation under bursty workloads.
- We implement and evaluate RUSH on a range of analytical workloads, demonstrating that it significantly reduces query latency by up to 35% and lowers costs by up to 67% compared to state-of-the-art approaches.

2 PRELIMINARIES

In this section, we first present key characteristics of cloud compute services commonly used for OLAP workloads, highlighting their performance, cost, and elasticity properties. Then we formally define the problem of resource-aware query scheduling across heterogeneous services, which serves as the foundation of RUSH.

2.1 Characteristics of Cloud Compute Services

Modern cloud platforms offer a diverse set of compute services, each optimized for distinct workload patterns. This paper focuses on three widely adopted service types for OLAP workloads: virtual machines, cloud functions, and Query-as-a-Service.

Virtual machines (VMs) provide dedicated compute resources with predictable performance and customizable hardware configurations, making them well-suited for resource-intensive and long-running OLAP queries. However, VMs exhibit limited elasticity. Despite support for auto-scaling, provisioning new instances takes tens of seconds to several minutes, rendering them ineffective for

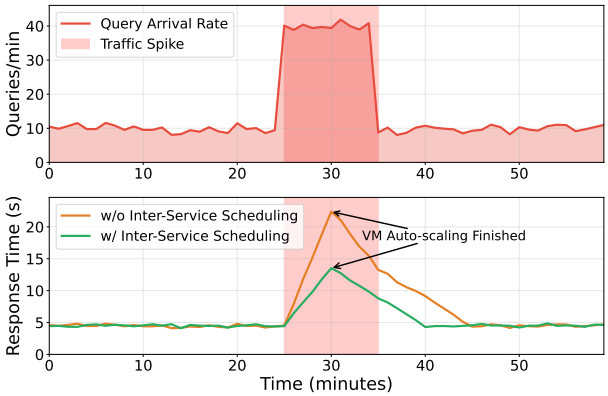


Figure 3: Query arrival rate and average response time under a bursty workload. The green curve shows the effect of inter-service scheduling in reducing latency during load spikes.

handling sudden workload spikes. As a result, bursty queries must contend with fixed resource capacity, leading to resource contention and performance degradation.

Cloud functions (CFs) provide a serverless execution model with fine-grained billing and rapid elasticity, making them highly suitable for short-lived and bursty workloads. They can scale out quickly in response to sudden query bursts, launching hundreds of instances within seconds. However, CFs face significant limitations for complex OLAP queries. They are constrained in per-instance resources (e.g., up to 6 vCPUs and 10 GB memory on AWS Lambda [8]) and execution duration (e.g., limited to 15 minutes for AWS Lambda), forcing complex queries to launch hundreds of concurrent instances to achieve acceptable performance. Moreover, the total number of concurrent instances is capped (e.g., 1000 by default on AWS), limiting scalability under heavy loads. Critically, CFs lack direct inter-function communication, requiring intermediate data exchange through external storage systems such as S3 [2], which introduces significant latency overhead.

Query-as-a-Service (QaaS) enables users to execute complex OLAP queries without managing underlying infrastructure, as it automatically handles resource provisioning and distributed query execution. Unlike CFs, where distributed operations such as joins and shuffles must be manually implemented using external storage for intermediate data exchange, QaaS natively supports these operations, making it suitable for complex OLAP queries with large intermediate data exchange. However, QaaS platforms typically charge based on the amount of data scanned, which can lead to high costs for queries that process datasets. Furthermore, the internal execution behavior is not exposed to users, limiting the opportunities for performance tuning.

2.2 Problem Statement

We consider the problem of scheduling dynamically arriving OLAP queries across heterogeneous cloud compute services. All data is stored in a shared cloud storage system (e.g., Amazon S3 [2]) accessible to all compute services. For each query, the system must determine which compute service to execute the query and when to start its execution. The goal is to jointly minimize query latency and execution cost, achieving an optimal cost-performance trade-off

Table 2: Notations

Notation	Description	Notation	Description
Q	List of queries	α	Cost-performance trade-off parameter
q	A query in Q	p_{ij}	j-th pipeline stage of q_i
q_i	i-th query in Q	$\mathcal{Q}^{(s)}$	Set of pending queries on s
S	Set of compute services	$\mathcal{A}(t)$	Set of active queries at time t
s	A service in S	$\hat{\mathbf{r}}_{ij}$	Est. resource demand vector of p_{ij}
s_i	The service q_i execute on	$\mathbf{c}^{(s)}_{\text{capacity}}$	Resource capacity vector of s
s_v	VM service	C_i	Completion time of q_i
s_f	CF service	$\hat{\mathbf{R}}_q$	Est. resource demand matrix of q
s_q	QaaS service	$\hat{\mathbf{r}}_q$	Flattened vector of $\hat{\mathbf{R}}_q$
$T_{q,s}$	Execution time of q on s	$\hat{\mathbf{C}}^{(s)}_{\text{remain}}$	Est. remaining resource capacity matrix of s
$M_{q,s}$	Monetary cost of q on s	$\hat{\mathbf{c}}^{(s)}_{\text{remain}}$	Flattened vector of $\hat{\mathbf{C}}^{(s)}_{\text{remain}}$
$\hat{T}_{q,s}$	Est. execution time of q on s	$W^{(s)}_q$	Waiting time of q in the queue of s
$\hat{M}_{q,s}$	Est. monetary cost of q on s	$\hat{\mathbf{C}}^{(s)}_{\text{capacity}}$	Resource capacity of s over time slots
a_i	Arrival time of q_i	$\hat{\mathbf{C}}^{(s)}_{\text{used}}$	Est. used resources of s over time slots
t_i	Execution start time of q_i	$W^{(s)}$	Est. waiting time of s

under dynamic workloads. Table 2 summarizes the key notations used in this paper.

Let $Q = \{q_1, q_2, \ldots, q_n\}$ be the set of OLAP queries to be scheduled, where each query q_i arrives at time a_i. The system must assign each query to one of the heterogeneous compute services $S = \{s_v, s_f, s_q\}$, where s_v denotes VMs, s_f denotes CFs, and s_q denotes QaaS. For any query q_i and service $s_i \in S$, we define T_{q_i,s_i} and M_{q_i,s_i} as the runtime and monetary cost of executing q_i on s_i respectively. In addition, each service has capacity constraints that limit the number of concurrent queries it can execute.

For each incoming query q_i, we must make two decisions: the target compute service $s_i \in S$ and the start time $t_i \geq a_i$ at which the query begins execution on s_i. The execution of q_i completes at time $t_i + T_{q_i,s_i}$, resulting in a response time of $t_i + T_{q_i,s_i} - a_i$. The objective is to jointly optimize query response time and monetary cost by minimizing the performance-cost score, defined as:

$$\sum_{i=1}^{n} \left(t_i - a_i + T_{q_i,s_i}\right) \cdot M^{\alpha}_{q_i,s_i} \tag{1}$$

where $\alpha \geq 0$ is a tunable parameter that controls the trade-off between cost and performance. This formulation follows the approach adopted in cost-aware scheduling systems such as RAIS [28], where α is typically specified by the user to reflect their performance-cost preferences. A higher value of α emphasizes cost minimization, while a lower value prioritizes reducing query latency.

3 SYSTEM OVERVIEW

RUSH is a resource-aware query scheduling framework that dynamically schedules OLAP queries across heterogeneous cloud compute services to optimize cost-performance trade-offs. As Figure 4 shows, RUSH acts as a proxy layer between clients and a pool of cloud services, with all data stored in shared cloud storage. In this architecture, all compute services access the cloud storage for input data, and CFs also use it to exchange intermediate results during multi-stage query execution. The decoupling of storage and compute allows RUSH to treat all compute services as stateless execution units, eliminating dependencies on local state and enabling flexible query scheduling and migration across services. RUSH performs multi-level scheduling. It predicts query resource demands at the pipeline level to guide initial routing, employs a resource-aware intra-service scheduler to reduce contention within each service, and applies a load-aware inter-service scheduler to migrate long-waiting queries across services when imbalance occurs. The following paragraphs detail these three main components.

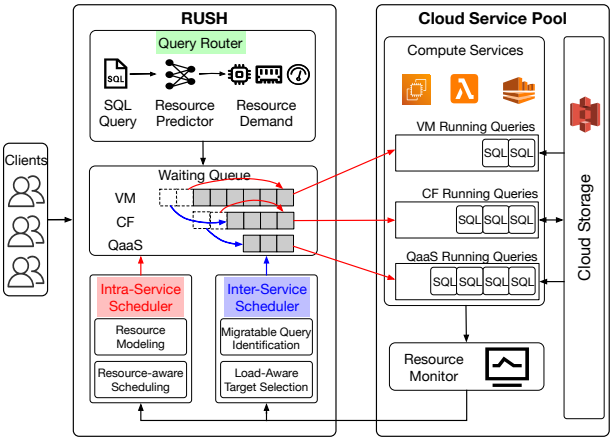


Figure 4: System overview of RUSH.

