Riveter: Adaptive Query Suspension and Resumption Framework for Cloud Native Databases

Rui Liu
University of Chicago
ruiliu@cs.uchicago.edu

Aaron J. Elmore
University of Chicago
aelmore@cs.uchicago.edu

Michael J. Franklin
University of Chicago
mjfranklin@uchicago.edu

Sanjay Krishnan
University of Chicago
skr@cs.uchicago.edu

2024 IEEE 40th International Conference on Data Engineering (ICDE) | 979-8-3503-1715-2/24/$31.00 ©2024 IEEE | DOI: 10.1109/ICDE60146.2024.00304

***Abstract*—In modern cloud environments, ephemeral resources with intermittent availability and fluctuating monetary costs are becoming common. This dynamic nature presents a new challenge when deploying cloud-native databases: adaptive query execution, which can suspend queries when the resources are scarce or costs unexpectedly soar, and then resume them when the resources become available or cost-effective. Addressing this challenge requires the design and implementation of query suspension and resumption with a mechanism that can adaptively determine when, if, and how to suspend queries. In this paper, we propose Riveter, a query suspension and resumption framework that can adaptively pause ongoing queries using various strategies, including (1) a redo strategy that terminates queries and subsequently re-runs them, (2) a pipeline-level strategy that suspends a query once one of its pipelines has completed to reduce the storage requirements for intermediate data, (3) and a process-level strategy that enables the suspension of query execution processes at any given moment but generates a substantial volume of intermediate data for query resumption. We also devise a cost model to estimate query latency using various strategies and an algorithm to select the one that causes minimum latency. To demonstrate the effectiveness of Riveter, we conduct evaluations based on the TPC-H benchmark to investigate intermediate data persistence, strategy selection, and cost model-based estimation. Our results not only present the difference among the strategies of Riveter in terms of the size of persisted intermediate data and the time of triggering the suspension but also confirm the adaptive and efficient query suspension and resumption delivered by Riveter.**

***Index Terms*—cloud-native, query execution, query optimization, cost-model, suspension, resumption, resource-adaptive**

I. Introduction

With the recent advancement in modern hardware and virtualization, an increasing number of data processing and analytic workloads are shifting towards database systems deployed on large-scale cloud infrastructure [1]. This shift is giving rise to a new database paradigm: cloud-native databases [2], which are designed and optimized to run seamlessly within cloud environments, and there have been a number of successful instances of cloud-native databases. Nevertheless, the increasing prevalence of ephemeral cloud resources is prompting a re-thinking of the principles of cloud-native databases and necessitating a novel query execution. *First, ephemeral cloud resources are ever-changing in availability*. Spot instances provide short-lived computing infrastructure for short-running jobs [3], [4], [5], [6]. Recent developments in cloud computing are amplifying this transient capacity. One noteworthy example is the rise of serverless computing [7], [8], which empowers applications to leverage lightweight cloud resources characterized by constrained runtime, memory capacity, and computation specification. Another emerging trend is "zero-carbon clouds" [9]. In this cloud paradigm, data centers are designed to be entirely ephemeral, driven primarily by renewable energy sources, such as sunlight and wind, which are inherently unstable in availability. *Second, the monetary cost of cloud resources can fluctuate.* The inherent multi-tenancy nature of cloud resources [10] inevitably leads to price adjustments based on resource supply and demand dynamics [11]. Reportedly, the prices of cloud resources can surge to 200 to 400 times the normal rate during peak demand [7]. Moreover, the recent rise of data science and data-driven AI has introduced increasingly complex and heterogeneous workloads, combining both long-running and short-running queries [12], [13], [14]. This diversity can potentially lead to resource saturation, thereby exacerbating pricing fluctuations.

These two emerging trends challenge cloud-native databases from two perspectives: (1) the assumption that cloud resources are stable is not always applicable, and thus preserving the resources for a relatively long term is not feasible all the time; (2) the widely-used latency-oriented SLA (Service Level Agreement) may not be the best option for all users, particularly for those who prioritize cost-efficiency over speed. An ideal cloud-native database designed for ephemeral resources needs to perform queries in an adaptive manner: *suspending a query when resource accessibility or cost-efficiency is no longer viable and resuming it when resources become available or cost-effective again while minimizing the overhead caused by the query suspension and resumption.*

Motivated by this novel requirement, we proposed *Riveter*, an adaptive query suspension and resumption framework. Riveter supports various query suspension and resumption strategies, including redo, process-level, and pipeline-level strategies. Specifically, the redo strategy allows for terminating a query at any given moment and subsequently rerunning it. However, with this strategy, all current query progress is forfeited upon termination. In contrast, the process-level strategy can also suspend a query at any point and preserve the current progress by persisting all intermediate data and context information of the process in which the query is executed. This strategy typically requires that each query runs in an isolated process; otherwise, the strategy may pause the entire database system. The process-level strategy comes with notable drawbacks: the

2375-026X/24/$31.00 ©2024 IEEE
DOI 10.1109/ICDE60146.2024.00304

persisted data can be exceedingly large, and the requirement for resuming processes requires identical resource configurations, such as the number of hardware threads and allocated memory size, as were in use at the time of suspension.

