

EASE: Resource-aware Query Scheduling across Heterogeneous Cloud Compute Services

WENBO LI, Tsinghua University, China

HAOQIONG BIAN*, Renmin University of China, China

CHAO ZHANG, Renmin University of China, China

GUOLIANG LI*, Tsinghua University, China

Modern cloud platforms provide diverse compute services, including virtual machines, Function-as-a-Service, and Query-as-a-Service, each offering unique trade-offs in performance, elasticity, and cost. While these services collectively cover the diverse needs of OLAP workloads, existing systems typically rely on a single service type or static rule-based scheduling. However, no single service is optimal for dynamic OLAP workloads with diverse performance characteristics and resource requirements. Consequently, efficient query execution requires scheduling workloads across heterogeneous services, which introduces three challenges. (1) Heterogeneous service models. Distinct billing models and execution paradigms (e.g., stateful or stateless) complicate the comparison of services. (2) Scheduling complexity at scale. High query concurrency exacerbates resource contention, creating a vast search space that makes identifying optimal scheduling decisions in real time computationally expensive. (3) Unpredictable runtime dynamics. Cloud infrastructure exhibits performance fluctuations due to factors like multi-tenant interference and scaling delays, which can invalidate initial scheduling decisions and cause suboptimal performance.

In this paper, we present EASE, a resource-aware query scheduling framework designed to orchestrate OLAP queries across heterogeneous cloud compute services. EASE addresses these challenges through three integrated mechanisms. First, EASE incorporates a heterogeneity-aware cost model that captures distinct service characteristics to guide service selection. Second, EASE employs a real-time query scheduler that resolves resource contention under high concurrency. Third, EASE implements a dynamic rescheduler that mitigates runtime fluctuations by proactively migrating queries subject to unexpected delays. Our evaluation demonstrates that EASE significantly improves query performance and cost efficiency, reducing query latency by up to 35% and monetary cost by up to 67% compared to state-of-the-art approaches.

CCS Concepts: • **Information systems** → **Data warehouses; Online analytical processing engines; DBMS engine architectures**; • **Computer systems organization** → **Cloud computing**.

Additional Key Words and Phrases: OLAP, FaaS, QaaS, cloud databases, cloud functions, query scheduling

ACM Reference Format:

Wenbo Li, Haoqiong Bian, Chao Zhang, and Guoliang Li. 2026. EASE: Resource-aware Query Scheduling across Heterogeneous Cloud Compute Services. *Proc. ACM Manag. Data* 4, 4, Article 270 (September 2026), 27 pages. <https://doi.org/10.1145/3837108>

*Corresponding authors.

Authors' Contact Information: Wenbo Li, Tsinghua University, Beijing, China, lwb21@mails.tsinghua.edu.cn; Haoqiong Bian, Renmin University of China, Beijing, China, bianhq@ruc.edu.cn; Chao Zhang, Renmin University of China, Beijing, China, cycchao@ruc.edu.cn; Guoliang Li, Tsinghua University, Beijing, China, liguoliang@tsinghua.edu.cn.



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

© 2026 Copyright held by the owner/author(s).

ACM 2836-6573/2026/9-ART270

<https://doi.org/10.1145/3837108>

1 Introduction

Modern cloud platforms provide a wide range of compute services, each optimized for specific types of workloads. For instance, virtual machines (VMs) [3, 5, 6] offer stable performance and are well-suited for long-running queries with steady resource usage. Function-as-a-Service (FaaS) [9, 11, 12] provides high elasticity and fine-grained billing, making it suitable for embarrassingly parallel queries and short-lived bursty workloads. Query-as-a-Service (QaaS) [4, 10, 14] removes the burden of infrastructure provisioning, enabling on-demand execution of ad-hoc analytical queries. As summarized in Table 1, these services differ in startup time, billing granularity, and concurrency limits, making them suitable for different use cases. This disparity is illustrated in Figure 1a, which shows that the same query can exhibit vastly different execution times across different 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 cause query backlogs during sudden workload spikes. FaaS lacks native support for inter-function communication, making it inefficient for complex queries involving large-scale data exchanges. QaaS platforms typically adopt a pay-per-data-scanned pricing model, incurring high costs for queries that scan large volumes of data.

In practice, cloud users often rely on multiple cloud compute services to meet diverse workload demands [25, 39, 56]. Real-world e-commerce and booking workloads often experience traffic spikes, which require elastic capacity beyond provisioned resources [17, 18]. Industrial systems like Amazon Redshift also integrate serverless capabilities to supplement provisioned clusters [23].

Nevertheless, deciding how to route queries across heterogeneous services is difficult for users because each service exhibits different performance, elasticity, and pricing characteristics. Existing scheduling approaches primarily operate within a single type of execution environment and do not exploit the variety of modern cloud platforms. Most prior schedulers focus on homogeneous and static clusters [29, 53]. Their primary objective is to optimize resource utilization within a fixed resource pool, leaving the potential of heterogeneous cloud services unexplored. As shown in Figure 1b, restricting query execution to a single service type results in suboptimal performance compared to EASE, which leverages heterogeneous services to execute the queries.

This gap highlights the need for a query scheduling framework that effectively exploits the complementary strengths of heterogeneous cloud compute services while addressing their inherent limitations. However, we face three key challenges.

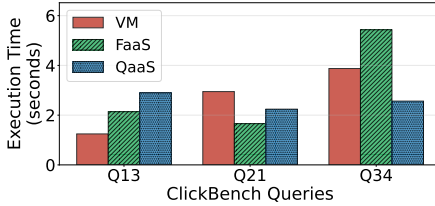
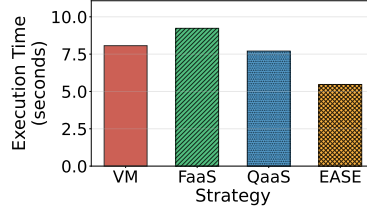
C1. How to compare heterogeneous services with distinct execution models? Cloud compute services exhibit differences across multiple dimensions, including execution paradigms, resource provisioning, and pricing models. Meanwhile, OLAP queries exhibit complex and varying resource requirements, and no single service is universally optimal for the diverse range of OLAP queries. The lack of a unified metric complicates the comparison of heterogeneous services, making it challenging to determine the optimal service for a specific query.

C2. How to efficiently schedule highly concurrent workloads? Bursty OLAP workloads can generate hundreds of concurrent queries, triggering intense competition for shared resources. To mitigate contention, the scheduler must determine both the optimal service placement and execution timing for each query. This creates a vast search space, making it computationally expensive to identify a high-quality schedule within the strict time constraints of online scheduling.

C3. How to adapt to unpredictable runtime dynamics? Cloud environments are inherently volatile, where opaque factors like multi-tenant interference cause significant performance fluctuations. This means that the performance estimated during scheduling often differs from actual runtime behavior. Consequently, a scheduling decision that is optimal at submission time may become suboptimal as the runtime condition changes.

Table 1. Comparison of Cloud Compute Services

Dimension	Metric	VM [3, 5, 6]	FaaS [9, 11, 12]	QaaS [4, 10, 14]
Elasticity	Startup time	Long (tens of seconds)	Short (several milliseconds)	Medium (sub-second to seconds)
Cost	Billing model	Uptime	Execution time	Scanned data volume
Capacity	Resource constraints	Limited hardware resources	Limited function concurrency	Limited query slots
Performance	Best use case	Long-running queries	Bursty short queries	Ad-hoc analytical queries



(a) Query execution time of individual queries. (b) Total execution time of Q13, Q21, and Q34.

Fig. 1. Execution time of ClickBench queries across heterogeneous cloud compute services.

To tackle these challenges, we propose EASE, a rEsoure-Aware query Scheduling framework for hEterogeneous cloud compute services. To address **C1**, we develop a heterogeneity-aware cost model that explicitly captures service-specific characteristics, enabling cross-service comparisons to guide service selection (Section 4). To address **C2**, we design a real-time query scheduler that employs a lightweight resource-aware heuristic to efficiently resolve resource contention, enabling real-time scheduling for highly concurrent workloads (Section 5). To address **C3**, we implement a dynamic rescheduler that mitigates runtime fluctuations by proactively migrating queries when performance volatility invalidates initial scheduling decisions (Section 6).

In summary, this paper makes the following contributions.

- We present EASE, a resource-aware query scheduling framework that orchestrates queries across heterogeneous cloud compute services to handle diverse OLAP workloads efficiently.
- We propose a heterogeneity-aware cost model that explicitly captures distinct characteristics of diverse services, enabling comparison across heterogeneous services.
- We design a real-time scheduler that handles concurrent queries with negligible overhead.
- We introduce a dynamic rescheduler that adapts to performance fluctuations by proactively reassigning queries when initial scheduling decisions become suboptimal.
- We implement and evaluate EASE on real cloud platforms using diverse OLAP workloads, demonstrating that it 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 used for OLAP workloads, highlighting their differences in performance, cost, and elasticity. Then we define the problem of resource-aware query scheduling across heterogeneous services.

2.1 Characteristics of Cloud Compute Services

Modern cloud platforms provide a diverse set of compute services, each presenting unique trade-offs in elasticity, cost, capacity, and performance. To understand the design space for query scheduling across heterogeneous services, we characterize the distinct execution models and resource constraints of the representative services available in modern cloud environments.

Virtual machines (VMs) provide dedicated compute resources with stable performance and customizable hardware configurations, making them well-suited for long-running OLAP queries

with steady resource demands. However, VMs exhibit limited elasticity. Despite support for auto-scaling, provisioning new instances takes tens of seconds to several minutes, rendering them poorly suited for sudden workload spikes. Consequently, the slow scaling forces bursty queries to compete for limited resources, leading to severe contention and performance degradation.

Function-as-a-Service (FaaS) provides a serverless execution model with fine-grained billing and rapid elasticity, making it highly suitable for short-lived and bursty workloads. It can scale out instantly in response to sudden query bursts by rapidly provisioning hundreds of instances. However, FaaS faces significant limitations for complex OLAP queries. Individual function instances are constrained in resources (e.g., up to 6 vCPUs and 10 GB memory on AWS Lambda [9]) and execution duration (e.g., 15 minutes for AWS Lambda), requiring complex queries to launch hundreds of concurrent instances to achieve acceptable performance. Moreover, the total number of concurrent instances is restricted (e.g., 1000 by default on AWS), limiting their scalability under heavy loads. Critically, FaaS lacks direct inter-function communication, forcing reliance on external storage systems like S3 [2] for intermediate data exchange, which incurs substantial overhead.

Query-as-a-Service (QaaS) allows users to execute complex OLAP queries without managing underlying infrastructure by automatically handling resource provisioning. In contrast to FaaS, which relies on external storage for intermediate data exchange during distributed operations like joins and shuffles, QaaS natively supports distributed query execution, 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 result in prohibitive costs for queries that scan large volumes of data. Furthermore, the internal execution behavior remains a black box to users, precluding fine-grained performance tuning.