Query Router. The query router determines the initial compute service for each incoming query based on predicted resource demands. To improve prediction accuracy, it models resource usage at the pipeline level. Each query is decomposed into a sequence of pipeline stages, and the resource demands of each stage are predicted using a lightweight model. These per-pipeline predictions are aggregated into a holistic resource profile, which is used to evaluate compatibility with compute services. The service that best matches the query's profile is selected, and the query is placed in the corresponding waiting queue of that service. The router is designed to be lightweight, enabling efficient handling of a large number of queries with minimal overhead and negligible latency. The implementation details are described in Section 4.

Intra-service Scheduler. The intra-service scheduler manages the query execution within each service to reduce performance interference under concurrency. It maintains a waiting queue for each service and selects queries for execution based on their predicted resource demands and current system load. Leveraging fine-grained resource profiles from the query router, the intra-service scheduler employs a resource-aware scheduling algorithm that co-schedules queries with complementary resource demands to minimize contention. Furthermore, the scheduler periodically re-evaluates the queue based on updated load conditions and query progress, enabling adaptive prioritization to dynamic workloads. The scheduling algorithm is detailed in Section 5.

Inter-service Scheduler. The inter-service scheduler addresses load imbalance across heterogeneous compute services by migrating long-waiting queries that experience prolonged queueing delays. When a query remains pending in an overloaded service beyond a threshold, the scheduler triggers a migration evaluation which assesses whether migrating the query to a less-utilized service would lead to improvement in completion time. This decision is based on a benefit analysis that weighs the potential of reduced queueing delays in a less-loaded service against the risk of inefficient execution due to the mismatch between the query's resource demands and the target service's capability. If the migration is beneficial, the query is moved to the new service's waiting queue, where it can be executed more promptly. By proactively migrating queries across services, the inter-service scheduler improves overall resource utilization and reduces query response time. Section 6 details the design of our migration policy.

In summary, the components of RUSH work together to enable adaptive and resource-aware query scheduling across heterogeneous cloud compute services. The query router makes initial routing decisions using pipeline-level resource predictions. The intra-service scheduler ensures efficient execution by minimizing resource contention within each service. When load imbalance occurs, the inter-service scheduler dynamically migrates long-waiting queries to less-utilized services, reducing queueing delays and improving overall system performance. This modular architecture supports easy integration of new service types and independent refinement of individual components.

4 QUERY ROUTER

The query router in RUSH selects the most appropriate compute service for each OLAP query by evaluating the cost-performance trade-off based on predicted resource demands. To enable accurate prediction, we model resource usage at the pipeline level by decomposing queries into multiple pipeline stages and predicting their individual resource demands. These fine-grained resource predictions are then used to assess how well each service matches the query's requirements. The service that offers the best cost-performance balance is selected for execution. In the following, we first present our pipeline-level resource prediction approach in Section 4.1, followed by the service selection mechanism in Section 4.2.

4.1 Pipeline-level Resource Prediction

Accurate resource prediction is crucial for effective query routing. However, treating the entire query as a single unit often leads to inaccurate predictions, especially when different execution phases exhibit drastically different resource consumption patterns. To address this, we decompose each query plan into a sequence of pipeline stages and model the resource usage of each pipeline individually. This fine-grained approach enables us to capture the varying resource usage patterns across different stages of query execution, leading to more accurate estimations.

Our design is motivated by the observation that while resource demands may fluctuate significantly across pipelines, they tend to be relatively stable within each pipeline. For instance, we consider a query plan consisting of three pipelines, as shown in Figure 5. Pipeline 1 and pipeline 2 perform data scan and hash join operations, incurring high CPU, memory and I/O usage. In contrast, pipeline 3 primarily performs hash aggregation and projection, where CPU usage peaks while memory consumption and I/O throughput drop substantially. This variation highlights the importance of fine-grained, pipeline-level modeling to accurately capture resource usage patterns.

4.1.1 Query Decomposition. We decompose each query plan into multiple pipelines by identifying blocking operators such as hash joins and aggregations. These operators typically require their input to be fully consumed before producing output. This forces materialization of intermediate results, which naturally defines the boundary between consecutive pipeline stages. Each pipeline consists of a sequence of non-blocking operators (e.g., filters, projections, and scans) that typically exhibit relatively stable resource

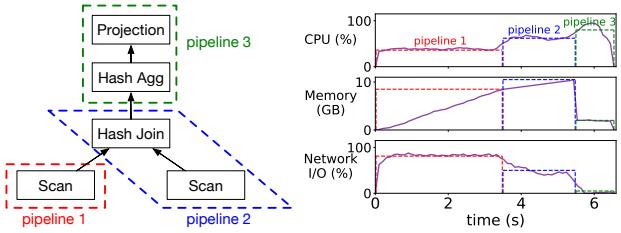


Figure 5: A query plan decomposed into three pipelines and its corresponding resource usage over time.

usage patterns. Our decomposition follows the execution model widely adopted in modern OLAP systems like DuckDB [12] and Presto [7], which organize query execution into pipelines to improve parallelism and resource utilization.

4.1.2 Pipeline Encoding. Prior work [21, 26, 35, 42] has proposed various query encoding methods, each designed for specific downstream tasks such as query performance prediction and cardinality estimation. In our case, the goal is to predict the resource usage of each pipeline stage. To achieve this, we design a task-specific pipeline encoding that extracts informative features from the query plan with low computational overhead. We extract a fixed-dimension feature vector for each pipeline by summarizing properties across all operator types. For each operator type, we collect three general features: the operator count within the pipeline, the total estimated input cardinalities, and the total estimated data sizes processed by the operator.

In addition, we incorporate operator-specific features to capture computational intensity not reflected in general features. For example, filter operators may have multiple predicates with different complexities, which can significantly affect CPU overhead. Similarly, aggregation operators with complex functions (e.g., `COUNT(DISTINCT)`) incur higher CPU and memory pressure than simple sums or averages. By encoding these operator-specific features, we improve the model's ability to distinguish between pipelines that are structurally similar but exhibit divergent resource usage patterns. A list of these features is provided in Table 3. This set of features is extensible and can be expanded to include additional operator-specific characteristics as needed.

For complex operators such as hash join that exhibit distinct resource behaviors across different execution stages, we further split them into sub-operators (e.g., hash build and hash probe) and extract their features independently. This refinement provides a more accurate representation of these operators by explicitly capturing their multi-stage behavior. These features together form a compact yet expressive representation of each pipeline, enabling both lightweight processing and accurate resource prediction.

4.1.3 Resource Usage Prediction. We model the resource usage of each pipeline as a multi-dimensional vector consisting of four key metrics: average CPU utilization, peak memory usage, average network I/O throughput, and execution duration. These metrics reflect the resource demands that influence query performance and cost across heterogeneous compute services. Our goal is to predict these values based on the encoded pipeline features, capturing how pipeline characteristics influence resource consumption.

To implement this prediction task, we adopt XGBoost [19], a gradient boosting decision tree framework widely used in query

Table 3: Operator-Specific Features for Computational Intensity Modeling

Operator	Feature	Description
Filter	like_cost	Est. cost of `LIKE` predicates, calculated as the number of wildcards in the pattern multiplied by the average length of the corresponding string column.
Aggregation	basic_agg_size	Total number of rows and data size processed by `MAX` and `MIN`.
	arith_agg_size	Total number of rows and data size processed by `SUM`, `COUNT`, and `AVG`.
	distinct_agg_card	Est. cardinality of distinct keys used in `COUNT(DISTINCT)` and similar aggregations.
	group_key_card	Est. cardinality of distinct combinations of `GROUP BY` keys.
Projection	string_replace_cost	Est. cost of string replacement operations (e.g., `REGEXP_REPLACE`), calculated by summing the replacement pattern complexity (defined as the pattern's length multiplied by the number of wildcards) multiplied by the average length of the corresponding string column.

performance prediction tasks [33, 40]. We choose XGBoost for its lightweight architecture and fast inference speed, which is essential for online query routing to avoid introducing significant latency. Compared to more complex deep learning models [21], XGBoost introduces significantly lower training and inference overhead, making it more suitable for online query routing where low-latency decisions are required.

To predict the overall resource usage of a query, we aggregate predictions across all its constituent pipelines. For CPU and I/O, we concatenate predicted values in execution order, forming a piecewise linear approximation of resource consumption over time. For memory, we take the maximum peak memory usage across all pipelines, as exceeding available memory can severely degrade performance or cause execution failures. This conservative aggregation ensures that sufficient memory is reserved throughout execution, avoiding costly disk spilling or out-of-memory errors. The overall workflow of resource usage prediction is illustrated in Figure 6.