The pipeline-level strategy offers an alternative approach by persisting the intermediate data of each pipeline in the query plan for potential resumption. This implies that suspending queries and persisting their intermediate data only occur after specific pipelines have completed their execution, but this strategy usually reduces the volume of persisted intermediate data since they are typically aggregated upon pipeline completion. Riveter also employs a cost model that can estimate the latency of various strategies and adaptively determine if, when, and which strategy should be used.

For the implementation of Riveter, we modify DuckDB [15] to realize the pipeline-level strategy and develop the process-level strategy building on top of CRIU [16]. We also devise an adaptive query suspension and resumption algorithm based on the proposed cost model. We further conduct a set of evaluations using the TPC-H benchmark [17] to study the effectiveness of Riveter. We first compare intermediate data persistence when the suspension starts under process-level and pipeline-level strategy. For instance, using the TPC-H SF-100 dataset, when query Q21 is suspended around halfway using the process-level strategy, the intermediate data size is 28GB. Conversely, with a pipeline-level strategy in the same setting, suspension may be postponed until the completion of the current pipeline, significantly reducing the intermediate data size to 79 bytes. We also analyze how Riveter determines the strategy with the least latency for specific queries and reveal the selected strategy introduces negligible overhead. We validate the efficiency of our cost model in terms of estimation accuracy and runtime.

Contributions. This work aims to design and implement an adaptive query execution framework for ephemeral cloud resources. This framework not only provides a cost model and query suspension/resumption algorithm, but also introduces the mechanism of selecting and executing diverse query suspension and resumption strategies that are developed within Riveter. To the best of our knowledge, Riveter is the first work to provide a pipeline-level suspension and resumption strategy. These advancements position our framework as a critical extension to the query optimization capabilities of cloud-native databases, thereby enhancing their efficiency and adaptability. Our contributions are summarized:

- Characterizing the adaptive query suspension and resumption for cloud-native databases with ephemeral resources (§II).
- Designing and implementing various query suspension and resumption strategies within the proposed adaptive query execution framework Riveter. (§III-B, §III-A).
- Devising a cost model and an algorithm to adaptively determine the suspension and resumption strategies for various queries (§III-C).
- Evaluating Riveter using the TPC-H benchmark and revealing insights: strategies bring distinct overheads due to intermediate data persistence (§IV-A), adaptively selecting strategies is necessary as their performance varies markedly across different scenarios (§IV-B), and the proposed cost model-based estimation is vital for informed strategy selection but introduces neglect overhead (§IV-C).

II. Motivation

We revisit the original concept of query suspension and resumption and further highlight the essential role and potential benefits of adaptive query suspension and resumption, particularly in cloud-native databases with ephemeral resources.

A. Query Suspension and Resumption

The concept of query suspension and resumption is rooted in recovery mechanisms in database systems [18], [19], and the work can be considered one of the earliest works of query suspend and resume is database checkpoint mechanism. It can create a point from which the execution engine can persist the current state of the database into non-volatile storage [20]. In light of the checkpoint mechanisms, a query suspend and resume approach is proposed for pull-based query execution [21]. It generates a query suspend plan that may involve a combination of persisting current state and going back to previous checkpoints. Within this approach, the potential suspension points are always the ones that have minimized memory usage, thus mitigating significant overhead during suspension and resumption. Alternatively, instead of persisting the intermediate data, certain systems suspend queries and retain the intermediate data in memory during suspension. For instance, Amber [22] converts the operator DAG of the execution plan to an actor DAG and passes messages of "suspend" and "resume" among actors so that the query suspension or resumption can be triggered. Nevertheless, maintaining query processing in a suspended state without persistence can consume substantial memory resources and lead to significant delays for other queries. Moreover, this approach is not suitable for query or database migration, which is not uncommon in cloud-native databases.