We select VMs, FaaS, and QaaS to cover the broad spectrum of cloud resource abstractions, spanning from Infrastructure-as-a-Service (IaaS) to fully managed serverless offerings. We differentiate these services based on their distinct resource granularities and pricing models (including time-based and scan-based). Collectively, these services effectively represent the diverse compute options available for OLAP workloads in modern cloud environments. VMs provide dedicated control for steady workloads, FaaS offers fine-grained elasticity for bursty workloads, and QaaS enables massive scalability for complex queries.

Alternatives. Within this spectrum, we employ VMs as the representative for provisioned, long-running compute services. This category conceptually includes **Database-as-a-Service (DBaaS)** (e.g., provisioned Amazon Redshift [7]), which shares a similar resource model and time-based billing. Regarding **serverless DBaaS** (e.g., Redshift Serverless [15]), we consider it a hybrid model that combines the opaque auto-scaling of QaaS with the resource-based billing of VMs. However, a critical distinction is that serverless DBaaS delegates resource provisioning entirely to the cloud provider while charging based on the consumed compute units (e.g., RPU). This implies that costs are determined by the vendor's internal resource allocation policies rather than controllable user decisions, introducing significant uncertainty in cost estimation. Therefore, we use VMs, FaaS, and QaaS as the representatives of three fundamental compute models, while evaluating serverless DBaaS against our approach in Section 7.

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 determines which service should execute the query and when to start execution. The goal is to jointly minimize query latency and monetary cost, achieving an optimal cost-performance trade-off under dynamic workloads. Table 2 summarizes the key notations used in this paper.

Table 2. Notations

Notation	Description	Notation	Description
Q	List of queries	C_i	Completion time of q_i
q	A query in Q	α	Cost-performance trade-off parameter
q_i	i -th query in Q	g_{ij}	j -th concurrent group of q_i
S	Set of compute services	$Q^{(s)}$	Set of queued queries on s
s	A service in S	$\mathcal{A}(t)$	Set of active queries at time t
s_i	The service q_i executes on	$\hat{\mathbf{r}}_{ij}$	Estimated resource demand vector of g_{ij}
s_V	VM service	$\mathbf{c}_{\text{capacity}}^{(s)}$	Resource capacity vector of s
s_F	FaaS service	$\hat{\mathbf{R}}_q$	Estimated 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	$\hat{\mathbf{C}}_{\text{remain}}^{(s)}$	Estimated remaining resource capacity matrix of s
$M_{q,s}$	Monetary cost of q on s	$\hat{\mathbf{c}}_{\text{remain}}^{(s)}$	Flattened vector of $\hat{\mathbf{C}}_{\text{remain}}^{(s)}$
$\hat{T}_{q,s}$	Estimated execution time of q on s	$W_q^{(s)}$	Waiting time of q in the queue of s
$\hat{M}_{q,s}$	Estimated monetary cost of q on s	$\mathbf{C}_{\text{capacity}}^{(s)}$	Resource capacity of s over time slots
A_i	Arrival time of q_i	$\hat{\mathbf{C}}_{\text{used}}^{(s)}$	Estimated used resources of s over time slots
t_i	Execution start time of q_i	$\hat{W}^{(s)}$	Estimated waiting time of s

Let $Q = \{q_1, q_2, \dots, q_n\}$ be the sequence of OLAP queries to be scheduled, where each query q_i arrives at time A_i . The system schedules each query to one of the heterogeneous compute services $S = \{s_V, s_F, s_Q\}$, representing VMs, FaaS, and QaaS, respectively. Each service operates under distinct capacity constraints. VMs are bounded by hardware resources (CPU, memory, and I/O), FaaS by function concurrency limits, and QaaS by query slot quotas.

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 execution time and monetary cost of executing q_i on s_i , respectively. For each incoming query q_i , we make two decisions: the target compute service $s_i \in S$ and the execution start time $t_i \geq A_i$. The query completes at time $C_i = t_i + T_{q_i, s_i}$, resulting in a total latency of $C_i - A_i$. The objective is to jointly optimize query latency and monetary cost by minimizing the performance-cost objective, defined as:

$$\sum_{i=1}^n (C_i - A_i) \cdot M_{q_i, s_i}^{\alpha} \quad (1)$$

where $\alpha \geq 0$ is a preference parameter that controls the relative importance of cost versus performance. This formulation follows the approach adopted in industrial cost-aware scheduling systems such as RAIS [41] in Redshift, where α is specified by users to reflect their performance-cost preferences. A higher value of α emphasizes cost savings, while a lower value favors reducing query latency. In our evaluation, we set $\alpha = 1$ as the default configuration to achieve a balanced trade-off between performance and cost.

3 System Overview

EASE is a resource-aware query scheduling framework designed to orchestrate OLAP queries across heterogeneous cloud compute services. As Figure 2 illustrates, EASE acts as a proxy layer between clients and a diverse pool of cloud services. Users submit SQL queries to EASE without manually selecting target services or managing resources for each service. The system operates on a shared-storage architecture where compute services access data from shared cloud storage.

The internal architecture of EASE consists of three components: the heterogeneity-aware cost model, the real-time query scheduler, and the dynamic rescheduler. These components work in tandem to optimize query latency and execution cost under dynamic conditions. We provide an overview of each component below.

Heterogeneity-aware Cost Model. This component serves as the foundation for scheduling decisions. It employs a hybrid strategy that combines analytical models for services with exposed resource metrics and data-driven learning for opaque managed services. By transforming disparate service-specific characteristics into standardized metrics of execution time and cost, the model establishes a unified ground for cross-service evaluation. The details are provided in Section 4.

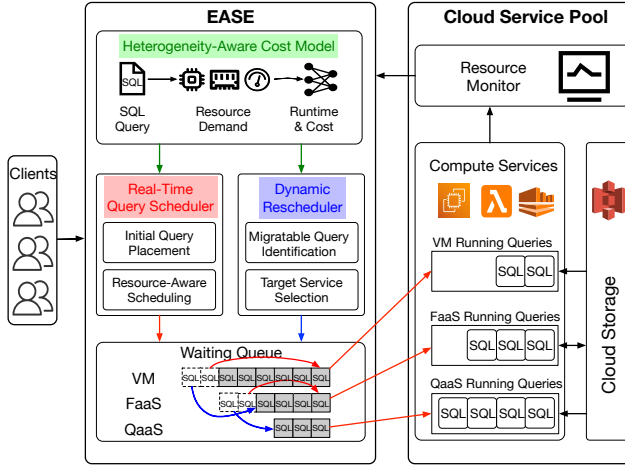


Fig. 2. System overview of EASE.

Real-time Query Scheduler. The scheduler determines the initial placement and execution order of incoming queries. It uses the cost model to evaluate the cost-performance trade-offs of different services, identifying the optimal service and execution start time for each query. In our implementation, queries are dispatched to one of three execution backends. The VM backend runs each query on a single instance from the provisioned pool, the FaaS backend distributes each query across multiple function instances, and the QaaS backend submits each query to the service managed by cloud providers. To handle highly concurrent workloads, the scheduler employs a resource-aware heuristic that mitigates resource contention by co-scheduling queries with complementary resource demands. The query scheduler is described in Section 5.

Dynamic Rescheduler. The dynamic rescheduler handles runtime fluctuations in the cloud. It continuously monitors the system for abnormal delays, acting as a corrective mechanism when queries experience excessive stagnation. When such delays are detected, the rescheduler evaluates the potential benefits of migrating queries to alternative services, proactively adjusting placements to improve overall efficiency. The rescheduler strategy is presented in Section 6.

4 Heterogeneity-aware Cost Model

Effective scheduling across heterogeneous cloud services requires a unified framework to compare distinct execution paradigms and pricing models. In this section, we present our heterogeneity-aware cost model that transforms diverse service characteristics into comparable estimates of execution time and monetary cost. Table 3 summarizes the required parameters and their acquisition methods. We first describe our modeling approach in Section 4.1, followed by query resource profiling in Section 4.2.

4.1 Service Execution Modeling

We model the execution of a query as a directed acyclic graph (DAG) of pipelines, consistent with the execution model used in modern OLAP systems like DuckDB [13] and Presto [8]. A pipeline starts only after all its upstream pipelines complete, and independent pipelines can execute in parallel. To capture this, we represent a query q as a sequence of concurrent groups, denoted as $g_1, g_2, \dots, g_k$, where each group g_i consists of one or more pipelines. Pipelines in each group execute concurrently, while the groups execute sequentially due to data dependencies. For each concurrent group, we characterize its resource demands, including CPU usage, memory consumption, network

Table 3. Parameters of the cost model.

Category	Parameters (Symbols)	Acquisition Methods
Query profile	Data processed by CPU ($\hat{D}_{\text{cpu}}^{(g)}$); data processed by I/O ($\hat{D}_{\text{io}}^{(g)}$); shuffled write and read data sizes ($\hat{D}_{\text{write}}^{(g)}$, $\hat{D}_{\text{read}}^{(g)}$); dominant resource footprint ratio (ρ_q).	Database optimizer estimation, table statistics, profiling, or learned models.
Service profile	VM compute throughput (B_{cpu}), network bandwidth (B_{net}), local SSD bandwidth (B_{ssd}), and memory bandwidth (B_{mem}); FaaS CPU and I/O throughput ($B_{\text{cpu}}(m)$, $B_{\text{io}}(m)$).	Cloud provider specifications and microbenchmarks.
VM data access	Memory and local SSD hit ratios ($\hat{P}_{\text{mem}}^{(q)}$, $\hat{P}_{\text{ssd}}^{(q)}$); remote storage GET requests ($\hat{N}_{\text{get}}^{(q,sv)}$).	Query access ranges, block metadata, and cache status.
FaaS execution	Number of function instances (N_g); function memory size (m_i); function runtime ($\hat{d}_i$); remote storage GET/PUT requests ($\hat{N}_{\text{get}}^{(q,sf)}$, $\hat{N}_{\text{put}}^{(q,sf)}$).	Input data size, partition size, memory-performance profile, and shuffled data size.
Pricing parameters	Cloud storage request prices (R_{get} , R_{put}), VM instance price (R_{vm}), FaaS execution price (R_{faas}), and QaaS scan price (R_{scan}).	Cloud provider pricing specifications.

and disk I/O data volume. Our model maps the resource demands to expected execution time and cost for each service, accounting for their distinct execution paradigms and pricing structures.

4.1.1 Virtual Machines. Virtual machines provide a stateful execution environment characterized by a multi-tier storage hierarchy comprising memory, local SSDs, and remote storage. Unlike serverless services, VMs allow data to be cached locally to reduce I/O latency. Since a VM instance operates within specific hardware provisions, its execution performance is constrained by the bottleneck between compute capacity and I/O bandwidth.