4.2 Service Selection

The service selection mechanism assigns each query to the most suitable compute service based on its predicted resource usage and the performance and cost characteristics of the services. We quantify the compatibility between a query and each service using a structured model built upon fine-grained resource predictions. This enables the query router to select the service that best balances execution efficiency and monetary cost. In the following, we first

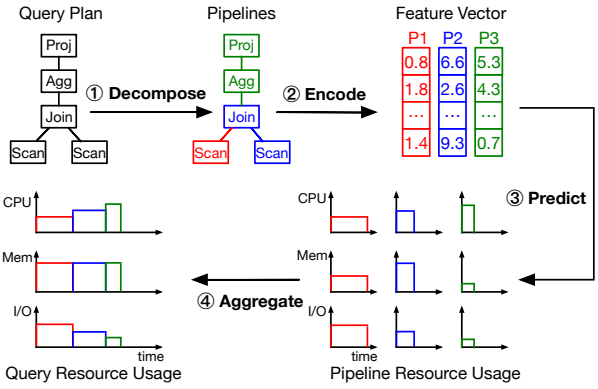


Figure 6: Workflow of resource usage prediction in RUSH.

formalize the compatibility modeling on each service and then present the decision process for service selection.

4.2.1 Virtual Machines. The expected time of executing a query q on a VM, denoted as $\hat{T}_{q,s_v}$, is predicted using the pipeline-level resource modeling approach from Section 4.1. Since VMs are billed per second, compute cost is proportional to execution duration. However, simply multiplying $\hat{T}_{q,s_v}$ by the billing rate R_{vm} would overestimate the query's monetary cost, as it assumes the query exclusively occupies the VM for its entire duration. In reality, multiple queries can execute concurrently and share resources on a VM. To avoid overestimation, we model each query's compute cost based on its dominant resource footprint, defined as the highest fraction of CPU, memory, and network I/O resource demands relative to the VM's capacity across all pipeline stages. This ratio, denoted as ρ_q, serves as an estimate of the query's share of the total VM cost.

Although concurrent execution may introduce interference, RUSH's intra-service scheduler (will be described in Section 5) minimizes contention by co-scheduling queries with complementary resource demands. This ensures that execution time remains close to the predicted value $\hat{T}_{q,s_v}$, making the estimate reliable even under concurrency.

In addition, queries executed on VMs typically read input data from cloud storage, generating GET requests. These requests are billed at a fixed rate per request, denoted as R_{get}. Let $\hat{N}_{\text{get}}^{(q,s_v)}$ represent the total number of GET requests issued during query execution. The total monetary cost of executing a query q on VM is modeled as:

$$\hat{M}_{q,s_v} = \rho_q \cdot \hat{T}_{q,s_v} \cdot R_{\text{vm}} + \hat{N}_{\text{get}}^{(q,s_v)} \cdot R_{\text{get}} \tag{2}$$

4.2.2 Cloud Functions. In RUSH, each pipeline stage executed on CFs is parallelized across multiple function instances, with intermediate results shuffled through cloud storage between stages. This execution model follows the serverless execution paradigm in [18], where the number of instances allocated to each stage p, denoted as N_p, is determined based on the sizes and rows of data processed by the stage. For a given stage p, let $\hat{D}_{\text{cpu}}^{(p)}$ and $\hat{D}_{\text{io}}^{(p)}$ denote the predicted volume of data processed by CPU operations and I/O operations, respectively. The expected execution time of the stage is modeled as:

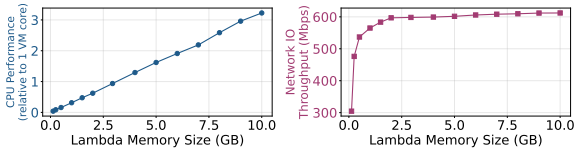


Figure 7: CPU performance and network throughput of AWS Lambda functions with different memory configurations.

$$\hat{T}_p = \max\left(\frac{\hat{D}_{\text{cpu}}^{(p)}}{N_p \cdot V_{\text{cpu}}(m)}, \frac{\hat{D}_{\text{io}}^{(p)}}{N_p \cdot V_{\text{io}}(m)}\right) \tag{3}$$

where m denotes the memory size allocated to each function instance, which determines its CPU and I/O throughput, denoted as $V_{\text{cpu}}(m)$ and $V_{\text{io}}(m)$ respectively. As shown in Figure 7, $V_{\text{cpu}}(m)$ scales nearly linearly with m, while $V_{\text{io}}(m)$ increases with memory up to 2 GB and plateaus thereafter. To minimize cost and maximize performance, we choose the smallest memory configuration that minimizes $\hat{T}_{\text{stage}}$. For I/O intensive stages, this typically results in choosing around 2-4 GB of memory, as further memory allocation yields no benefit. In contrast, CPU-intensive stages often require higher memory settings to fully utilize available compute capacity.

Between stages, intermediate data is shuffled through shared storage. The upstream stage, executed on $N_{p_{\text{up}}}$ instances, writes intermediate results of size $\hat{D}_{\text{write}}^{(p_{\text{up}})}$; the downstream stage, executed on $N_{p_{\text{down}}}$ instances, reads data of size $\hat{D}_{\text{read}}^{(p_{\text{down}})}$. These sizes are estimated from the query plan and statistical metadata, including cardinalities and average widths of accessed columns. The shuffle time is modeled as the sum of write and read times:

$$\hat{T}_{p_{\text{shuffle}}} = \frac{\hat{D}_{\text{write}}^{(p_{\text{up}})}}{N_{p_{\text{up}}} \cdot V_{\text{io}}(m)} + \frac{\hat{D}_{\text{read}}^{(p_{\text{down}})}}{N_{p_{\text{down}}} \cdot V_{\text{io}}(m)} \tag{4}$$

CFs are billed per-instance based on execution duration and allocated memory. Let m_i and $\hat{l}_i$ be the memory size and predicted execution time of the i-th function invocation respectively, and $R_{\text{cf\_compute}}$ denote the unit billing rate for function execution (in USD per GB-second). Given n total invocations across all stages, the compute cost is modeled as:

$$\hat{M}_{\text{cf\_compute}} = \sum_{i=1}^{n} \hat{l}_i \cdot m_i \cdot R_{\text{cf\_compute}} \tag{5}$$

In addition to compute costs, executing queries on CFs incurs storage access costs from input reads and intermediate data shuffling. Input reads and shuffle outputs generate a number of GET and PUT requests. Let $\hat{N}_{\text{get}}^{(q,s_f)}$ and $\hat{N}_{\text{put}}^{(q,s_f)}$ be the total number of GET and PUT requests across all stages, which are charged at fixed rates per request, denoted as R_{get} and R_{put}. The total storage access cost is modeled as:

$$\hat{M}_{\text{cf\_storage}} = \hat{N}_{\text{get}}^{(q,s_f)} \cdot R_{\text{get}} + \hat{N}_{\text{put}}^{(q,s_f)} \cdot R_{\text{put}} \tag{6}$$

The total execution time and monetary cost for a query executed on CFs are calculated as:

$$\hat{T}_{q,s_f} = \sum_{p \in \text{stages}} \hat{T}_p + \sum_{f \in \text{shuffles}} \hat{T}_f \tag{7}$$

$$\hat{M}_{q,s_f} = \hat{M}_{\text{cf\_compute}} + \hat{M}_{\text{cf\_storage}} \tag{8}$$

Our model focuses on predictable and measurable components that are sufficient for effective service selection, omitting highly variable factors like cold start latency and straggler effects, which are difficult to estimate accurately. However, these issues can be

mitigated by existing optimizations, such as pre-warming [24] and duplicate requests [31], and their impact is typically bounded. Therefore, our model provides a reliable basis for service selection without being overly sensitive to these unpredictable factors.

4.2.3 Query-as-a-Service. The cost of executing a query on QaaS is exclusively determined by the volume of data scanned, which can be estimated from the query plan and statistical metadata by analyzing accessed columns, their average widths, and cardinalities. The resulting data size provides a lightweight yet effective approximation for monetary cost.

As resource allocation in QaaS is opaque to users, execution time cannot be modeled based on resource usage like in VM and CF settings. Instead, we employ a prediction model trained on historical execution data, which uses the same architecture and query encoding mechanism as the pipeline-level resource prediction model described in Section 4.1. We extend the input features with the estimated data size to be scanned, as we observe that the computational resources provisioned by QaaS are closely related to the amount of data processed. This enhancement enables the model to better capture platform behavior and produce more accurate runtime predictions.