We identify three pivotal aspects of suspension and resumption strategies: (1) suspension flexibility, (2) intermediate data persistence, and (3) progress preservation. Based on the perspectives, the most ideal strategy is the one that can suspend a query at any given time and preserve the progress of the suspending time point with the least volume of intermediate data to persist. Suspension flexibility measures the granularity of triggering suspension a strategy can support. Within this spectrum, two extremes can be identified: one where queries can be suspended at any point during their executions and another where suspension occurs only upon query completion. Progress preservation signifies how much query processing progress can be retained, whether partially or in its entirety, when a suspension is initiated. Intermediate data persistence specifically refers to the size of persisted intermediate data, which directly impacts the latency of query suspension.

We describe three strategies according to the above perspectives, as presented in Table I.

	Suspension Flexibility	*Data Persistence*	*Progress Preservation*
Redo Strategy	Terminate at anytime	No intermediate data	Loses all progress
Process-Level Strategy	Suspend anytime at process level	Intermediate data of process	Preserves all progress
Pipeline-Level Strategy	Suspend at pipeline completion	Intermediate data of pipeline	Keeps progress of persisted pipeline

TABLE I: Representative suspension & resumption strategies

Redo strategy allows for query suspension at any given time through immediate termination. It preserves no progress of query processing, resulting in no persisted intermediate data.

Process-level strategy can suspend a query at any given time and keep the current progress during suspension by persisting all necessary intermediate data.

Pipeline-level strategy only checks if a query can be suspended when a pipeline is completed and preserves progress by persisting the intermediate data at this completion point.

B. Motivational Cases

Adaptive query suspension and resumption become evident in the evolving scenarios of cloud-native databases, considering that ephemeral cloud resources are becoming increasingly prevalent. The rise of spot instances [3], [4], [5] and the emergence of zero-carbon clouds [9], [23] have provided transient and intermittent computing infrastructures. Meanwhile, the prices of these cloud resources can fluctuate. In these scenarios, the adaptive query suspension and resumption enable queries to execute when resources are available or costs are within acceptable bounds, and it allows query suspension when resources become unavailable or costs become prohibitive.

We describe the following motivational cases in cloud-native databases for adaptive query suspension and resumption:

Case 1: Heterogeneous workloads. Cloud native databases are deployed for multi-tenants and process the heterogeneous workloads that are mixed by short-running and long-running queries. When long-running queries occupy resources for unexpectedly long periods, they potentially cause resource saturation. This can lead to significant delays for shorter-running queries, which otherwise could have been completed promptly with sufficient resources. Existing methods, such as query scheduling or resource reservation, may fall short of expectations as they often treat the queries as indivisible entities, whereas the proposed adaptive query suspension and resumption essentially converts a long-running query into a series of short-running ones by suspending and resuming it for multiple times, allowing more flexible scheduling for execution.

Case 2: Database migration. Database migration is vital for maintaining elasticity and scalability in cloud-native databases. The current state-of-the-art approach is live database migration, which strives to move database services with minimal service disruption and negligible impact on performance. However, migrating entire database systems remains non-trivial and brings substantial overhead. The adaptive query suspension and resumption framework can enable the migration of individual queries, potentially reducing the overhead associated with migrating cloud-native databases. Furthermore, in certain circumstances where a single resource-intensive, long-running query dominates the database, migrating that specific query can free up resources for more efficient utilization.

Case 3: Computation with ephemeral resources. An increasing number of cloud-native databases now support serverless computing, which leverages lightweight and short-lived resources to enable users to invoke functions deployed in the cloud and retrieve results locally. These functions can be likened to queries or analytical tasks performed on specific datasets. However, ephemeral resources can lead to increased latency in serverless computing due to fluctuating resource costs and unstable resource availability. The proposed adaptive query suspension and resumption can mitigate these issues by avoiding periods when resources are either unavailable or expensive for utilization.

III. Riveter Design and Implementation

Inspired by the motivational cases, we propose Riveter, an adaptive query suspension and resumption framework. Riveter supports three strategies as we identify in §II-A and adaptively select the strategy that is associated with minimized latency based on a cost model.

A. Process-level Suspension and Resumption

Riveter can suspend and resume queries at any given point by employing a process-level query suspension and resumption strategy. This strategy operates within the context of a process, ensuring the persistence of the complete context and all intermediate data upon initiation of suspension. Riveter reconstructs the entire process context and execution environment during the resumption phase, subsequently reloading the intermediate data associated with that process. The process-level strategy in Riveter empowers the suspension and resumption of queries at will, albeit accompanied by a notable overhead due to the necessity of data persistence and loading. This overhead arises from the persistence of the entire process context and state during the suspension phase, which is then reinstated upon query resumption. Furthermore, the resumption of queries triggers the meticulous reconstruction of the process context to precisely match its state during suspension, which also ensures that the resource configuration remains consistent and unaltered between the suspension and resumption phases.