For a specific concurrent group g , let $\hat{D}_{\text{cpu}}^{(g)}$ and $\hat{D}_{\text{io}}^{(g)}$ denote the data volumes required for CPU processing and I/O access, respectively. The compute-bound latency $\hat{T}_{\text{cpu}}^{(g)}$ is calculated as the ratio of the data volume to the effective compute throughput of the VM, denoted as B_{cpu} :

$$\hat{T}_{\text{cpu}}^{(g)} = \frac{\hat{D}_{\text{cpu}}^{(g)}}{B_{\text{cpu}}} \quad (2)$$

The I/O-bound latency $\hat{T}_{\text{io}}^{(g)}$ depends on data placement across the storage hierarchy. We introduce two cache hit ratios $\hat{P}_{\text{mem}}^{(g)}$ and $\hat{P}_{\text{ssd}}^{(g)}$ that represent the fraction of data accessed from memory and local SSDs, respectively. The remaining fraction $1 - \hat{P}_{\text{mem}}^{(g)} - \hat{P}_{\text{ssd}}^{(g)}$ corresponds to data fetched from remote storage. These ratios quantify multi-query data reuse by tracking how queries utilize existing caches. Repeated access to common base data results in higher hit ratios, allowing the cost model to capture the reduced overhead of remote storage transfers. The I/O time is modeled as the weighted sum of access latencies across these tiers:

$$\hat{T}_{\text{io}}^{(g)} = \hat{D}_{\text{io}}^{(g)} \left(\frac{1 - \hat{P}_{\text{mem}}^{(g)} - \hat{P}_{\text{ssd}}^{(g)}}{B_{\text{net}}} + \frac{\hat{P}_{\text{ssd}}^{(g)}}{B_{\text{ssd}}} + \frac{\hat{P}_{\text{mem}}^{(g)}}{B_{\text{mem}}} \right) \quad (3)$$

In this equation, B_{net} , B_{ssd} , and B_{mem} represent the bandwidth limits for remote network, local SSD, and memory, respectively. The expected execution time of concurrent group g is determined by the bottleneck resource.

$$\hat{T}_g = \max(\hat{T}_{\text{cpu}}^{(g)}, \hat{T}_{\text{io}}^{(g)}) \quad (4)$$

The total expected execution time for query q on the VM, denoted as $\hat{T}_{q,sv}$, is the sum of durations of all its sequential concurrent groups: $\hat{T}_{q,sv} = \sum_g \hat{T}_g$.

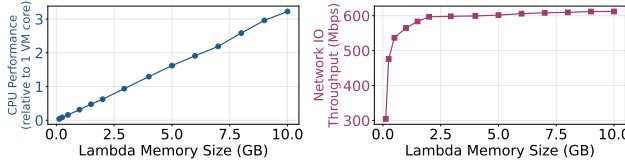


Fig. 3. CPU performance and network throughput of AWS Lambda functions across memory configurations.

Regarding monetary cost, VMs are billed based on instance uptime at a fixed rate R_{vm} per second. However, simply multiplying $\hat{T}_{q,sv}$ by the billing rate R_{vm} would overestimate the 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 the VM. To ensure fair cost attribution, we model each query's compute cost based on its dominant resource footprint, defined as the maximum ratio of CPU, memory, network and disk I/O resource demands relative to the VM's capacity. This ratio, denoted as ρ_q , represents the query's share of the cost.

In addition to compute costs, queries incur charges for data retrieval from cloud storage (e.g., Amazon S3 GET requests), which are billed at a fixed rate R_{get} per request. Let $\hat{N}_{get}^{(q,sv)}$ represent the total number of GET requests required to scan the query's input data. Since local caching eliminates the need for remote retrieval, the costs apply only to the portion of data not found in the local cache. The total monetary cost is modeled as follows:

$$\hat{M}_{q,sv} = \rho_q \cdot \hat{T}_{q,sv} \cdot R_{vm} + (1 - \hat{P}_{mem}^{(q)} - \hat{P}_{ssd}^{(q)}) \cdot \hat{N}_{get}^{(q,sv)} \cdot R_{get} \quad (5)$$

4.1.2 Function-as-a-Service. In EASE, each concurrent group executed on FaaS is parallelized across multiple function instances, with intermediate results shuffled through cloud storage between stages. This execution model follows the serverless execution paradigm in [25, 40, 44]. Specifically, the input data is divided into fixed-size partitions (256 MB in our implementation), and each function instance processes one partition independently. The number of function instances allocated to a concurrent group g , denoted as N_g , is determined based on the data volume processed by the pipelines within the group. Instead of requiring a separate FaaS profile for each query, EASE estimates latency and cost on FaaS from the pipeline-level profiles described in Section 4.2. For a given concurrent group g , let $\hat{D}_{cpu}^{(g)}$ and $\hat{D}_{io}^{(g)}$ denote the volumes of data processed by CPU and network I/O operations, respectively. The expected execution time of g is modeled as:

$$\hat{T}_g = \max \left(\frac{\hat{D}_{cpu}^{(g)}}{N_g \cdot B_{cpu}(m)}, \frac{\hat{D}_{io}^{(g)}}{N_g \cdot B_{io}(m)} \right) \quad (6)$$

where m denotes the memory size allocated to each function instance, which determines its CPU and network I/O throughput, denoted as $B_{cpu}(m)$ and $B_{io}(m)$, respectively. As shown in Figure 3, $B_{cpu}(m)$ scales nearly linearly with m , while $B_{io}(m)$ increases with memory up to 2 GB and plateaus thereafter. To balance cost and performance, we select the smallest memory configuration that minimizes $\hat{T}_g$. For I/O-intensive pipelines, this typically results in around 2-4 GB of memory, as further memory allocation provides diminishing returns. In contrast, CPU-intensive pipelines often require higher memory settings to fully utilize available compute capacity. Given the fixed input partition size of 256 MB, these configurations also ensure sufficient memory for processing the partition to prevent out-of-memory errors during execution.

Between concurrent groups, intermediate data is shuffled through shared storage. This process involves a write phase by the upstream group and a read phase by the downstream group. Let g_{up}

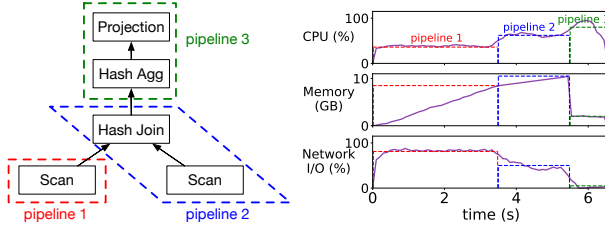


Fig. 4. A query plan decomposed into three pipelines and its corresponding resource usage over time.

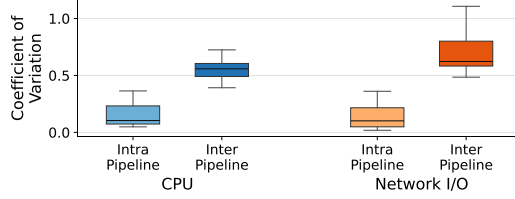


Fig. 5. Coefficient of variation of CPU and network I/O usage within and across pipelines.

and g_{down} denote the upstream and downstream groups, run on $N_{g_{\text{up}}}$ and $N_{g_{\text{down}}}$ function instances, respectively. The shuffle latency $\hat{T}_h$ is modeled as the sum of write and read durations:

$$\hat{T}_h = \frac{\hat{D}_{\text{write}}^{(g_{\text{up}})}}{N_{g_{\text{up}}} \cdot B_{\text{io}}(m)} + \frac{\hat{D}_{\text{read}}^{(g_{\text{down}})}}{N_{g_{\text{down}}} \cdot B_{\text{io}}(m)} \quad (7)$$

FaaS is billed based on execution duration and allocated memory of each function invocation. We denote the billing rate as R_{faas} (USD per GB-second). For the i -th function invocation, let m_i and $\hat{d}_i$ denote its memory size and execution time, respectively. With n invocations across all pipelines, the compute cost is modeled as:

$$\hat{M}_{\text{faas_compute}} = \sum_{i=1}^n \hat{d}_i \cdot m_i \cdot R_{\text{faas}} \quad (8)$$

In addition to compute costs, executing queries on FaaS incurs storage access costs from input data retrieval and intermediate data shuffling. Data retrieval and shuffling generate GET and PUT requests. Let $\hat{N}_{\text{get}}^{(q, s_F)}$ and $\hat{N}_{\text{put}}^{(q, s_F)}$ denote 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} , respectively. The total storage access cost is modeled as:

$$\hat{M}_{\text{faas_storage}} = \hat{N}_{\text{get}}^{(q, s_F)} \cdot R_{\text{get}} + \hat{N}_{\text{put}}^{(q, s_F)} \cdot R_{\text{put}} \quad (9)$$

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

$$\hat{T}_{q, s_F} = \sum_{g \in \text{groups}} \hat{T}_g + \sum_{h \in \text{shuffles}} \hat{T}_h \quad (10)$$

$$\hat{M}_{q, s_F} = \hat{M}_{\text{faas_compute}} + \hat{M}_{\text{faas_storage}} \quad (11)$$

Our model focuses on measurable and stable components that dominate the performance of data-intensive OLAP workloads, omitting stochastic factors such as cold start latency and straggler effects, which are difficult to predict accurately. While these factors can introduce variability to the tail latency, existing techniques such as pre-warming [35] and duplicate requests [44] can reduce their impact in practice. The goal of our cost model is to compare candidate services for scheduling

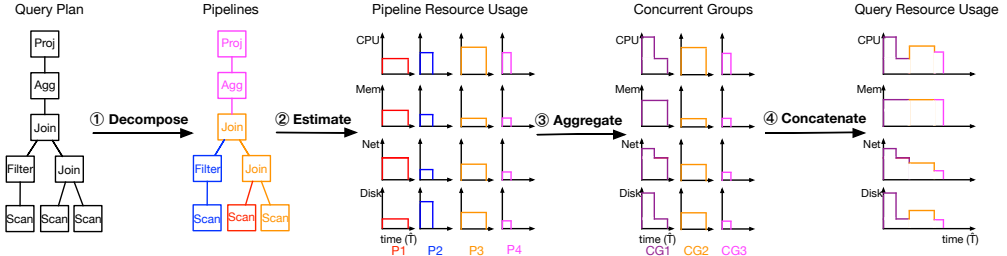


Fig. 6. Workflow for deriving query-level resource demands from pipeline profiles.

decisions, instead of predicting latency with perfect precision. As shown in Section 7, our model is sufficient to identify the optimal service in the presence of runtime noise.

4.1.3 Query-as-a-Service. Modeling QaaS presents a unique challenge due to the opacity of internal resource allocation, interference from multi-tenant resource sharing, and continuous evolution of the underlying execution engine. Consequently, the white-box resource modeling approach used for VMs and FaaS is not applicable. To address this, we adopt the learned query performance prediction technique proposed in BRAD [56], which is designed to model opaque cloud services where internal metrics are inaccessible and has demonstrated high accuracy in modeling QaaS platforms. Following this methodology, we extract a feature vector x_q for each query q , encompassing logical query properties (e.g., operator types, filter selectivities) and dataset statistics. We then map the feature vector into expected execution time and monetary cost using the learned regression models, denoted as f_T and f_D , respectively.