4.2.4 Service Selection. For each incoming query q, the query router selects the most suitable compute s^* service by minimizing the load-aware cost-performance objective function:

$$s^* = \underset{s \in S}{\operatorname{argmin}} \left(\hat{T}_{q,s} \cdot \hat{M}_{q,s}^{\alpha} \cdot (1 + \beta_s \cdot L_s) \right) \tag{9}$$

where S is the set of available compute services, $\hat{T}_{q,s}$ and $\hat{M}_{q,s}$ denote the estimated execution time and monetary cost of the query q on service s, and α is the user-defined parameter that reflects the user's trade-off preference described in Section 2.2. To prevent service overloading, we incorporate a lightweight load balance penalty based on the number of pending queries L_s on service s. The parameter β_s controls the strength of this penalty for each service, allowing users to adjust the balance between performance and load distribution.

5 INTRA-SERVICE SCHEDULER

The intra-service scheduler, manages the execution order to reduce resource contention within each compute service. We begin by introducing the resource modeling approach that captures query resource usage and service capacity, followed by the problem formulation and complexity analysis. Building on these foundations, we present a practical scheduling algorithm that dynamically prioritizes queries based on their resource demands and the current resource availability.

5.1 Resource Modeling

We model each query as a sequence of resource rectangles aligned along the timeline, where each rectangle represents a pipeline stage. The rectangle's width corresponds to the stages's predicted duration, and its height represents the resource demand. As shown in Figure 8, the resource rectangle model provides a unified representation of query resource usage across different compute services, while respecting their specific resource capacity.

The meaning of the resource dimension is adapted to the capacity of each compute service. On VMs, the height of a rectangle

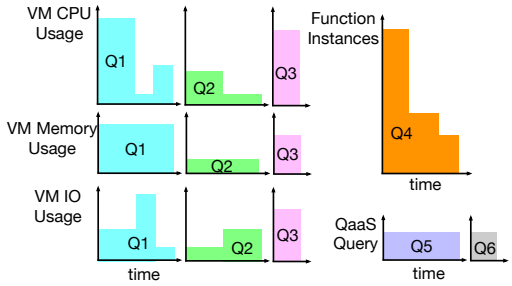


Figure 8: The resource rectangle model for heterogeneous compute services.

represents the demand of CPU, memory, or I/O throughput during the stage. On CFs, the height indicates the number of concurrent function instances required for a stage. On QaaS, each query is represented as a fixed-height rectangle occupying one unit of concurrency throughout its execution, as QaaS typically limits the number of concurrent queries. Despite differing underlying semantics, the unified rectangular abstraction allows the intra-service scheduler to apply a consistent policy across all heterogeneous compute services.

5.2 Problem Formulation

The intra-service scheduler aims to determine optimal start times for a set of pending queries to minimize total response time while respecting the service's resource capacity. Let $Q^{(s)} = \{q_1, \ldots, q_n\}$ be the set of queries waiting on s. Each query $q_i \in Q^{(s)}$ arrives at time a_i and consists of n_i pipeline stages $p_{i1}, p_{i2}, \ldots, p_{in_i}$. Each stage p_{ij} has a predicted execution duration $\hat{d}_{ij}^{(s)}$ and a resource demand vector $\hat{\mathbf{r}}_{ij}^{(s)} \in \mathbb{R}^k$, where k is the number of resource dimensions of the service (e.g., CPU, memory, and I/O bandwidth for VM, function concurrencies for CFs and query concurrencies for QaaS).

Each service s has a fixed resource capacity vector $\mathbf{c}_{\text{capacity}}^{(s)} \in \mathbb{R}^k$, which represents maximum available amount of each resource. The scheduler must assign a start time $t_i \geq a_i$ for each query q_i, such that the total resource consumption of all active queries (denoted $\mathcal{A}(t)$) never exceeds the service's capacity at any time t:

$$\forall t, \quad \sum_{q_i \in \mathcal{A}(t)} \sum_{p_{ij} \text{ active at } t} \hat{\mathbf{r}}_{ij}^{(s)} \leq \mathbf{c}_{\text{capacity}}^{(s)} \tag{10}$$

The objective is to minimize the total response time, defined as:

$$\min \sum_{j=1}^{n} \left(C_j - a_j \right) \tag{11}$$

where C_j and a_j denote the completion time and arrival time of query q_j respectively.

5.3 Complexity Analysis

We prove that the intra-service scheduling problem is NP-hard by reduction from the classical single-machine scheduling problem with release time r_j and the objective of minimizing total completion time $\sum C_j$, denoted as $1|r_j|\sum C_j$ [23]. This problem is known to be NP-hard even when each job consists of a single stage and only one resource dimension is considered.

Consider an instance of $1|r_j|\sum C_j$ with n jobs, each having a release time r_j and a processing time π_j. We construct an equivalent instance of the intra-service scheduling problem as follows:

- Each job j corresponds to a query q_j with a single stage p_{j1}.
- The job's release time r_j corresponds to the arrival time a_j of query q_j.
- The job's processing time π_j corresponds to the execution duration $d_{j1}^{(s)}$ of stage p_{j1}.
- The resource demand vector for stage p_{j1} is $\mathbf{r_{j1}}^{(s)} = (1, 0, \ldots, 0)$, indicating that each query consumes one unit of the first resource dimension and zero in others.
- The service's resource capacity vector is $\mathbf{c}_{\text{capacity}}^{(s)} = (1, 0, \ldots, 0)$, meaning that it can process at most one concurrent query.

Under this construction, any valid schedule for the intra-service scheduling problem corresponds to a valid schedule for the single-machine scheduling problem with release dates. Furthermore, minimizing the total response time $\sum(C_j - a_j)$ in the intra-service scheduling problem is equivalent to minimizing the total completion time $\sum C_j$ in the single-machine scheduling problem, as $\sum a_j$ is a constant offset that does not affect the optimization objective. Therefore, any algorithm that solves the intra-service scheduling problem can also solve the single-machine scheduling problem, proving that the intra-service scheduling problem is NP-hard.

5.4 Resource-Aware Scheduling Algorithm

Given the NP-hardness of the intra-service scheduling problem, we design a lightweight heuristic algorithm for real-time scheduling. It leverages predicted query resource demands and current resource availability to minimize total response time while respecting the service's resource capacity constraints.

We discretize the timeline into fixed-length time slots and model each query's resource usage over a finite horizon of t slots. For each query q on service s, we represent its resource demand as a matrix $\hat{\mathbf{R}}_q^{(s)} \in \mathbb{R}^{k \times t}$, where k is the number of resource dimensions, and each entry $\hat{\mathbf{R}}_q^{(s)}[i, j]$ denotes the predicted demand for resource type i in time slot j. We also maintain a remaining capacity matrix $\hat{\mathbf{C}}_{\text{remain}}^{(s)} \in \mathbb{R}^{k \times t}$, representing the available capacity across all resource dimensions and t time slots. To simplify the representation, we flatten both matrices into vectors $\hat{\mathbf{r}}_q^{(s)} \in \mathbb{R}^{kt}$ and $\hat{\mathbf{c}}_{\text{remain}}^{(s)} \in \mathbb{R}^{kt}$.

The scheduling algorithm is triggered at key decision points, such as the arrival of new queries, completion of existing queries, or after a fixed time interval. In each scheduling round, we first identify the set of feasible queries $\mathcal{F}$ whose resource demands do not exceed the current remaining capacity $\hat{\mathbf{c}}_{\text{remain}}^{(s)}$. For each feasible query $q \in \mathcal{F}$, we compute a compatibility score to measure how well the its resource demand aligns with the available capacity:

$$\text{score}(q) = \frac{\hat{\mathbf{c}}_{\text{remain}}^{(s)} \cdot \hat{\mathbf{r}}_q^{(s)}}{\|\hat{\mathbf{r}}_q^{(s)}\|^2} \tag{12}$$

The scoring function balances two aspects. The numerator favors queries whose resource usage aligns with the available capacity, improving packing efficiency and reducing resource fragmentation. The denominator penalizes queries with large or long-lasting resource demands that may block other queries and increase contention. As a result, the compatibility score prioritizes lightweight and well-aligned queries that can be executed quickly with minimal resource contention.

After scoring, we select the query with the highest score q^* and add it to the scheduled set $\mathcal{S}$. We then update the remaining capacity $\hat{\mathbf{c}}_{\text{remain}}^{(s)}$ by subtracting $\hat{\mathbf{r}}_{q^*}^{(s)}$ and rebuild the feasible query set $\mathcal{F}$ with queries that can still fit within the updated remaining capacity. This process continues until no more queries can be scheduled. Finally, all queries in the scheduled set $\mathcal{S}$ are submitted for execution. The complete scheduling procedure is summarized in Algorithm 1.