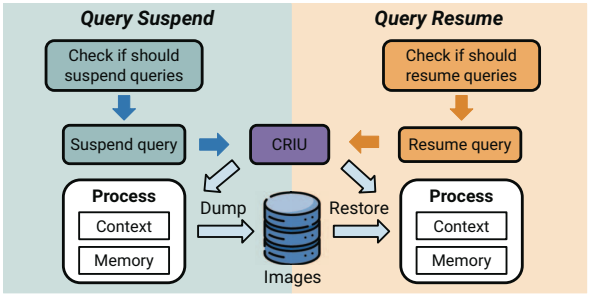

Fig. 1: Workflow of process-level strategy.

We implement process-level query suspension and resumption in Riveter on top of CRIU [16], as demonstrated in Fig. 1. When suspending a query, the entire query execution process will be dumped as intermediate image files, which can be used to restore the process and the associated query.

B. Pipeline-level Suspension and Resumption

Pipeline-based query execution divides the query plan into a number of pipelines for parallel execution. Dividing query plans into pipelines is based on the pipeline breakers that usually block operators and collect sufficient intermediate results for further process. Thus, the pipeline-level suspension and resumption strategy in Riveter takes the pipeline breakers as the natural suspension and resumption points.

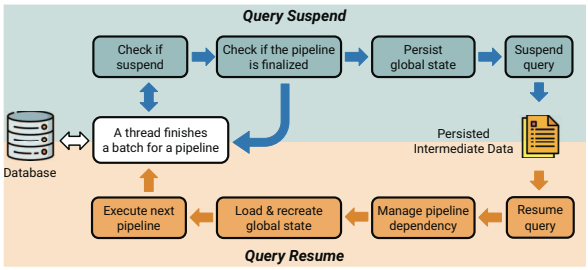

Fig. 2: Workflow of pipeline-level strategy.

We modify DuckDB [15] to implement a prototype system for the pipeline-level strategy. The workflow of pipeline-level suspension and resumption is illustrated in Fig. 2. When a thread completes processing a pipeline with a data batch, Riveter checks whether the query execution has reached the suspension point and determines whether to initiate the suspension process. If affirmative, Riveter proceeds to verify if the current pipeline is finalized—indicating that all the intermediate results at the thread level have been merged into the global state. If so, Riveter serializes and persists the global state before suspending the query. When resuming a query, Riveter first attempts to identify the persisted intermediate data and resource configurations in order to re-establish the query execution environment. Subsequently, Riveter manages pipeline dependencies; for example, it bypasses pipelines processed prior to suspension and marks successor pipelines as ready for execution while also reconstructing the global states for future execution. Afterward, Riveter carries out the query until it reaches the next suspension point or produces the final results.

We illustrate a set of common query operators under pipeline-level strategy in Fig. 3 and Fig. 4. Although a pipeline can be executed by multiple threads, it maintains a global state to merge the intermediate data of each thread. Taking in-memory hash join as an example, the query plan is split into two pipelines: one is for building hash tables, and the other is for probing hash tables. Thus, there are two pipeline breakers at the end of each pipeline for suspension and resumption; each breaker maintains a global state.

The implementation of pipeline-level strategy only performs at pipeline breakers for query suspension and resumption, but the intermediate data are usually processed and aggregated, which makes the necessary persisted data reasonably small. Our implementation brings an extra advantage, running queries using adaptive resources. This is because Riveter can check and exploit the available resources when loading and recreating the execution context for resume queries.

C. Cost Model and Strategy Selection

We devise a cost model to adaptively decide if, when, and how to suspend queries, formulating the cost model as follows.

Given. A query q that may encounter a termination within a specific time window T, starting from T_s to T_e. Considering the potential termination, the query must confront a decision point: it can either be forcefully terminated, resulting in the loss of current progress, or strategically suspended while retaining all intermediate data. When the query is suspended before the termination, the intermediate data is persisted, ensuring the continuity of the ongoing progress. However, persisting intermediate data at query suspension and reloading them during query resumption may introduce additional latency, denoted as L_s and L_r, which are typically denominated by the size of intermediate data. In cases where a query cannot be suspended before its intended termination, both the intermediate data and the current progress are forfeited, which necessitates a complete re-execution from its starting point.