The expected execution time is modeled as $\hat{T}_{q,s_Q} = f_T(x_q)$. Since QaaS pricing typically depends on the amount of data scanned, the monetary cost is calculated as $\hat{M}_{q,s_Q} = f_D(x_q) \cdot R_{\text{scan}}$, where R_{scan} denotes the price per unit of data scanned. By periodically training on recent execution logs, these models characterize the current runtime behavior of the service. This approach allows EASE to adapt to the evolution of underlying execution engine despite the opacity of internal mechanisms.

4.2 Fine-grained Resource Profiling

To enable accurate cost modeling and effective resource-aware scheduling across heterogeneous services, our framework requires resource demands of each query to be estimated at the pipeline granularity rather than the query level. This is because OLAP queries typically consist of multiple execution pipelines with significantly varying resource requirements. As shown in Figure 4, resource usage tends to remain stable within each pipeline but changes drastically across pipelines. To validate this pattern, we measure resource time series from TPC-H queries and compute the coefficient of variation, defined as the standard deviation divided by the mean. Figure 5 shows that resource variation within individual pipelines is much lower than the variation across different pipelines. To capture this, we propose a workflow that builds query resource profiles from individual pipelines. As illustrated in Figure 6, the workflow consists of four sequential phases.

Decomposition. The first phase decomposes the query plan into a DAG of pipelines at blocking operators such as hash joins and aggregations, following the execution model of modern OLAP systems [8, 13]. Each pipeline represents a sequence of operators that can execute without blocking. A pipeline starts only after all its upstream pipelines complete, and independent pipelines can execute in parallel.

Estimation. For each decomposed pipeline, we estimate a resource profile consisting of four key metrics: average CPU utilization, peak memory consumption, average network and disk I/O throughput. We treat estimation as an orthogonal upstream task, agnostic to the specific

method used. These metrics can be derived from various sources, such as lightweight profiling [46], analytical models [22, 26, 52], or learning-based predictors [27, 30, 34, 38, 48, 58], depending on the user's context and available data. Operating at the pipeline granularity reduces prediction variance compared to query-level estimation, as pipelines exhibit more stable resource usage patterns.

Aggregation. We then reconstruct the query's parallel execution behavior by grouping pipelines that can execute concurrently into **concurrent groups**. Within each group, we aggregate the resource demands to model resource contention. For CPU and I/O, we sum the usage of all pipelines at each time point. If the aggregated demand exceeds the VM's capacity (i.e., over 100% utilization), we scale the group's execution duration by the ratio of the dominant resource demand to the VM's capacity to account for contention. For memory, we conservatively derive the peak footprint of a concurrent group by summing the peak memory consumption of its constituent pipelines. If the sum exceeds the VM's memory limit, the query cannot be executed on the VM due to the risk of out-of-memory failure or severe performance degradation from disk spilling.

Concatenation. Finally, we treat each concurrent group as a single logical unit and concatenate their resource demands according to the execution order determined by data dependencies. The final query resource profile consists of piecewise constants for CPU and I/O usage over time, and a global peak memory representing the maximum memory demand across all concurrent groups. This peak value is used during scheduling to ensure sufficient memory capacity for query execution, since exceeding available memory can severely degrade performance or cause execution failures.

5 Real-time Query Scheduler

The query scheduler serves as the core decision-making component of EASE, responsible for determining the placement and execution start time of queries across heterogeneous compute services. This section presents our initial query placement strategy in Section 5.1, followed by the formulation of the scheduling problem in Section 5.2 and the resource-aware scheduling algorithm in Section 5.3.

5.1 Initial Query Placement

The scheduling process begins by identifying the optimal compute service for each incoming query. Using the heterogeneity-aware cost model in Section 4.1, the scheduler calculates the expected execution time and monetary cost of the query on each service. It then selects the service s^* that minimizes the objective function, which balances the trade-off between performance and cost while incorporating a load penalty to prevent service overloading.

$$s^* = \operatorname{argmin}_{s \in S} \left(\hat{T}_{q,s} \cdot \hat{M}_{q,s}^\alpha \cdot (1 + \beta_s \cdot L_s) \right) \quad (12)$$

where S is the set of available compute services, $\hat{T}_{q,s}$ and $\hat{M}_{q,s}$ denote the expected execution time and monetary cost of the query q on service s , and α is the tunable parameter that controls the relative importance of cost versus performance, as described in Section 2.2. To prevent service overloading, we incorporate a load 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 for fine-grained tuning over the balance between performance and load distribution.

The value of β_s is configured based on the service's elasticity. Less elastic services use a larger β_s because additional queued queries are more likely to increase waiting time and resource contention. More elastic services use a smaller β_s , as they can scale more rapidly to accommodate increased load. In our experiments, we set $\beta_V = 0.1$ for VMs, $\beta_F = 0.05$ for FaaS, and $\beta_Q = 0.02$ for QaaS, reflecting the relative elasticity of the three services.

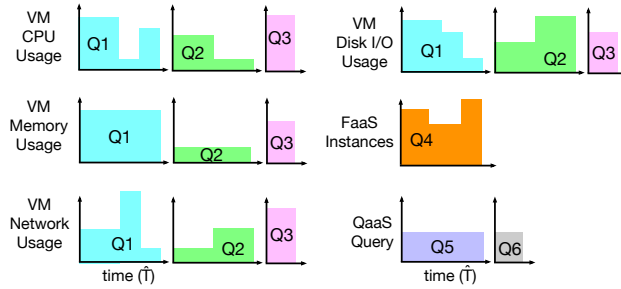


Fig. 7. Resource representations for VMs (CPU, memory, network, and disk I/O), FaaS (concurrent function instances), and QaaS (query slots).

5.2 Scheduling Problem Formulation

Since queries placed on the service s share the same resource pool, unrestricted concurrent execution can lead to severe resource contention and performance degradation. To address this, the scheduler must determine the appropriate execution start time for each query. We formulate this as a resource-constrained scheduling problem, formally defining the objective and constraints in this section.

We model each query as a sequence of resource rectangles aligned along the time axis, where each rectangle corresponds to a concurrent group derived in Section 4.2. The rectangle's width represents the group's execution duration, and its height represents the aggregated resource demand. The meaning of the resource dimension varies according to the capacity model of each service. On VMs, the height of a rectangle represents the aggregated demand of CPU, memory, network, or disk I/O throughput of the group. On FaaS, the height indicates the peak number of concurrent function instances required for the group. On QaaS, each query occupies one slot throughout its execution, as QaaS typically enforces query slot limits at the user level. Although FaaS and QaaS use consumption-based pricing models, they still expose finite capacity constraints through function concurrency quotas and query slot limits. Resource-aware scheduling is still useful for these services because it reduces queueing and contention when service capacity is saturated. Figure 7 shows the resource representation.

The scheduler determines optimal start times for queued queries to minimize total response time while respecting the service's resource capacity constraints. Let $Q^{(s)} = \{q_1, \dots, q_n\}$ be the set of queries queued on s . Each query $q_i \in Q^{(s)}$ arrives at time A_i and consists of n_i concurrent groups $g_{i1}, g_{i2}, \dots, g_{in_i}$. Each group g_{ij} is characterized by its execution duration $\hat{d}_{ij}^{(s)}$ and 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, network, and disk I/O for VM, function concurrency for FaaS, and query slots for QaaS).

Each service s is associated with a resource capacity vector $\mathbf{c}_{\text{capacity}}^{(s)} \in \mathbb{R}^k$, which represents the maximum capacity of each resource. The scheduler assigns a start time $t_i \geq A_i$ for each query q_i , such that the resource consumption of all active queries (denoted $\mathcal{A}(t)$) does not exceed the service's capacity at any time t :

$$\forall t, \sum_{q_i \in \mathcal{A}(t)} \sum_{g_{ij} \text{ active at } t} \hat{\mathbf{r}}_{ij}^{(s)} \leq \mathbf{c}_{\text{capacity}}^{(s)} \quad (13)$$

The objective is to minimize the total response times:

$$\min \sum_{j=1}^n (C_j - A_j) \quad (14)$$

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

5.3 Resource-aware Scheduling Algorithm

The resource-constrained scheduling problem is NP-hard. This can be shown by reduction from the classical single-machine scheduling problem with release time and the objective of minimizing total completion time ($|r_j| \sum C_j$) [33], which is known to be NP-hard even for single-stage jobs and a single resource dimension. Given this complexity, we design a lightweight heuristic algorithm for real-time scheduling. We discretize the timeline into fixed-length time slots and model each query's resource usage over a finite horizon of H slots. For each query q on service s , its resource demand is represented as a matrix $\hat{\mathbf{R}}_q^{(s)} \in \mathbb{R}^{k \times H}$, where k is the number of resource dimensions, and each entry $\hat{\mathbf{R}}_q^{(s)}[i, j]$ denotes the demand for resource type i in time slot j . Correspondingly, we maintain a remaining capacity matrix $\hat{\mathbf{C}}_{\text{remain}}^{(s)} \in \mathbb{R}^{k \times H}$, representing the remaining capacity across all resource dimensions over H time slots. To simplify the representation, we flatten both matrices into vectors $\hat{\mathbf{r}}_q^{(s)} \in \mathbb{R}^{kH}$ and $\hat{\mathbf{c}}_{\text{remain}}^{(s)} \in \mathbb{R}^{kH}$.

The scheduling algorithm is triggered by events 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 can be accommodated within 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 both its resource demand's alignment with the remaining capacity and its overall resource footprint:

$$\text{score}(q) = \frac{\hat{\mathbf{c}}_{\text{remain}}^{(s)} \cdot \hat{\mathbf{r}}_q^{(s)}}{\|\hat{\mathbf{r}}_q^{(s)}\|^2} \quad (15)$$

The scoring function encapsulates two heuristic principles. The numerator acts as a reward for resource affinity. It favors queries whose resource usage aligns with the remaining capacity, improving packing efficiency and reducing resource fragmentation. The denominator acts as a penalty for resource intensity. It penalizes queries with large or long-lasting resource demands that may block other queries and increase contention. As a result, the compatibility score prioritizes queries that are less resource-intensive and well-matched to the remaining capacity.

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.

Runtime calibration. The resource-aware scheduling algorithm relies on resource usage derived from the query's resource profile, which may be inaccurate due to runtime variance and limitations of estimation methods. To mitigate the impact of inaccuracy, we continuously monitor the actual resource consumption of running queries and periodically correct the remaining capacity matrix $\hat{\mathbf{C}}_{\text{remain}}^{(s)}$ based on observed consumption. For FaaS and QaaS, runtime calibration updates how long function instances or query slots are expected to remain occupied, which affects when subsequent queries can be scheduled. This calibration limits error propagation and ensures scheduling decisions rely on up-to-date system state rather than outdated estimations. Furthermore, this observed data can be fed back to the upstream estimation module to improve its accuracy over time.