Algorithm 1: Resource-Aware Intra-Service Scheduling

Input: Remaining capacity $\hat{\mathbf{c}}_{\text{remain}}^{(s)} \in \mathbb{R}^{kt}$;
Set of waiting queries Q, each with a resource demand vector $\hat{\mathbf{r}}_q^{(s)} \in \mathbb{R}^{kt}$
Output: Set of scheduled queries $\mathcal{S}$

$\mathcal{S} \leftarrow \emptyset$
$\mathcal{F} \leftarrow \{q \in Q \mid \hat{\mathbf{r}}_q^{(s)} \le \hat{\mathbf{c}}_{\text{remain}}^{(s)}\}$
while $\mathcal{F} \neq \emptyset$ **do**
  foreach $q \in \mathcal{F}$ **do**
    $\text{score}(q) \leftarrow \frac{\hat{\mathbf{c}}_{\text{remain}}^{(s)} \cdot \hat{\mathbf{r}}_q^{(s)}}{\|\hat{\mathbf{r}}_q^{(s)}\|^2}$;
  $q^* \leftarrow \text{argmax}_{q \in \mathcal{F}} \text{score}(q)$;
  $\mathcal{S} \leftarrow \mathcal{S} \cup \{q^*\}$;
  $\hat{\mathbf{c}}_{\text{remain}}^{(s)} \leftarrow \hat{\mathbf{c}}_{\text{remain}}^{(s)} - \hat{\mathbf{r}}_{q^*}^{(s)}$;
  $\mathcal{F} \leftarrow \{q \in \mathcal{F} \setminus \{q^*\} \mid \hat{\mathbf{r}}_q^{(s)} \le \hat{\mathbf{c}}_{\text{remain}}^{(s)}\}$;
return $\mathcal{S}$

The intra-service scheduling algorithm relies on predicted resource usage and execution durations, which may be inaccurate due to workload variability. To mitigate the impact of prediction errors, we continuously monitor the actual execution state of running queries, including their current stage and start time. Using this real-time information, we periodically update the remaining capacity matrix $\hat{\mathbf{C}}_{\text{remain}}^{(s)}$ to reconcile predicted and observed resource consumption. This calibration prevents error accumulation and ensures scheduling decisions are based on accurate resource availability.

6 INTER-SERVICE SCHEDULER

The inter-service scheduler complements the intra-service scheduler by dynamically migrating queries across compute services. While the intra-service scheduler prioritizes lightweight and short queries to reduce bottlenecks, this can lead to prolonged delays for long-running or resource-intensive queries under heavy load. The inter-service scheduler addresses this by reducing waiting time through load-aware balancing, while minimizing the performance and cost overhead introduced by query migration. We first describe how to identify queries that are candidates for migration, and then present the load-aware target selection mechanism.

6.1 Migratable Query Identification

To identify queries that are experiencing excessive queueing delays and could benefit from migration, we define a criterion for migration eligibility based on both the query's predicted execution duration and a minimum delay bound:

$$W_q^{(s)} > \max(\gamma \cdot \hat{T}_{q,s}, \tau) \tag{13}$$

where $W_q^{(s)}$ is the current waiting time of query q in the waiting queue of service s. The parameter γ controls how long a query must

wait relative to its expected runtime before becoming eligible for migration, while τ defines a minimum absolute delay.

This design enables adaptive candidate identification. A large γ prevents unnecessary migrations by requiring longer waiting times for long-running queries, while a small γ allows early intervention when such queries are blocked. Meanwhile, τ prevents short queries from being migrated due to transient queueing. These parameters balance the trade-off between waiting time reduction and scheduling stability.

To further enhance stability, we introduce a cooling period after a query is evaluated for migration. During this period, the query is not re-evaluated, preventing excessive migration and redundant computation. Additionally, we track the migration history of each query to avoid ping-pong effects. If a query has been migrated recently, it is temporarily excluded from migration consideration to prevent oscillation between services.

6.2 Load-Aware Target Selection

For each query eligible for migration, the inter-service scheduler evaluates potential target services by estimating the expected completion time if scheduled on each service. The estimate consists of two components: the waiting time in the target service's queue and the execution time on that service. To avoid redundant computation, we reuse the predicted execution time and monetary cost generated from the query router (Section 4.2). This allows us to focus the evaluation on the waiting time component, which depends on the current load of each service. We present our estimation method below.

To estimate waiting time on each service, we model the service's current load across all resource dimensions. Let $\hat{\mathbf{R}}_q^{(s)} \in \mathbb{R}^{k\times t}$ be the predicted resource demand matrix of query q on service s, with k resource dimensions and a time horizon of t slots. The total capacity of the service s is represented by a k-dimensional vector $\mathbf{c}_{\text{capacity}}^{(s)} \in \mathbb{R}^k$, where each entry denotes the maximum available amount of a given resource type. To model temporal resource availability, we construct a broadcast capacity matrix $\mathbf{C}_{\text{capacity}}^{(s)} \in \mathbb{R}^{k\times t}$ by replicating $\mathbf{c}_{\text{capacity}}^{(s)}$ across all t time slots:

$$\mathbf{C}_{\text{capacity}}^{(s)} = \left[\mathbf{c}_{\text{capacity}}^{(s)} \ \cdots \ \mathbf{c}_{\text{capacity}}^{(s)}\right] \in \mathbb{R}^{k\times t} \tag{14}$$

We compute the current resource usage matrix $\hat{\mathbf{C}}_{\text{used}}^{(s)}$ as:

$$\hat{\mathbf{C}}_{\text{used}}^{(s)} = \mathbf{C}_{\text{capacity}}^{(s)} - \hat{\mathbf{C}}_{\text{remain}}^{(s)} \tag{15}$$

We estimate the waiting time $\hat{W}^{(s)}$ on service s by aggregating resource consumption of both pending and active queries. For each resource dimension i, we compute the total load as the sum of the predicted demand of pending queries $Q^{(s)}$ and the current resource usage. We then normalize the load by the service's per-dimension capacity. The estimated waiting time on service s is defined as the maximum normalized load across all resource dimensions:

$$\hat{W}^{(s)} = \max_{i=1,\ldots,k}\left(\frac{\sum_{q\in Q^{(s)}}\sum_{j=1}^{t}\hat{\mathbf{R}}_q^{(s)}[i,j] + \sum_{j=1}^{t}\hat{\mathbf{C}}_{\text{used}}^{(s)}[i,j]}{\mathbf{c}_{\text{capacity}}^{(s)}[i]}\right) \tag{16}$$

This estimate models queueing delay under the assumption of full resource utilization. While this does not capture precise scheduling behavior, it offers a practical and efficient way to assess service pressure. In production environments, OLAP queries are often dominated by short-running with modest resource demands [36], which can be packed tightly into the available capacity. This results in minimal fragmentation and consistently high resource utilization. Under such conditions, the full-utilization assumption offers a reasonable approximation of the actual queueing delays and enables effective inter-service scheduling decisions without the overhead of full simulation.

Once the estimated waiting time $\hat{W}^{(s)}$, execution time $\hat{T}_{q,s}$ and monetary cost $\hat{M}_{q,s}$ of query q are computed for each service s, we select the migration target using the following objective:

$$s^* = \operatorname*{argmin}_s \left(W_q^{(s)} + \hat{W}^{(s)} + \hat{T}_{q,s}\right) \cdot \hat{M}_{q,s}^{\alpha} \tag{17}$$

This formulation enables the inter-service scheduler to identify optimal migration targets based on a combination of estimated waiting time, execution duration, and cost, while respecting user preferences through the tunable parameter α. The query is migrated if the optimal target s^* differs from its current service. This decision process allows the system to react adaptively to workload fluctuations while avoiding unnecessary migrations overhead.

7 EVALUATION

In this section, we evaluate RUSH on a set of OLAP workloads to assess its effectiveness in reducing query latency and cost. Our evaluation includes end-to-end comparisons with baseline strategies, ablation studies to understand the contributions of the intra-service and inter-service schedulers, and additional experiments that examine the system's behavior under different workload intensities, the effect of the cost-performance trade-off parameter α, the accuracy of resource demand prediction, and the runtime overhead introduced by individual modules. These experiments demonstrate the effectiveness of RUSH in scheduling queries in heterogeneous cloud environments, highlighting its ability to adapt to dynamic workloads and optimize resource utilization.

7.1 Experimental Setup

Platform. We implement and evaluate RUSH on AWS using a heterogeneous set of compute services:

- **VM:** EC2 m5.8xlarge instances with 32 vCPUs, 128 GB memory, and 10Gbps network bandwidth, billed at $1.536 per hour.
- **CF:** Lambda with a default concurrency limit of 1000 function invocations, priced at $0.0000166667 per GB-second.
- **QaaS:** Athena with support for up to 30 concurrent queries, charged at $5 per TB of data scanned.