Assumption. A termination can happen within a time window T, determined by a probability $P_T, 0 \leq P_T \leq 1$. Specifically, P_T can be either predefined or decided by a probability distribution π. This setting can simulate the termination in real-world applications; for instance, users can be alerted when their spot instances are at risk of imminent termination, or servers in zero-carbon clouds may face an energy shortage within a future time window. The potential termination point can differ each time the query is executed. Additionally, the available amount of resources, such as CPU threads and memory, are known prior to the suspension and resumption of the query, but the availability of these resources might undergo changes during the suspension and resumption phases; thus, resuming a query is unviable when the resources available are inadequate for its execution.

Objective. Select the strategy to minimize the overall cost for q. When suspending q using the redo strategy, the cost is,

$$Cost_q^{redo} = P_T^{redo} * \Gamma^{redo} \tag{1}$$

When executing q using the process-level strategy, the cost is,

$$Cost_q^{proc} = L_s^{proc} + L_r^{proc} + P_T^{proc} * \Gamma^{proc} \tag{2}$$

When executing q using the pipeline-level strategy, the cost is,

$$Cost_q^{ppl} = L_s^{ppl} + L_r^{ppl} + P_T^{ppl} * \Gamma^{ppl} \tag{3}$$

Specifically, $Cost_q^{redo}$ is essentially the execution time before the termination point, which is jointly determined by the probability of termination P_T^{redo} and the redo cost Γ^{redo} under the redo strategy. The costs associated with the pipeline-level and the process-level strategy come from two perspectives:

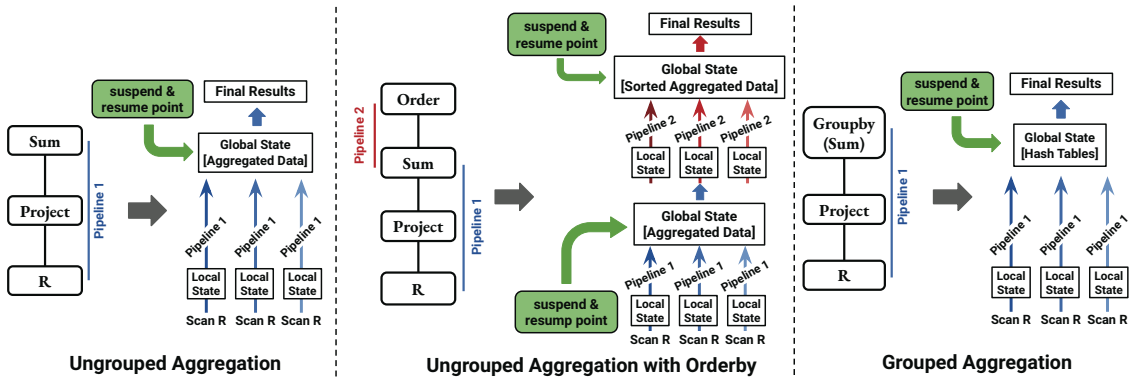

Fig. 3: Pipeline-level suspension and resumption strategy for aggregations in Riveter.

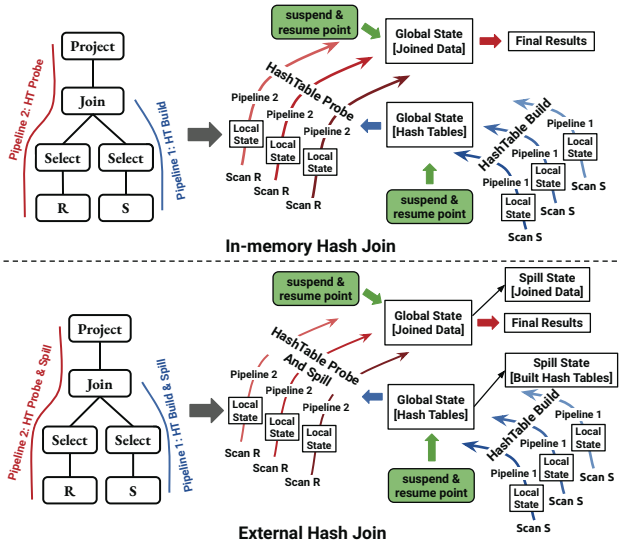

Fig. 4: Pipeline-level suspension and resumption strategy for hash-joins in Riveter.