Max-wait safeguard. To prevent starvation, EASE enforces a max-wait policy for long-waiting queries. The resource-aware scheduling may delay resource-heavy or long-running queries when many short queries keep arriving, because short queries often obtain higher scores due to their lower resource intensity. Once a queued query exceeds the max-wait threshold, the scheduler bypasses the compatibility-based ordering and dispatches the query for execution on the least-loaded service. This ensures that resource-heavy queries are not indefinitely postponed.

Algorithm 1: Resource-aware Query Scheduling

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

```

1  $S \leftarrow \emptyset$ 
2  $\mathcal{F} \leftarrow \{q \in Q \mid \hat{\mathbf{r}}_q^{(s)} \leq \hat{\mathbf{c}}_{\text{remain}}^{(s)}\}$ 
3 while  $\mathcal{F} \neq \emptyset$  do
4   foreach  $q \in \mathcal{F}$  do
5      $\text{score}(q) \leftarrow \frac{\hat{\mathbf{c}}_{\text{remain}}^{(s)} \cdot \hat{\mathbf{r}}_q^{(s)}}{\|\hat{\mathbf{r}}_q^{(s)}\|^2}$ ;
6    $q^* \leftarrow \operatorname{argmax}_{q \in \mathcal{F}} \text{score}(q)$ ;
7    $S \leftarrow S \cup \{q^*\}$ ;
8    $\hat{\mathbf{c}}_{\text{remain}}^{(s)} \leftarrow \hat{\mathbf{c}}_{\text{remain}}^{(s)} - \hat{\mathbf{r}}_{q^*}^{(s)}$ ;
9    $\mathcal{F} \leftarrow \{q \in \mathcal{F} \setminus \{q^*\} \mid \hat{\mathbf{r}}_q^{(s)} \leq \hat{\mathbf{c}}_{\text{remain}}^{(s)}\}$ ;
10 return  $S$ 

```

Scheduling granularity. EASE schedules at the query level to keep scheduling overhead low and avoid frequent state synchronization across distributed workers. Scheduling at a finer granularity such as individual pipelines would require cross-service coordination and intermediate state management. However, since EASE already builds its cost model from pipeline-level profiles, the framework can be extended to schedule individual pipelines. We leave this for future work.

6 Dynamic Rescheduler

The dynamic rescheduler serves as a corrective mechanism to handle performance instability in the cloud environment. Even with optimal initial scheduling decisions, various factors like multi-tenant interference or network jitter can still cause unexpected congestion. To mitigate these bottlenecks, the rescheduler migrates queries that are still waiting in service queues to alternative services. Before migration, it checks the query status on the original service. If the query has already been dispatched for execution, the migration is cancelled to avoid redundant execution. We first describe the identification of migration candidates in Section 6.1, followed by the target service selection strategy in Section 6.2.

6.1 Migratable Query Identification

To identify queries experiencing excessive stagnation, we define a migration eligibility rule that considers both the query's relative waiting time and a minimum absolute delay. A query q becomes a migration candidate if its current waiting time $W_q^{(s)}$ satisfies:

$$W_q^{(s)} > \max(\gamma \cdot \hat{T}_{q,s}, \tau) \quad (16)$$

The parameter γ acts as a patience factor that controls how long a query is required to wait relative to its expected runtime before becoming eligible for migration, preventing premature migrations of long-running queries. The parameter τ imposes a minimum delay threshold, ensuring that short queries are not migrated due to transient fluctuations. In our evaluation, we configure $\gamma = 1$ and set τ to the system's average query latency, balancing responsiveness and stability.

6.2 Target Service Selection

For each migration candidate, the rescheduler evaluates potential target services by calculating the expected completion time, including waiting time and execution time on the service. While the execution time is derived from the heterogeneity-aware cost model (Section 4), the waiting time must be determined by the resource pressure of the target service.

We first reconstruct the service's resource state. Let $\hat{\mathbf{R}}_q^{(s)} \in \mathbb{R}^{k \times H}$ be the resource demand matrix of query q on service s , with k resource dimensions and a time horizon of H slots, where each slot represents a fixed time duration Δt (e.g., 1 second). 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 capacity of a given resource type. To model temporal resource availability, we construct a capacity matrix $\mathbf{C}_{\text{capacity}}^{(s)} \in \mathbb{R}^{k \times H}$ by replicating $\mathbf{c}_{\text{capacity}}^{(s)}$ across all H time slots:

$$\mathbf{C}_{\text{capacity}}^{(s)} = \begin{bmatrix} \mathbf{c}_{\text{capacity}}^{(s)} & \cdots & \mathbf{c}_{\text{capacity}}^{(s)} \end{bmatrix} \in \mathbb{R}^{k \times H} \quad (17)$$

The occupied resource matrix $\hat{\mathbf{C}}_{\text{used}}^{(s)}$ is then derived as:

$$\hat{\mathbf{C}}_{\text{used}}^{(s)} = \mathbf{C}_{\text{capacity}}^{(s)} - \hat{\mathbf{C}}_{\text{remain}}^{(s)} \quad (18)$$

We estimate the waiting time $\hat{W}^{(s)}$ on service s by aggregating resource consumption of all pending queries, including both queued and running queries. For each resource dimension i , we sum the demands of all queued queries $Q^{(s)}$ and the residual consumption of currently running queries (captured in $\hat{\mathbf{C}}_{\text{used}}^{(s)}$). We then normalize this total resource demand by the resource capacity to determine the backlog duration, which is the time required to process the accumulated load. The expected waiting time is defined as the maximum backlog duration across all resource dimensions:

$$\hat{W}^{(s)} = \max_{i=1, \dots, k} \left(\frac{\sum_{q \in Q^{(s)}} \sum_{j=1}^H \hat{\mathbf{R}}_q^{(s)}[i, j] + \sum_{j=1}^H \hat{\mathbf{C}}_{\text{used}}^{(s)}[i, j]}{\mathbf{c}_{\text{capacity}}^{(s)}[i]} \right) \cdot \Delta t \quad (19)$$

Finally, we select the target service s^* that minimizes the weighted cost-performance objective:

$$s^* = \underset{s \in S}{\operatorname{argmin}} \left(\hat{W}^{(s)} + \hat{T}_{q,s} \right) \cdot \hat{M}_{q,s}^\alpha \quad (20)$$

The query is migrated if the objective value of the selected target service is lower than the current service. Before migration, the rescheduler checks the query status on the current service. The migration proceeds only if the query is still queued. Otherwise, the migration is cancelled to avoid duplicate execution.

Failure handling. In practice, queries may encounter transient errors or persistent failures. EASE handles transient errors such as S3 throttling and temporary invocation failures through retries within the original service. For persistent failures such as out-of-memory errors, EASE marks the execution as failed and returns the query to the rescheduler. The rescheduler then treats the affected query as a migration candidate and includes its accumulated delay when selecting an alternative service for re-execution.

7 Evaluation

In this section, we evaluate EASE on a set of OLAP workloads to assess its performance and cost-efficiency. Our evaluation includes end-to-end comparisons with baselines (Section 7.2), ablation studies (Section 7.3) quantifying the contributions of individual components, and additional experiments (Section 7.4) covering cost-model accuracy, parameter sensitivity, and runtime overhead.

Table 4. Characteristics of evaluated benchmarks.

Benchmark	Tables	Max Joins	Data Size (Parquet format)
ClickBench	1	–	13.7 GB
SSB (SF 300)	5	4	61.3 GB
TPC-H (SF 300)	8	8	103.8 GB
JOB	21	17	1.8 GB
Stack	10	12	27.6 GB

Table 5. Statistics of workload patterns.

Pattern	Duration (min)	#Queries	Avg. QPM	Peak QPM	CoV
High burst	20	161	8.05	52	1.31
Mid peak	30	267	8.90	31	0.86
Rising load	40	259	6.48	31	1.09
Falling load	60	452	7.53	23	0.81

7.1 Experimental Setup

Platform. We conducted our experiments on AWS using a diverse set of services to represent the heterogeneous cloud environment:

- **VM:** EC2 [3] m5.8xlarge instances (32 vCPUs, 128 GB memory, and 10Gbps network bandwidth), billed at \$1.536 per hour.
- **FaaS:** Lambda [9] with a default concurrency limit of 1000 function invocations, priced at \$0.0000167 per GB-second.
- **QaaS:** Athena [10] supporting 30 concurrent queries, charged at \$5 per TB of data scanned.

Implementation details. Our experiments were conducted in the us-east-2 region. All data is stored in Amazon S3 and can be accessed directly by all compute services without additional replication. For EC2-based execution, local caches are populated on demand as data blocks are fetched from S3 using DuckDB cache_https extension [20], which applies an LRU replacement policy. To ensure semantic consistency across different services, we utilize SQLGlot [19] to transpile queries into the target system’s dialect. We employ DuckDB [13] as the underlying execution engine for both EC2 and Lambda. Specifically, Lambda execution follows a serverless data processing model similar to existing serverless execution engines [25, 40, 44]. In this model, DuckDB runs on each function instance, processing multiple input data partitions in parallel and exchanging intermediate results via S3. To adapt to dynamic workloads, the EC2 cluster is configured with the target tracking auto-scaling policy, maintaining an average CPU utilization of 70%. We use on-demand provisioning without a warm pool, so instance startup delays are included in the evaluation. All EC2 instances use the same m5.8xlarge nodes. While our implementation is specific to AWS, the design principles are general and portable to other cloud providers. In the evaluation, we use query plans and table statistics to estimate the data processed by each pipeline and shuffled data between pipelines. Cache status is obtained via the DuckDB cache_https extension, and S3 request counts are derived from query access ranges and cache status.

Benchmarks. We evaluate EASE using five OLAP benchmarks: ClickBench [16], SSB [42], TPC-H [1], JOB [32], and Stack [36]. These benchmarks cover diverse query patterns, including scan-heavy analytics, star-schema joins, decision-support queries, and real-world analytical workloads with complex joins. Table 4 summarizes the characteristics of these benchmarks. All datasets are stored in Parquet format with Snappy compression.

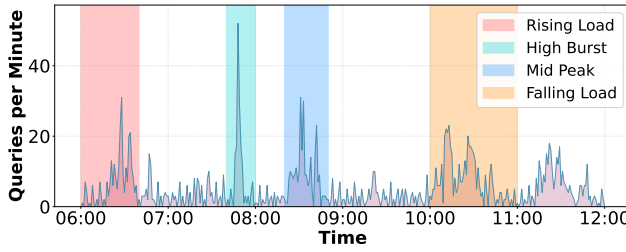


Fig. 8. Query arrival rates over a six-hour trace (6:00–12:00, February 28, 2018) from a single database in the Snowset [49]. The shaded regions indicate the four representative workload patterns used in our experiments.

Workload patterns. To evaluate system adaptability under realistic dynamics, we generate workloads based on Snowset [49], a large-scale query trace collected by Snowflake. As shown in Figure 8, 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 sudden spikes in query arrivals; (2) **mid peak**, where the arrival rate peaks in the middle of the workload; (3) **rising load**, where query load gradually increases; and (4) **falling load**, where query load gradually decreases. Table 5 summarizes the statistics of these patterns, including duration, total queries, average and peak queries per minute (QPM), and coefficient of variation (CoV) of arrival rates. For each arrival event in the trace, we randomly select a query from the benchmark’s query set.