All experiments are conducted in the us-east-1 region, and the pricing reflects the rates as of August 2025. All data is stored in Amazon S3 and can be accessed directly by all compute services without additional replication. On EC2 and Lambda, queries are executed using DuckDB [12] (version 1.1.3). While our implementation is specific to AWS, the design principles are general and portable to other cloud providers. To adapt to dynamic workloads, we enable auto-scaling on EC2. New instances are launched if average CPU utilization exceeds 80% for more than 1 minute, and idle instances are terminated when utilization remains below 20% for 5 minutes. This strategy balances responsiveness to load spikes and cost efficiency, avoiding frequent scale-in/out due to short-lived load fluctuations.

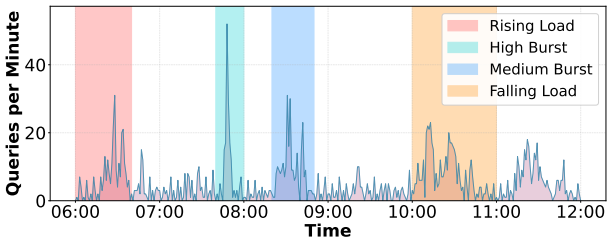


Figure 9: Query arrival rates over a six-hour trace (6:00-12:00, February 28, 2018) from a single database in the Snowset [37]. The shaded regions indicate the four representative workload patterns used in our experiments.

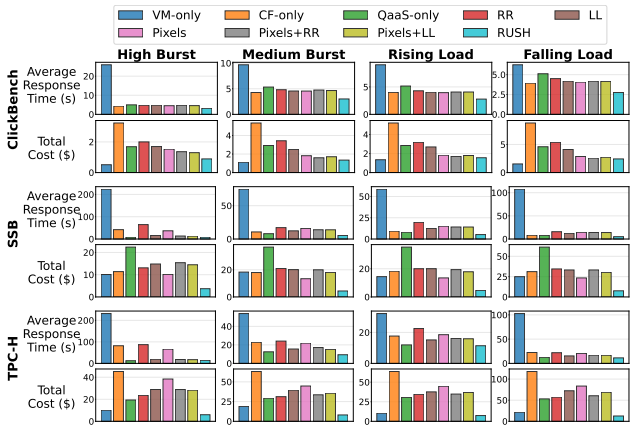


Figure 10: End-to-end experimental results.

Benchmarks and workload patterns. We evaluate RUSH using three widely adopted analytical benchmarks: ClickBench [14], SSB [29], and TPC-H [1]. SSB and TPC-H are configured at a scale factor of 300, and all datasets are stored in Parquet format with Snappy compression, resulting in compressed sizes of approximately 13.7 GB (ClickBench), 61.3 GB (SSB), and 103.8 GB (TPC-H). To simulate realistic workload patterns, we use query traces from Snowset [37], a large-scale query trace collected by Snowflake. As shown in Figure 9, we extract four representative workload patterns from a single database during a six-hour period (6:00-12:00) on February 28, 2018: (1) high burst, characterized by a sudden spike in query arrivals; (2) medium burst, with moderate fluctuations in query intensity; (3) rising load, where query load gradually increases; and (4) falling load, where query load gradually decreases. For each arrival event in the trace, we randomly select a query from the benchmark's query set. These patterns allow us to evaluate RUSH under diverse dynamic workload conditions.

Baselines. We compare RUSH against the following baselines:

- **VM-only, CF-only, and QaaS-only**: all queries are routed to a single type of compute service.
- **Round-Robin (RR)**: queries are distributed across all compute services in cyclic order, without considering query characteristics or system load.
- **Least-Loaded (LL)**: each query is assigned to the service with the fewest unfinished queries at the time of arrival.
- **Pixels** [18]: a heuristic that prioritizes VM execution and falls back to CFs when the number of pending queries on VM exceeds a threshold (set to 3 in our experiments).
- **Pixels+RR**: an extension of Pixels that dispatches overflow queries to CF and QaaS in a round-robin manner when the VM queue is full.
- **Pixels+LL**: another extension of Pixels that selects the less-loaded service between CF and QaaS for overflow queries.

Metrics. We evaluate both performance and cost using the following metrics:

- **Average Response Time:** the average time from query arrival to completion, including both queueing and execution time.
- **Total Monetary Cost:** the total monetary cost incurred during query execution, including compute and storage charges.

7.2 Overall Results

Figure 10 compares RUSH with all baselines across three benchmarks and four workload patterns. In these experiments, we set the cost-performance trade-off parameter α to 1, which provides a balanced weighting between response time and monetary cost. Our key observations are as follows.

First, RUSH achieves the lowest response time across all benchmarks and workload patterns, and also offers strong cost efficiency. On SSB and TPC-H, it delivers the lowest overall cost. Specifically, on SSB, RUSH reduces response time by 35% compared to QaaS-only (the best baseline in response time) and lowers monetary cost by 67% compared to Pixels (the best baseline in cost). On ClickBench, its cost is higher than VM-only, but remains lower than all other baselines. This is because ClickBench queries are simple and lightweight, requiring limited compute resources and allowing VM-only to perform well without frequent auto-scaling. However, for more complex and resource-intensive queries in SSB and TPC-H, VM-only suffers from significant queueing delays and slow auto-scaling response, leading to high latency. In contrast, RUSH distributes queries across heterogeneous compute services based on real-time workload characteristics. By offloading bursty queries to highly elastic services like CF and QaaS, RUSH avoids VM queue buildup and unnecessary scaling, reducing both response time and monetary cost under diverse workloads.

Second, the performance of CF-only and QaaS-only highlights the limitations of relying on a single compute service. CF-only shows highly variable performance across benchmarks, reflecting its sensitivity to query complexity. On ClickBench, where queries are simple and contain only one or two pipeline stages with minimal intermediate results, CF-only achieves the second-best latency, benefiting from the high concurrency and fast startup of CFs. However, on more complex benchmarks like SSB and TPC-H, the overhead of exchanging intermediate results across multiple function instances leads to increased latency and cost. In contrast, QaaS-only delivers more stable latency across workloads and is particularly effective under bursty patterns due to its support for high query concurrency. Nevertheless, it suffers from relatively high response time on short-running queries such as those in ClickBench, where cold start latency constitutes a significant portion of total execution time. For long-running queries in SSB and TPC-H, cold start becomes less impactful, but QaaS incurs high monetary cost on scan-heavy

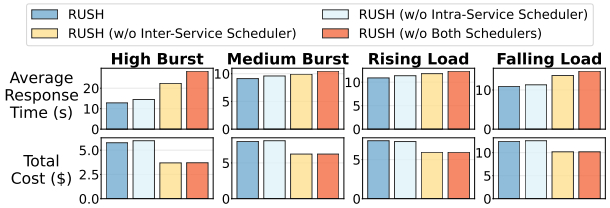


Figure 11: Ablation study on TPC-H under four workloads.

queries, particularly in SSB, due to its per-scanned-data pricing model. These results demonstrate that no single compute service performs best across all scenarios, highlighting the importance of a system that adaptively exploits the strengths of different services based on query and workload characteristics.

Third, multi-service baselines such as RR, LL, Pixels, and their variant show the benefits of leveraging heterogeneous compute services, but still fall short of optimal performance. Among them, Pixels+LL performs the best by prioritizing VM execution with a bounded queue and offloading overflow queries to the less-loaded service between CF and QaaS. This strategy reduces both VM idle time and queueing delays, achieving better performance than other multi-service strategies. However, its decisions are purely based on queue length, without considering the complexity or resource requirements of queries. For instance, it may route complex multi-stage queries to CFs, leading to significant communication overhead, or it may schedule numerous simple queries to occupy the VM resources unnecessarily. In contrast, RUSH leverages fine-grained resource modeling to make informed routing decisions, demonstrating the benefits of resource-aware scheduling in heterogeneous environments.

In summary, the evaluation shows that existing approaches are highly sensitive to workload characteristics and query complexity, and their lack of fine-grained resource awareness limits their efficiency under dynamic conditions. In contrast, RUSH adapts to changing workload patterns and effectively leverages the complementary strengths of heterogeneous compute services, achieving superior performance and cost efficiency across diverse OLAP workloads. These results highlight the effectiveness of RUSH in addressing the challenges of modern cloud-based OLAP systems.

7.3 Ablation Study

To assess the effectiveness of each scheduling component in RUSH, we conduct an ablation study on TPC-H under four workload patterns. We evaluate four variants of RUSH: the full RUSH, RUSH without intra-service scheduling, RUSH without inter-service scheduling, and RUSH with both schedulers disabled. Figure 11 reports the average query response time and total cost for each variant.