(1) the latency caused by persisting intermediate data during suspension and reloading them for query resumption, and (2) the redo cost if the suspension fails to complete before the termination point. Therefore, the cost of pipeline-level and process-level strategies can be expressed as $Cost_q = (L_s + L_r) + P_T * \Gamma$. The latency L_s and L_r can be estimated based on the size of intermediate data and hardware specification to facilitate prompt decision-making during query execution.

Using the cost model, we devise an algorithm for adaptive strategy selection. Since continuously monitoring the status of query execution and performing cost calculations for potential suspension and resumption can significantly impede regular query execution, our algorithm employs a proactive approach to support adaptive query execution and meanwhile reduce the associated overhead. We present an illustrative example in Fig. 5, and more details are provided in Algorithm 1.

Riveter estimates the cost of employing the three strategies and makes latency-oriented decisions when the query execution reaches a pipeline breaker. For the query q, the cost of redo

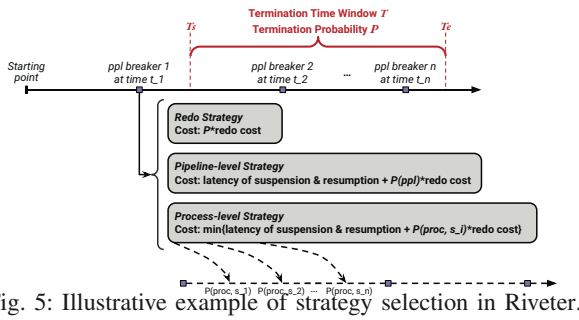

Fig. 5: Illustrative example of strategy selection in Riveter.

strategy $Cost_q^{redo}$ is expressed as $P_T^{redo} * \Gamma^{redo}$, where P_T^{redo} is calculated in terms of the overlap between termination time window T and the time of completing future pipelines (lines 9-17 in Algorithm 1). The completion time of future pipelines can be estimated based on the average of previous pipelines. For the cost estimation of pipeline-level strategy, we serialize and persist the intermediate data in binary format, essentially representing a large vector of bytes, which allows us to determine its size. Then, the latency L_s^{ppl} and L_r^{ppl} can be estimated based on the size of intermediate data and the estimated I/O performance. The cost associated with the pipeline-level strategy, denoted as $Cost_q^{ppl}$, is calculated by aggregating the above factors, as presented from line 33 to 46.

The cost estimation of process-level strategy requires probing more future suspension points since the strategy can suspend queries anytime (lines 18-32). The suspension point under a process-level strategy associated with minimum latency can be selected. During the probing process, Riveter employs an iterative approach, advancing suspension time points by each time unit that can be predefined according to the termination time window to estimate the latency of each future suspension point. It is non-trivial to estimate the latency L_s^{proc} and L_r^{proc} of process-level strategy at each potential suspension point since the intermediate data include the processing data and all the necessary context information as well. We exploit a regression-based approach to estimate the size of intermediate data under the process-level strategy. The regression-based approach can fit a curve based on a number of key factors, such as the size

Algorithm 1: Adaptive Strategy Selection

Input : Query q, Termination time window T (from T_s to T_e), Probability of termination P_T