Baselines. We compare EASE with the following baselines:

- **VM-only, FaaS-only, and QaaS-only.** These baselines assign all queries to a single type of compute service.
- **Pixels [25].** A rule-based system that assigns queries to the VM service by default and offloads to FaaS when the VM queue exceeds a threshold. In our implementation, we set the threshold to 3, following the original configuration in [25].
- **Tetris [29].** A cluster scheduler that determines task placement based on resource alignment. It focuses on maximizing resource utilization within a fixed set of hardware resources. We implement its scheduling logic in the same framework as EASE, treating each query as a task.
- **MemQ [53].** A memory-centric scheduler that employs the bidirectional fit packing algorithm. It prioritizes tasks based on memory footprint, ignoring other dimensions. We apply its packing policy at query granularity within each service.
- **Redshift-Serverless (Redshift-SL) [15].** A state-of-the-art commercial serverless cloud data warehouse service provided by AWS that automatically scales resources based on workload demands. We configure it to balanced mode (50% price-performance target [21, 41]).

Metrics. We use the following performance and cost metrics:

- **Latency:** elapsed time from query arrival to completion, including queueing and execution time.
- **Total Cost:** the total monetary cost incurred for processing the workload, including compute and storage charges.

7.2 Overall Results

We evaluate EASE against all baselines across three benchmarks and four workload patterns. The analysis covers three aspects: (1) average query latency and total cost (Figure 9); (2) latency distribution (Figure 10); and (3) system responsiveness under load spikes measured by the number of pending queries over time (Figure 11).

As shown in Figure 9, EASE achieves the lowest average latency across all benchmarks and workload patterns, and also offers strong cost efficiency. On SSB and TPC-H, it delivers the lowest

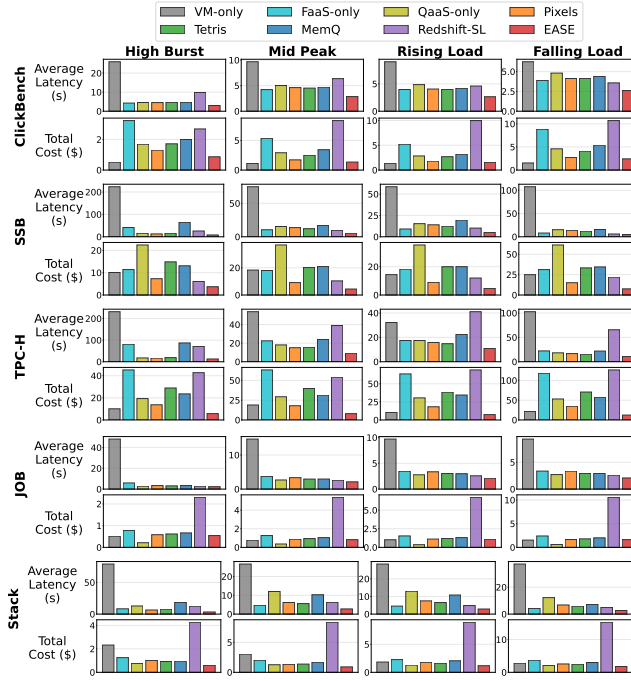


Fig. 9. Average latency and total cost.

total cost, with improvements of up to 35% in latency and 67% in cost compared to the best-performing baseline. On ClickBench, its cost is slightly higher than VM-only, but remains lower than all other baselines. This is because most ClickBench queries are simple and lightweight, resulting in low resource demands that rarely trigger auto-scaling under the VM-only strategy, keeping its total cost low. 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 higher latency and cost. For the more complex JOB and Stack benchmarks, EASE also achieves the lowest average latency across all workload patterns. On JOB, its total cost is higher than QaaS-only but remains lower than the other baselines. The main reason is that the data size of JOB is small, making QaaS more cost-effective since it charges based on the amount of data scanned. On Stack, EASE achieves the lowest latency and cost, demonstrating the benefit of heterogeneous service scheduling for complex join-intensive workloads. Redshift-SL exhibits inconsistent performance due to the limitations of its auto-scaling mechanism. It often struggles to precisely match resource provisioning with demand, either over-provisioning during light loads or under-provisioning during sudden spikes, resulting in suboptimal latency and cost. In contrast, EASE leverages the heterogeneity-aware cost model and resource-aware scheduling to effectively distribute queries across services, achieving superior performance and cost-efficiency across diverse workloads.

Figure 10 presents the cumulative distribution of query latency, demonstrating that EASE not only achieves the lowest average latency but also maintains low tail latency. Redshift-SL performs well in median latency under light or stable loads due to its abundant resource provisioning but exhibits a heavy tail during load spikes caused by scaling lag. Pixels relies on queue-based offloading rules, failing to account for query resource demands and causing suboptimal placement. Similarly, heterogeneity-oblivious schedulers like Tetris and MemQ focus on packing efficiency within a service, ignoring the disparities of different services and leading to prolonged execution

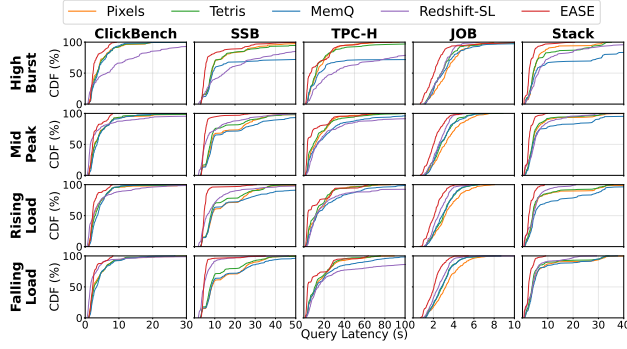


Fig. 10. Cumulative distribution function of query latency.

for mismatched queries. EASE effectively mitigates these issues by dynamically matching query resource requirements with service characteristics, ensuring both low average and tail latency.

Figure 11 tracks the number of pending queries over time under dynamic workloads. Redshift-SL shows a significant backlog accumulation during sudden load spikes due to the delay in its auto-scaling response, which results in temporarily under-provisioning of resources. Pixels and Tetris both help reduce queue buildup, with Pixels leveraging heterogeneous services and Tetris packing queries efficiently within the service. MemQ clears backlogs more slowly because it only models memory usage and overlooks CPU and I/O demands, leading to resource contention and prolonged query execution. EASE continuously monitors system state and dynamically redistributes queued queries to the suitable services, effectively mitigating queue buildup and ensuring efficient processing of queries.

7.3 Ablation Study

We conduct an ablation study on TPC-H under four workload patterns to evaluate the contribution of each core component in EASE. We evaluate the following three variants against the complete system: (1) disabling the cost model, which is replaced by random service selection for initial placement; (2) disabling resource-aware scheduling, which is substituted with first-come-first-served (FCFS) scheduling policy; and (3) disabling the dynamic rescheduler, which removes the ability to migrate queries across services.

Figure 12 shows that all three components contribute to the overall performance of EASE. Disabling the cost model leads to frequent mismatches between queries and services, resulting in higher latency and cost. Removing the dynamic rescheduler decreases total cost slightly, as queries are no longer migrated to potentially more expensive but idle services. However, this comes at the expense of higher latency, especially under bursty workloads where latency increases by up to 109% without migration. Disabling resource-aware scheduling also increases latency due to more resource contention within services, and indirectly raises cost as longer waiting times trigger more migrations by the dynamic rescheduler. These results demonstrate that each component plays a distinct and necessary role in reducing latency and cost across dynamic workloads.

7.4 Additional Experiments

7.4.1 Model accuracy. We evaluate the accuracy of our heterogeneity-aware cost model against BRAD [56], a learned query performance prediction model designed for cloud environments. Table 6 reports the median Q-error of both models across different services. For QaaS, EASE and BRAD exhibit identical accuracy, as EASE adopts the same approach to model the opaque performance

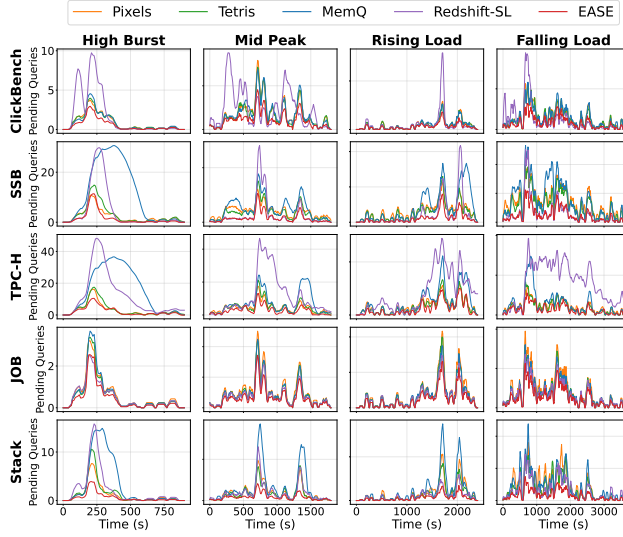


Fig. 11. Number of pending queries over time.

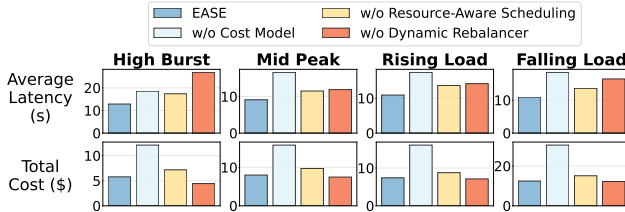


Fig. 12. Ablation study on TPC-H under four workloads.

characteristics of QaaS. For VMs and FaaS, our white-box models outperform BRAD in both latency and cost predictions, reducing Q-error by up to 25%.

More importantly, the primary goal of our model is to guide relative ranking for query placement rather than absolute prediction. Figure 13 shows that EASE identifies the optimal service more accurately than BRAD. These results demonstrate that our model effectively captures the distinct characteristics of heterogeneous services, enabling accurate query scheduling decisions.

7.4.2 Impact of parameter α . We evaluate how EASE responds to different user preferences by varying α (defined in Section 2.2), which controls the trade-off between query latency and cost. As shown in Figure 14a, increasing α leads to higher average latency and lower total cost, demonstrating that EASE provides a flexible and effective trade-off between performance and cost.

Figure 14b further shows how EASE realizes this trade-off through adaptive query placement. At low α , EASE prioritizes performance by placing more queries to FaaS, which offers rapid scaling and low-latency execution, especially for short queries. As α increases, EASE shifts more queries to QaaS and VMs, particularly those with high resource demands, because these services provide better cost efficiency than FaaS. These results show that EASE adapts its scheduling decisions to user preferences by selecting services that best match the desired cost-performance trade-off.