Disabling the intra-service scheduler leads to a moderate increase in response time, particularly under the high burst workload. Without intra-service scheduling, queries are executed in a first-come-first-served (FCFS) manner within each service, disregarding their resource demands. This often causes multiple resource-intensive queries being processed concurrently, resulting in severe contention and increased latency. The impact is especially evident during load spikes, where the sudden influx of queries causes the queue to grow rapidly, making the lack of resource-aware scheduling a major bottleneck and leading to substantial performance degradation.

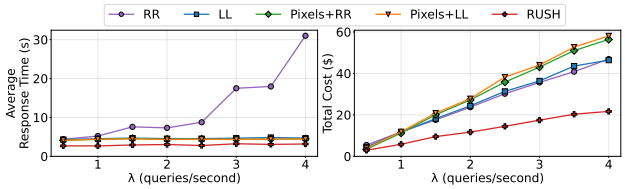


Figure 12: Impact of query arrival rates on ClickBench.

These results highlight the importance of intra-service scheduling in managing resource contention under dynamic workloads.

Removing the inter-service scheduler leads to a more substantial increase in response time, especially under the high burst workload. Without the ability to redistribute queries across services, the system cannot effectively alleviate congestion, resulting in long queues and delayed execution on overloaded services. Interestingly, this variant incurs lower cost, as it avoids assigning queries to less-utilized services which typically have higher costs. However, this comes at the expense of higher latency, since queries with long waiting time are no longer proactively migrated, increasing the risk of starvation and prolonging tail latency.

When both schedulers are disabled, the system loses the ability to coordinate query execution both within and across services, resulting in the worst performance across all workloads. Queries are scheduled in a purely reactive manner, with no mechanism to balance resource usage or respond to load fluctuations. The results confirm that both intra-service and inter-service scheduling are essential components of RUSH, and their combination enables effective resource management under diverse and dynamic workloads.

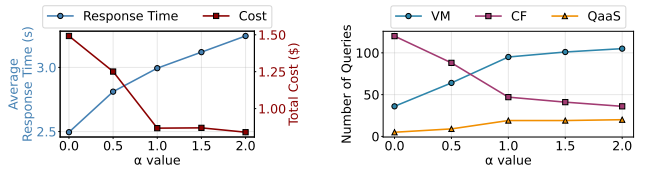
7.4 Additional Experiments

7.4.1 Impact of Query Arrival Rate. To examine how different strategies perform under varying workload intensity, we compare RUSH and four baselines (RR, LL, Pixels+RR, and Pixels+LL) under varying arrival rates using Poisson-distributed query arrivals with rate λ ranging from 0.5 to 4 queries per second. Figure 12 shows the average response time and total cost on ClickBench.

As λ increases, the average response time remains stable for all methods except RR, which exhibits a sharp rise. This is because RR blindly distributes queries without considering service load, causing severe imbalance and queueing delays under high arrival rates. In contrast, LL and Pixels-based methods adapt to load conditions by steering queries toward less utilized services. RUSH further incorporates both load and resource-awareness in scheduling, achieving the lowest response time across all arrival rates. Despite increasing load, VM auto-scaling helps absorb the added pressure due to the stable nature of the Poisson process, enabling all adaptive strategies to maintain stable latency.

Total cost increases gradually with λ for all methods as more queries are executed. RUSH consistently incurs the lowest cost at every arrival rate thanks to its ability to assign queries to cost-efficient services while respecting performance goals. These results demonstrate that RUSH achieves both low latency and low cost across a wide range of arrival rates, highlighting its robustness and adaptability to varying workload intensities.

7.4.2 Impact of parameter α. We evaluate how RUSH responds to different user preferences by varying α (defined in Section 2.2),



(a) Average response time and total cost

(b) Distribution of query executions across services

Figure 13: Impact of parameter α on ClickBench under high burst workload in RUSH.

which controls the trade-off between response time and cost. Experiments are conducted on ClickBench under the high burst workload. As shown in Figure 13a, increasing α leads to higher average response time and lower total cost, demonstrating that RUSH provides a smooth and tunable balance between performance and cost.

Figure 13b further reveals how this trade-off is achieved through adaptive query placement across services. At low α, RUSH prioritizes performance by routing most queries to CFs, which are effective for simple queries in ClickBench. As α increases, more queries are shifted to VMs and QaaS to leverage their cost efficiency for complex or scan-light queries. These results show that RUSH dynamically aligns scheduling decisions with user preferences by exploiting service capabilities and workload characteristics.

7.4.3 Resource Usage Prediction. We assess the accuracy of our pipeline-level resource usage prediction model on three benchmarks, using 70% of the queries for training and the remaining 30% for testing. Both pipeline-level and query-level predictions are implemented using XGBoost with identical feature sets, as described in Section 4.1. The results are shown in Table 4.

Pipeline-level prediction consistently achieves lower prediction errors than query-level prediction across all benchmarks and resource types. The improvement is particularly notable on SSB and TPC-H, which contain more complex queries with multiple pipeline stages. Modeling resource usage at the pipeline level allows the model to capture the relatively stable resource behavior within each pipeline while avoiding the complexity of learning abrupt transitions across pipelines. In contrast, query-level models must directly predict resource usage over time, making it harder to learn accurate temporal patterns, especially for complex queries with highly variable behavior. These results highlight the effectiveness of pipeline-level modeling in improving prediction quality, which enhances scheduling precision and overall system performance.

7.4.4 Overhead Analysis. We analyze the runtime overhead introduced by each component of RUSH under four workload patterns. As shown in Table 5, the total overhead remains below 1% of the average response time, indicating that RUSH incurs negligible runtime overhead compared to the overall query execution time.

The highest overhead is observed under the high burst workload, where a large number of queries arrive in a short time window. This increases scheduling complexity, as the schedulers must evaluate more queries and determine their service assignments under limited resources. Despite this, the total overhead remains negligible, thanks to the use of lightweight models and efficient algorithms. The low runtime cost ensures that RUSH remains responsive even under bursty workloads, enabling real-time scheduling decisions without introducing significant delays.

Table 4: Comparison of pipeline-level and query-level resource usage prediction accuracy.

Benchmark	Metric	Method	CPU (%)	Memory (GB)	I/O (%)
ClickBench	MAE	Pipeline-level	**5.65**	**0.34**	**5.89**
		Query-level	10.31	0.49	13.07
	RMSE	Pipeline-level	**7.50**	**0.54**	**7.38**
		Query-level	12.65	0.70	14.79
SSB	MAE	Pipeline-level	**3.36**	**0.08**	**5.20**
		Query-level	8.43	0.22	21.34
	RMSE	Pipeline-level	**4.77**	**0.17**	**10.55**
		Query-level	11.73	0.35	29.51
TPC-H	MAE	Pipeline-level	**4.62**	**0.22**	**7.38**
		Query-level	12.82	0.78	25.20
	RMSE	Pipeline-level	**6.91**	**0.41**	**12.22**
		Query-level	16.19	0.98	30.68
Average	MAE	Pipeline-level	**4.98**	**0.27**	**6.19**
		Query-level	10.71	0.53	17.87
	RMSE	Pipeline-level	**6.88**	**0.44**	**9.27**
		Query-level	13.49	0.72	21.72

8 RELATED WORK

Analytical query execution in the cloud. Cloud-based OLAP query execution has attracted substantial attention due to its scalability and flexibility. Early systems like Starling [31] and Lambada [27] leverage cloud functions to enable elastic and pay-as-you-go query execution. However, these systems are best suited for short, simple queries and suffer from high latency when handling complex queries. More recent solutions like Pixels [18] and Cackle [30] combine virtual machines and cloud functions to accelerate OLAP query execution, but they use rule-based strategies to decide between services, which can lead to suboptimal performance under bursty workloads. While these systems demonstrate the potential of combining multiple services for query execution, they lack a general scheduling framework that accounts for service heterogeneity, resource dynamics, and query diversity. Our work addresses these limitations by introducing a resource-aware and adaptive scheduling framework that explicitly models the capabilities of heterogeneous cloud services. This allows our system to dynamically schedule queries to the most appropriate service, improving both performance and cost efficiency across a broad range of OLAP workloads.

Resource modeling and prediction. Prior work has explored various methods for predicting query performance using analytical models [15, 16, 20, 39] or learning-based estimators [17, 21, 26, 32, 33, 40, 43]. These methods are typically designed for traditional database systems with homogeneous execution environments and do not account for the diversity of cloud services. Moreover, they often focus on predicting overall latency, providing limited insight into the varying resource demands of different query stages. In contrast, our work predicts resource usage at the pipeline level, capturing how resource demands evolve during execution. This enables

Table 5: Breakdown of per-query overhead introduced by different components in RUSH under four workloads.