$T_{sum} = 0$, $N_{ppl} = 0$;
Initialize termination probabilities $P_T^{redo}, P_T^{ppl}, P_T^{ppl}$;
while *q reaches a pipeline breaker* **do**
  Observe current time C_t;
  Observe available memory M;
  Observe running time of pipeline T_{ppl};
  $T_{sum} = T_{sum} + T_{ppl}$, $N_{ppl} = N_{ppl} + 1$;
  Return strategy based on $min\{\text{CostEstRedo}(), \text{CostEstPpl}(), \text{CostEstProc}()\}$
Function CostEstRedo():
  if $C_t \geq T_s$ *or* $C_t + \frac{T_{sum}}{N_{ppl}} \geq T_e$ **then**
    $P_T^{redo} = P_T$;
  else if $T_s \leq C_t + \frac{T_{sum}}{N_{ppl}} < T_e$ **then**
    Overlapped time window $T_o = C_t + \frac{T_{sum}}{N_{ppl}} - T_s$;
    $P_T^{redo} = \frac{T_o}{T_e - T_s} * P_T$;
  else
    $P_T^{redo} = 0$;
  Return $Cost_q^{redo} = P_T^{redo} * C_t$;
Function CostEstProc():
  for st_i, $i \leftarrow C_t$ **to** $C_t + \frac{T_{sum}}{N_{ppl}}$ **do**
    Estimate size of intermediate data $S_{st_i}^{proc}$ at st_i;
    if $S_{st_i}^{proc} \leq M$ **then**
      Estimate latency L_{s,st_i}^{proc} and L_{r,st_i}^{proc} based on $S_{st_i}^{proc}$;
    else
      $L_{s,st_i}^{proc} = \infty$, $L_{r,st_i}^{proc} = \infty$;
    if $st_i + L_{s,st_i}^{proc} \geq T_e$ **then**
      $P_T^{proc} = P_T$;
    else if $T_s \leq st_i + L_{s,st_i}^{proc} < T_e$ **then**
      Overlapped time window $T_o = st_i + L_{s,st_i}^{ppl} - T_s$;
      $P_T^{proc} = \frac{T_o}{T_e - T_s} * P_T$;
    else
      $P_T^{proc} = 0$;
    Return $Cost_q^{proc} = \min\{L_{s,st_i}^{proc} + L_{r,st_i}^{proc} + P_T^{proc} * st_i\}$
Function CostEstPpl():
  Obtain size of intermediate data S^{ppl} of pipeline-level strategy;
  if $S^{ppl} \leq M$ **then**
    Estimate latency L_s^{ppl} and L_r^{ppl} based on S^{ppl};
  else
    $L_s^{ppl} = \infty$, $L_r^{ppl} = \infty$;
  if $C_t + L_s^{ppl} \geq T_e$ **then**
    $P_T^{ppl} = P_T$;
  else if $T_s \leq C_t + L_s^{ppl} < T_e$ **then**
    Overlapped time window $T_o = C_t + L_s^{ppl} - T_s$;
    $P_T^{ppl} = \frac{T_o}{T_e - T_s} * P_T$;
  else
    $P_T^{ppl} = 0$;
  Return $Cost_q^{ppl} = L_s^{ppl} + L_r^{ppl} + P_T^{ppl} * C_t$;

and the cardinality of the input data, the metadata of the query (e.g., number of various core operators in the physical query plan), and the suspension point. In cases where regression-based estimation methods may not yield satisfactory results due to limited or unavailable historical data, we also resort to an optimizer-based approach for robust estimation. Specifically, the optimizer-based estimation of the persisted intermediate data size is an outcome of two factors: (1) memory utilization, which is determined through query plans generated by cost-based optimizers [24]. It takes into account the estimated cardinality of the core operator closest to the root node in the query plan, as well as the data types associated with each column in the input data; (2) the ratio of suspension time point, which is calculated by comparing the suspension time to the overall execution time. For example, we may choose to suspend queries at around 50% of their total execution time as a reference point for this ratio. Note that the aforementioned estimation approaches can be revised to fit different scenarios.

IV. Evaluation

Using the TPC-H benchmark, we:

- Investigate the intermediate data persistence under process-level and pipeline-level strategy (§IV-A),
- Analyze the strategy selection based on the cost model (§IV-B),
- Study the estimation accuracy and runtime of the cost model (§IV-C).

Riveter is evaluated on a server with two Intel Xeon Silver CPUs (2.10GHz, 16.5MB Cache, 12 physical cores), 192GB memory, and six 10TB hard disks (7200 RPM, SATA 6.0 Gb/s), running Ubuntu Server 18.04. We use the Parquet format [25] and assume that data is ingested for query processing. During the evaluation, query executions are initiated using Python APIs. All the results in the evaluation are averaged over three independent runs, unless stated otherwise.

We run all the queries from the TPC-H benchmark to conduct a comprehensive evaluation. Additionally, we highlight the queries Q1, Q3, Q17, and Q21, which feature diverse core operators and varying numbers of input tables. The queries are characterized in Table II.

Query	Core Operators	Tables
Q1	1 groupby	1 table
Q3	1 groupby, 2 join	3 tables
Q17	1 join, 1 unionall	2 tables
Q21	1 groupby, 4 join, 1 outer join	4 tables

TABLE II: Selected queries in TPC-H

A. Impact of Intermediate Data Persistence

We investigate the size of persisted intermediate data using the strategies in Riveter. Since the redo strategy allows query termination to occur and re-executes them subsequently without persisting any intermediate data, this evaluation focuses on the process-level and pipeline-level strategy.