7.4.3 Sensitivity to scheduling parameters. EASE uses several internal scheduling parameters to control load balancing and migration decisions. We evaluate the sensitivity of these parameters on TPC-H benchmark by varying one parameter at a time and measuring the resulting latency, cost, and migration behavior.

Service	Target	BRAD	EASE
VM	Latency	1.42	1.23
	Cost	1.45	1.29
FaaS	Latency	1.85	1.38
	Cost	1.92	1.49
QaaS	Latency	1.31	1.31
	Cost	1.25	1.25

Table 6. Median Q-error.

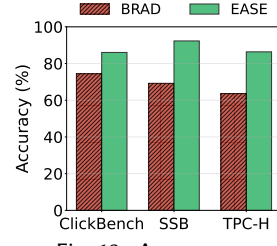
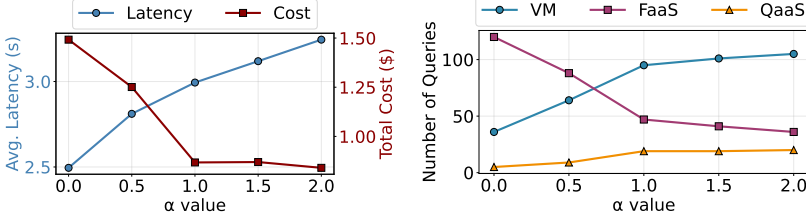


Fig. 13. Accuracy.



(a) Latency and cost

(b) Distribution across services

Fig. 14. Impact of parameter α on ClickBench under high burst workload.

Load penalty parameter β_s . We first evaluate the sensitivity of the load penalty parameter β_s in Equation 12. Since β_s controls the strength of the load penalty for each service, we preserve the relative ordering of the default setting and scale all values by a factor λ , i.e., $(\beta_V, \beta_F, \beta_Q) = \lambda \cdot (0.1, 0.05, 0.02)$. As shown in Figure 15a, a smaller load penalty ($\lambda = 0.5$) leads to higher latency because queries are more likely to be placed on busy services. When λ increases from 1.0 to 4.0, latency decreases slightly, while cost rises by 10% because a larger penalty value favors less-loaded services that may have higher execution costs. These results indicate that a weak load penalty increases queueing delay, while an excessively strong load penalty raises cost without significant latency improvement.

Migration parameters γ and τ . We analyze the sensitivity of the migration patience factor γ and delay threshold τ in Equation 16, with γ varied from 0.5 to 4.0 and τ scaled as a multiple of the average execution time. As shown in Figures 15b, 15c, and 16, smaller values trigger more migrations, which reduce latency but increase cost due to more frequent use of expensive services. Conversely, larger values reduce the number of migrations, which save monetary cost but slightly increase latency. Overall, latency and cost remain stable across these settings, indicating that the rescheduler does not introduce system-level thrashing.

7.4.4 Scoring function ablation. To verify the effectiveness of our scoring function in Equation 15, we compare it with two variants. The affinity-only variant uses only the numerator, and the intensity-only variant uses only the denominator. As shown in Figure 17a, the full scoring function achieves the lowest latency among all variants. The affinity-only variant favors queries whose resource demands align with the remaining capacity, but does not penalize queries with large or long-lasting resource demands. As a result, resource-heavy queries may occupy capacity for long periods and delay subsequent queries. In contrast, the intensity-only variant prefers lightweight queries, but it ignores whether their resource demands fit the currently available capacity, leading to inefficient resource utilization. These results show that both affinity and intensity are needed for effective scheduling.

7.4.5 Effect of max-wait policy. We evaluate the max-wait policy in Section 5.3 using a skewed workload designed to stress long-query starvation. The workload follows the high burst arrival

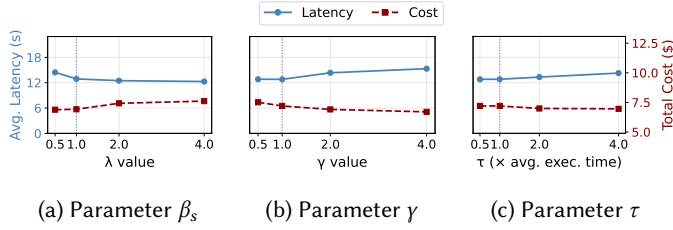


Fig. 15. Sensitivity of scheduling parameters on TPC-H.

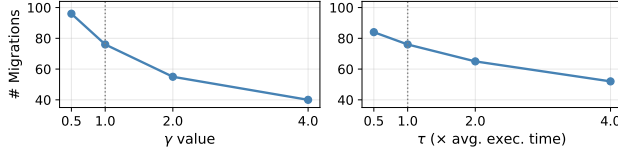


Fig. 16. Number of query migrations under different parameter settings on TPC-H.

pattern, with 90% short ClickBench queries and 10% long SSB queries. We set the max-wait threshold to 40 seconds, which is comparable to the longest execution time in this workload. As shown in Figure 17b, enabling the max-wait policy reduces the p95 latency of long queries by 57.12%, with 16.77% increase in the p95 latency of short queries. The result shows that the policy effectively mitigates starvation of long queries while keeping the impact on short queries limited.

7.4.6 Effect of runtime calibration. We evaluate runtime calibration on the TPC-H falling-load workload, where EASE periodically calibrates cost model parameters based on observed query performance every five minutes. As shown in Figure 17c, the latency Q-error decreases as more runtime observations are collected. This demonstrates that feedback-driven calibration effectively improves cost model accuracy, enabling better scheduling decisions over time.

7.4.7 Query migration statistics. Table 7 reports the number and percentage of queries migrated by the dynamic rescheduler across all benchmarks and workload patterns. Migrations occur more frequently under bursty workloads. For example, the high burst pattern triggers 56 migrations, accounting for 6.96% of queries. Across all settings, 229 queries are migrated, representing 4.02% of all queries. These results show that the rescheduler affects only a small fraction of queries, with most queries remaining on their initially assigned services.

7.4.8 Sensitivity to VM auto-scaling policy. We assess the sensitivity of EASE to the auto-scaling policies by varying the EC2 target CPU utilization from 50% to 90%. As shown in Figure 18a, a higher target delays scale actions, leading to longer queueing and increased resource contention. However, the performance variation remains small. This resilience is primarily enabled by the dynamic rescheduler. When higher utilization targets delay scaling and cause queue accumulation, the rescheduler detects the excessive waiting times and proactively migrates affected queries to alternative services, mitigating the impact of resource provisioning latency.

7.4.9 Scalability under varying FaaS concurrency limits. To evaluate the scalability of EASE with respect to FaaS concurrency, we vary the maximum number of concurrent Lambda invocations from 1000 to 2500. Figure 18b shows that EASE scales better than baselines, as it proactively places embarrassingly parallel and bursty queries on FaaS, fully utilizing the increased concurrency. In contrast, Pixels and Tetris ignore the compatibility between query demands and service characteristics, so they fail to leverage the additional concurrency, resulting in limited performance improvements.

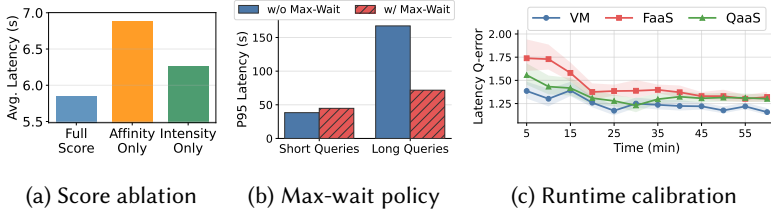


Fig. 17. Effect of key mechanisms: (a) scoring function ablation; (b) p95 latency without and with max-wait policy; (c) latency estimation error over time with runtime calibration.

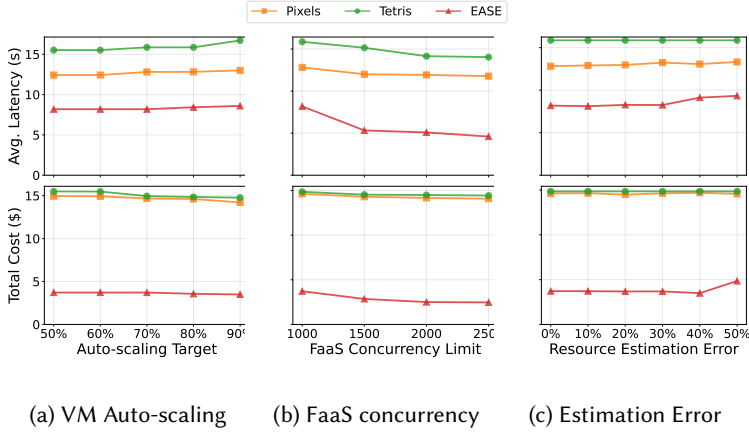


Fig. 18. Sensitivity analysis on SSB high burst workload.

7.4.10 Robustness to resource estimation error. We evaluate the robustness of EASE to resource estimation error by injecting random deviations from 0% to 50% into the resource demands of queries. As shown in Figure 18c, the increase in latency and cost remains below 20% compared to the error-free case.

This robustness is attributed to two factors. First, our heterogeneity-aware model captures the distinct characteristics of each service. Even with moderate estimation errors, the relative ranking of services remains correct, making the scheduling decisions resilient. Second, the runtime calibration mechanism in our scheduler continuously corrects estimation error based on real-time monitoring, preventing error accumulation over time.

7.4.11 Overhead Analysis. We analyze the runtime overhead introduced by each component of EASE under four workload patterns. As reported in Table 8, the total overhead accounts for less than 2% of the average latency, demonstrating that EASE incurs negligible runtime overhead compared to the overall execution time. Even under high burst workloads, where the scheduler needs to evaluate a large number of concurrent queries, the overhead remains low. This confirms that EASE’s scheduling is lightweight and does not introduce a performance bottleneck.

8 Related Work

Cloud resource management. Cloud providers employ various resource management techniques [23, 24, 45, 50] to manage physical infrastructures. These systems focus on maximizing hardware utilization through instance packing and scaling resources to meet service-level objectives. However, they operate at the provider level and expose resources as independent services with distinct pricing and performance characteristics, leaving the complexity of cross-service orchestration to the user. Unlike provider-side schedulers, EASE adopts a user-centric approach, managing the

Table 7. Frequency of query migrations.

Benchmark	High Burst	Mid Peak	Rising Load	Falling Load
ClickBench	4 (2.48%)	3 (1.12%)	1 (0.39%)	3 (0.66%)
SSB	19 (11.80%)	16 (5.99%)	16 (6.18%)	25 (5.53%)
TPC-H	19 (11.80%)	19 (7.12%)	14 (5.41%)	24 (5.31%)
JOB	6 (3.73%)	10 (3.75%)	7 (2.70%)	6 (1.33%)
Stack	8 (4.97%)	11 (4.12%)	9 (3.47%)	9 (1.99%)
Total	56 (6.96%)	59 (4.42%)	47 (3.63%)	67 (2.96%)

Table 8. Breakdown of per-query overhead introduced by different components in EASE.