Component	High Burst	Medium Burst	Rising Load	Falling Load
Plan generation	0.0274s	0.0258s	0.0262s	0.0248s
Plan decomposition	0.0039s	0.0038s	0.0039s	0.0040s
Query router	0.0244s	0.0234s	0.0229s	0.0229s
Intra-service scheduler	0.0681s	0.0360s	0.0453s	0.0464s
Inter-service scheduler	0.0017s	0.0011s	0.0005s	0.0006s
Total overhead	**0.1255s**	**0.0901s**	**0.0988s**	**0.0987s**
Average response time	14.240s	9.356s	11.410s	11.367s

accurate cost and performance estimation across heterogeneous services, allowing for more effective scheduling decisions.

Workload management and scheduling. Workload management has been studied with various objectives, such as buffer-awareness [22], throughput under high concurrency [33, 41], cost efficiency [25, 34, 42], and low-level resource sharing [38]. However, these systems typically operate within a single execution backend, where resource constraints and execution environments are uniform. Our work extends the scheduling problem to the heterogeneous cloud environment, aiming to match queries to services that best fit their resource demands and the current system status. This enables better performance and lower cost compared to traditional single-service or rule-based scheduling strategies.

9 CONCLUSION

We present RUSH, a resource-aware scheduling framework for executing OLAP queries across heterogeneous compute services. By predicting per-stage resource demands and dynamically scheduling queries within and across services, RUSH achieves both high performance and cost efficiency. Extensive experiments demonstrate that RUSH outperforms existing approaches under diverse workloads, highlighting the effectiveness of fine-grained resource modeling and adaptive scheduling in heterogeneous cloud environments.

REFERENCES

[1] 1999. *TPC-H.* https://www.tpc.org/tpch/
[2] 2004. *Amazon S3.* https://aws.amazon.com/s3/
[3] 2006. *Amazon EC2.* https://aws.amazon.com/ec2/
[4] 2011. *BigQuery.* https://cloud.google.com/bigquery
[5] 2012. *Azure Virtual Machines.* https://azure.microsoft.com/products/virtual-machines
[6] 2012. *Compute Engine.* https://cloud.google.com/products/compute
[7] 2013. *Presto.* https://prestodb.io
[8] 2014. *AWS Lambda.* https://aws.amazon.com/lambda/
[9] 2016. *Amazon Athena.* https://aws.amazon.com/athena/
[10] 2016. *Azure Functions.* https://azure.microsoft.com/products/functions
[11] 2017. *Cloud Run functions.* https://cloud.google.com/functions
[12] 2019. *DuckDB.* https://duckdb.org
[13] 2020. *Azure Synapse Analytics.* https://azure.microsoft.com/en-us/products/synapse-analytics
[14] 2022. *ClickBench.* https://github.com/ClickHouse/ClickBench
[15] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. 2011. Interaction-aware scheduling of report-generation workloads. *VLDB J.* 20, 4 (2011), 589–615. doi:10.1007/S00778-011-0217-Y
[16] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulnaga, and Shivnath Babu. 2011. Predicting completion times of batch query workloads using interaction-aware models and simulation. 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, 449–460. doi:10.1145/1951365.1951419
[17] 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. doi:10.1109/ICDE.2012.64
[18] Haoqiong Bian, Tiannan Sha, and Anastasia Ailamaki. 2023. Using Cloud Functions as Accelerator for Elastic Data Analytics. *Proc. ACM Manag. Data* 1, 2 (2023), 161:1–161:27. doi:10.1145/3589306
[19] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Francisco, CA, USA, August 13-17, 2016*, Balaji Krishnapuram, Mohak Shah, Alexander J. Smola, Charu C. Aggarwal, Dou Shen, and Rajeev Rastogi (Eds.). ACM, 785–794. doi:10.1145/2939672.2939785
[20] Jennie Duggan, Ugur Çetintemel, Olga Papaemmanouil, and Eli Upfal. 2011. Performance prediction for concurrent database workloads. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2011, Athens, Greece, June 12-16, 2011*, Timos K. Sellis, Renée J. Miller, Anastasios Kementsietsidis, and Yannis Velegrakis (Eds.). ACM, 337–348. doi:10.1145/1989323.1989359
[21] 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. doi:10.14778/3551793.3551799
[22] Yuwei Huang and Guoliang Li. 2024. Laser: Buffer-Aware Learned Query Scheduling in Master-Standby Databases. *Proc. VLDB Endow.* 18, 3 (2024), 743–755. https://www.vldb.org/pvldb/vol18/p743-li.pdf
[23] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. 1977. Complexity of Machine Scheduling Problems. In *Studies in Integer Programming*, P.L. Hammer, E.L. Johnson, B.H. Korte, and G.L. Nemhauser (Eds.). Annals of Discrete Mathematics, Vol. 1. Elsevier, 343–362. doi:10.1016/S0167-5060(08)70743-X
[24] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub
[25] Ryan Marcus and Olga Papaemmanouil. 2016. WiSeDB: A Learning-based Workload Management Advisor for Cloud Databases. *Proc. VLDB Endow.* 9, 10 (2016), 780–791. doi:10.14778/2977797.2977804
[26] Ryan Marcus and Olga Papaemmanouil. 2019. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *Proc. VLDB Endow.* 12, 11 (2019), 1733–1746. doi:10.14778/3342263.3342646
[27] Ingo Müller, Renato Marroquín, and Gustavo Alonso. 2020. Lambada: Interactive Data Analytics on Cold Data Using Serverless Cloud Infrastructure. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 115–130. doi:10.1145/3318464.3389758
[28] Vikram Nathan, Vikramank Y. 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, SIGMOD/PODS 2024, Santiago, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 269–279. doi:10.1145/3626246.3653394
[29] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *Performance Evaluation and Benchmarking, First TPC Technology Conference, TPCTC 2009, Lyon, France, August 24-28, 2009, Revised Selected Papers (Lecture Notes in Computer Science, Vol. 5895)*, Raghunath Othayoth Nambiar and Meikel Poess (Eds.). Springer, 237–252. doi:10.1007/978-3-642-10424-4_17
[30] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, Michael J. Cafarella, and Samuel Madden. 2023. Cackle: Analytical Workload Cost and Performance Stability With Elastic Pools. *Proc. ACM Manag. Data* 1, 4 (2023), 233:1–233:25. doi:10.1145/3626720
[31] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 131–141. doi:10.1145/3318464.3380609
[32] Maximilian Rieger and Thomas Neumann. 2025. T3: Accurate and Fast Performance Prediction for Relational Database Systems With Compiled Decision Trees. *Proc. ACM Manag. Data* 3, 3 (2025), 227:1–227:27. doi:10.1145/3725364
[33] 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*, Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia (Eds.). ACM, 225–237. doi:10.1145/3555041.3589677
[34] Tapan Srivastava and Raul Castro Fernandez. 2024. Saving Money for Analytical Workloads in the Cloud. *Proc. VLDB Endow.* 17, 11 (2024), 3524–3537. doi:10.14778/3681954.3682018
[35] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.* 13, 3 (2019), 307–319. doi:10.14778/3368289.3368296
[36] Alexander van Renen, Dominik Horn, Pascal Pfeil, Kapil 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. *Proc. VLDB Endow.* 17, 11 (2024), 3694–3706. doi:10.14778/3681954.3682031
[37] Midhul Vuppalapati, Justin Miron, Rachit Agarwal, Dan Truong, Ashish Motivala, and Thierry Cruanes. 2020. Building An Elastic Query Engine on Disaggregated Storage. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020*, Ranjita Bhagwan and George Porter (Eds.). USENIX Association, 449–462. https://www.usenix.org/conference/nsdi20/presentation/vuppalapati
[38] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1879–1891. doi:10.1145/3448016.3457260
[39] Wentao Wu, Yun Chi, Hakan Hacigümüs, and Jeffrey F. Naughton. 2013. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *Proc. VLDB Endow.* 6, 10 (2013), 925–936. doi:10.14778/2536206.2536219
[40] 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 *Companion of the 2024 International Conference on Management of Data, SIGMOD/PODS 2024, Santiago AA, Chile, June 9-15, 2024*, Pablo Barceló, Nayat Sánchez-Pi, Alexandra Meliou, and S. Sudarshan (Eds.). ACM, 280–294. doi:10.1145/3626246.3653391
[41] 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. *CoRR* abs/2501.16256 (2025). doi:10.48550/ARXIV.2501.16256 arXiv:2501.16256
[42] Geoffrey X. Yu, Ziniu Wu, Ferdinand Kossmann, Tianyu Li, Markos Markakis, Amadou Latyr Ngom, Samuel Madden, and Tim Kraska. 2024. Blueprinting the Cloud: Unifying and Automatically Optimizing Cloud Data Infrastructures with BRAD. *Proc. VLDB Endow.* 17, 11 (2024), 3629–3643. doi:10.14778/3681954.3682026
[43] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (2020), 1416–1428. doi:10.14778/3397230.3397238