Component	High Burst	Mid Peak	Rising Load	Falling Load
Cost model	0.0614s	0.0584s	0.0590s	0.0564s
Query scheduler	0.0432s	0.0284s	0.0349s	0.0267s
Dynamic rescheduler	0.0017s	0.0011s	0.0006s	0.0006s
Total overhead	0.1063s	0.0879s	0.0945s	0.0837s
Average latency	5.937s	4.427s	4.733s	4.649s

usage of heterogeneous cloud services rather than physical hardware, enabling users to optimize performance and cost across diverse cloud offerings.

Query execution in the cloud. Recent studies have explored leveraging specific cloud services to accelerate query processing. Some approaches utilize S3-Select to push down query execution to the storage layer [55, 57]. Others leverage serverless computing (FaaS) to achieve rapid scaling for bursty workloads [25, 40, 43, 44, 47]. While these works demonstrate the potential of cloud-native execution, they typically focus on optimizing for a single service or rely on simple rules for placement, without considering the distinct characteristics of heterogeneous cloud services. EASE addresses this limitation by explicitly modeling service heterogeneity, enabling more effective scheduling across diverse workloads.

Query scheduling. Existing query schedulers optimize for various objectives, such as fairness [28], latency [54], throughput [45], cost efficiency [37, 46, 56], low-level resource sharing [51], and buffer-awareness [31]. However, these systems assume homogeneous environments, limiting their applicability to the elastic and heterogeneous cloud. Approaches that require integration with the query engine [51] are incompatible with managed services like QaaS that do not expose internal interfaces. In contrast, EASE is designed for heterogeneous cloud services, leveraging their distinct characteristics to optimize query scheduling for both performance and cost.

9 Conclusion

We present EASE, a resource-aware scheduling framework for executing OLAP queries across heterogeneous cloud services. EASE integrates a heterogeneity-aware cost model, a real-time query scheduler, and a dynamic rescheduler to optimize query latency and execution cost. Extensive experiments show that EASE outperforms existing approaches under diverse workload patterns, demonstrating the effectiveness of heterogeneity-aware scheduling for cloud analytics.

Acknowledgments

This work was partially supported by National Key R&D Program of China (Grant No. 2023YFB4503600) NSF of China (Grant Nos. 62525202, 62232009, 62502520), and Scientific Research Innovation Capability Support Project for Young Faculty (Grant No. SRICSPYF-ZY2025001).

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. Amazon Redshift. <https://aws.amazon.com/redshift/>
- [8] 2013. Presto. <https://prestodb.io>
- [9] 2014. AWS Lambda. <https://aws.amazon.com/lambda/>
- [10] 2016. Amazon Athena. <https://aws.amazon.com/athena/>
- [11] 2016. Azure Functions. <https://azure.microsoft.com/products/functions>
- [12] 2017. Cloud Run functions. <https://cloud.google.com/functions>
- [13] 2019. DuckDB. <https://duckdb.org>
- [14] 2020. Azure Synapse Analytics. <https://azure.microsoft.com/en-us/products/synapse-analytics>
- [15] 2022. Amazon Redshift-Serverless. <https://docs.aws.amazon.com/redshift/latest/mgmt/working-with-serverless.html>
- [16] 2022. ClickBench. <https://github.com/ClickHouse/ClickBench>
- [17] 2022. Generating Dynamic Ads in Under 1 Second Using AWS Lambda with Booking.com. <https://aws.amazon.com/solutions/case-studies/booking-serverless-case-study/>
- [18] 2022. How to plan for peak demand on an AWS serverless digital-commerce platform. <https://aws.amazon.com/blogs/industries/how-to-plan-for-peak-demand-on-an-aws-serverless-digital-commerce-platform/>
- [19] 2022. SQLGlue: Python SQL Parser and Transpiler. <https://github.com/tobymao/sqlglue>
- [20] 2024. `cache_https`: A DuckDB extension for remote filesystem access cache. https://duckdb.org/community_extensions/extensions/cache_https
- [21] 2024. Optimize your workloads with Amazon Redshift Serverless AI-driven scaling and optimization. <https://aws.amazon.com/blogs/big-data/optimize-your-workloads-with-amazon-redshift-serverless-ai-driven-scaling-and-optimization/>
- [22] Mumtaz Ahmad, Songyun Duan, Ashraf Aboulmaga, 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
- [23] Nikos Armenatzoglou, Sanuj Basu, Naga Bhanoori, Mengchu Cai, Naresh Chainani, Kiran Chinta, Venkatraman Govindaraju, Todd J. Green, Monish Gupta, Sebastian Hillig, Eric Hotinger, Yan Leshinsky, Jintian Liang, Michael McCreedy, Fabian Nagel, Ippokratis Pandis, Panos Parchas, Rahul Pathak, Orestis Polychroniou, Foyzur Rahman, Gaurav Saxena, Gokul Soundararajan, Sriram Subramanian, and Doug Terry. 2022. Amazon Redshift Re-invented. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*, Zachary G. Ives, Angela Bonifati, and Amr El Abbadi (Eds.). ACM, 2205–2217. doi:10.1145/3514221.3526045
- [24] Bradley Barnhart, Marc Brooker, Daniil Chinenkov, Tony Hooper, Jihoun Im, Prakash Chandra Jha, Tim Kraska, Ashok Kurakula, Alexey Kuznetsov, Grant Mcalister, Arjun Muthukrishnan, Aravinthan Narayanan, Douglas Terry, Bhuvan Ugaonkar, and Jiaming Yan. 2024. Resource Management in Aurora Serverless. *Proc. VLDB Endow.* 17, 12 (2024), 4038–4050. doi:10.14778/3685800.3685825
- [25] 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
- [26] Jennie Duggan, Olga Papaemmanouil, Ugur Çetintemel, and Eli Upfal. 2014. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceedings of the 17th International Conference on Extending Database Technology, EDBT 2014, Athens, Greece, March 24-28, 2014*, Sihem Amer-Yahia, Vassilis Christophides, Anastasios Kementsietsidis, Minos N. Garofalakis, Stratos Idreos, and Vincent Leroy (Eds.). OpenProceedings.org, 109–120. doi:10.5441/002/EDBT.2014.11
- [27] Archana Ganapathi, Harumi A. Kuno, Umeshwar Dayal, Janet L. Wiener, Armando Fox, Michael I. Jordan, and David A. Patterson. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng (Eds.). IEEE Computer Society, 592–603. doi:10.1109/ICDE.2009.130
- [28] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. 2011. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*, David G. Andersen and Sylvia Ratnasamy (Eds.). USENIX Association. <https://www.usenix.org/conference/nsdi11/dominant-resource->

fairness-fair-allocation-multiple-resource-types

- [29] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. In *ACM SIGCOMM 2014 Conference, SIGCOMM'14, Chicago, IL, USA, August 17-22, 2014*, Fabián E. Bustamante, Y. Charlie Hu, Arvind Krishnamurthy, and Sylvia Ratnasamy (Eds.). ACM, 455–466. doi:10.1145/2619239.2626334
- [30] 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
- [31] 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>
- [32] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. doi:10.14778/2850583.2850594
- [33] 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
- [34] Zibo Liang, Xu Chen, Yuyang Xia, Runfan Ye, Haitian Chen, Jiandong Xie, and Kai Zheng. 2024. DACE: A Database-Agnostic Cost Estimator. In *40th IEEE International Conference on Data Engineering, ICDE 2024, Utrecht, The Netherlands, May 13-16, 2024*. IEEE, 4925–4937. doi:10.1109/ICDE60146.2024.00374
- [35] 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>
- [36] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Nesime Tatbul, Mohammad Alizadeh, and Tim Kraska. 2021. Bao: Making Learned Query Optimization Practical. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhui Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1275–1288. doi:10.1145/3448016.3452838
- [37] 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
- [38] 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
- [39] Michail Georgoulakis Misegiannis, Daniel Ritter, Viktor Leis, and Jana Giceva. 2025. CloudGlide: Deconstructing the Landscape of Cloud-Based Analytics. *Proc. VLDB Endow.* 18, 13 (2025), 5638–5651. doi:10.14778/3749646.3749727
- [40] Ingo Müller, Renato Marroquin, 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
- [41] 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
- [42] 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
- [43] 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
- [44] 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
- [45] 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

- [46] 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
- [47] Yongye Su, Yinqi Sun, Minjia Zhang, and Jianguo Wang. 2024. Vexless: A Serverless Vector Data Management System Using Cloud Functions. *Proc. ACM Manag. Data* 2, 3 (2024), 187. doi:10.1145/3654990
- [48] 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
- [49] Midhul Vuppapalapati, 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/vuppapalapati>
- [50] Vishal Vyas, Andrei Paduoiu, Srikanth Kandula, Hari Ohm Prasath Rajagopal, Mukesh Punhani, Marco Manzo, Ankur Goyal, Santosh Chandrachood, Rick Sears, Joseph Marques, and Sushant Majithia. 2025. Managed Resource Scaling in Amazon EMR. In *Companion of the 2025 International Conference on Management of Data, SIGMOD/PODS 2025, Berlin, Germany, June 22–27, 2025*, Volker Markl, Joseph M. Hellerstein, and Azza Abouzied (Eds.). ACM, 662–674. doi:10.1145/3722212.3724443
- [51] 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
- [52] 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
- [53] Yang Wu, Xuanhe Zhou, Xiaoguang Li, Jinhuai Kang, Chunxiao Xing, Tongliang Li, Xinjun Yang, Wenchao Zhou, Feifei Li, and Yong Zhang. 2025. MemQ: A Graph-Based Query Memory Prediction Framework for Effective Workload Scheduling. In *41st IEEE International Conference on Data Engineering, ICDE 2025, Hong Kong, May 19–23, 2025*. IEEE, 3876–3889. doi:10.1109/ICDE65448.2025.00289
- [54] Ziniu Wu, Markos Markakis, Chunwei Liu, Peter Baile Chen, Balakrishnan Narayanaswamy, Tim Kraska, and Samuel Madden. 2025. Improving DBMS Scheduling Decisions with Accurate Performance Prediction on Concurrent Queries. *Proc. VLDB Endow.* 18, 11 (2025), 4185–4198. <https://www.vldb.org/pvldb/vol18/p4185-wu.pdf>
- [55] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: rethinking computation pushdown for cloud OLAP DBMSs. *VLDB J.* 33, 5 (2024), 1643–1670. doi:10.1007/S00778-024-00867-8
- [56] 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
- [57] Xiangyao Yu, Matt Youill, Matthew E. Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *36th IEEE International Conference on Data Engineering, ICDE 2020, Dallas, TX, USA, April 20–24, 2020*. IEEE, 1802–1805. doi:10.1109/ICDE48307.2020.00174
- [58] Yue Zhao, Gao Cong, Jiachen Shi, and Chunyan Miao. 2022. QueryFormer: A Tree Transformer Model for Query Plan Representation. *Proc. VLDB Endow.* 15, 8 (2022), 1658–1670. doi:10.14778/3529337.3529349

Received 17 January 2026; revised 19 May 2026; accepted 12 June 2026