*1) **Process-level Strategy**:* We utilize the process-level strategy in Riveter to suspend all the queries from TPC-H benchmark. Specifically, Riveter suspends the queries at approximately 50% of their estimated execution time, persisting all intermediate data and process context. As depicted in Fig. 6, the sizes of intermediate data for most queries exhibit a proportional increase with the volume of input datasets. For example, when Q21 is suspended, the size of persisted data is 3.1GB for the SF-10 dataset, 14GB for the SF-50 dataset, and

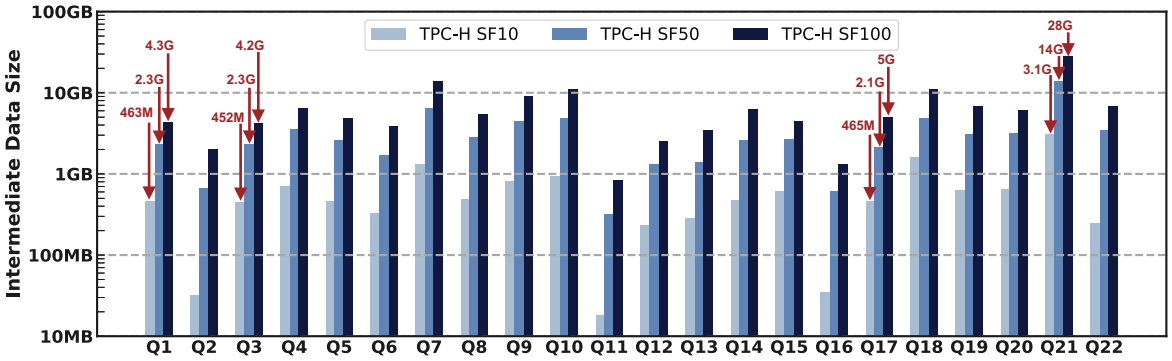

Fig. 6: Persisted intermediate data size of queries in TPC-H when suspending them at around 50% execution time using the process-level strategy on SF-10, SF-50, SF-100 datasets.

28GB for the SF-100 dataset. However, there were exceptions noted for queries Q2, Q11, Q16, and Q22 when using the SF-10 dataset. This deviation can be attributed to the lightweight nature of these four queries, which were completed rapidly when processing smaller datasets such as SF-10. For instance, the execution time of Q2 and Q11 are 0.9 and 0.4 seconds, respectively. Consequently, when Riveter suspends these queries at approximately 50% of their execution time (i.e., around 0.45 and 0.2 seconds, respectively), they were still in the early stages of execution, resulting in small sizes of intermediate data for persistence.

Following the above argument that queries in their early execution stages generate fewer intermediate data under the process-level strategy, we further investigate how different suspension points affect the size of intermediate data. Specifically, we measure the intermediate data size when suspending queries at approximately 30%, 60%, and 90% of their execution progress. Fig. 7 presents that the size of persisted intermediate data under the process-level strategy increases as suspension occurs later in the query execution. Our argument for this trend is that the memory allocation is not timely de-allocated during query execution. Thus, intermediate data accumulates until the query execution is completed. This observation underscores the trade-off between suspending a query immediately to reduce the space required for persistence and delaying suspension to make more progress, should it prove advantageous.

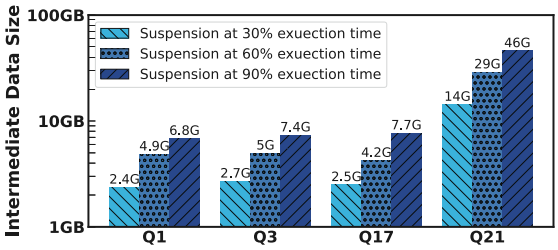

Fig. 7: Persisted intermediate data size of Q1, Q3, Q17, Q21 when suspending them at around 30%, 60%, and 90% execution time using process-level strategy on SF-100 dataset.

*2) **Pipeline-level Strategy**:* Riveter suspends queries from TPC-H benchmark at around 50% of their execution time using pipeline-level strategy. Fig. 8 illustrates the size of persisted intermediate data of 22 queries, each exhibiting notable differences. For example, when suspending Q1 on the SF-10 dataset, the size of intermediate data is less than 1KB, while the intermediate data generated by Q8 for the same dataset is approximately 12GB. Furthermore, while the intermediate data for certain queries (e.g., Q8, Q9, Q12, Q17, and Q18) that undergo suspension tends to grow proportionally with the size of the input datasets (SF-10, SF-50, SF-100), the size of intermediate data for other queries, such as Q1, Q4-7, Q11, Q14, and Q19-Q22, remains consistent across these datasets. This behavior is primarily influenced by the specific pipeline in which the query is situated when suspension is initiated. For instance, queries within hash-join pipelines tend to exhibit increased intermediate data sizes in tandem with larger input datasets, whereas queries within aggregation pipelines, such as those involving sum or average operators, tend to produce intermediate data of similar sizes due to the nature of their aggregated results. Moreover, in the case of certain queries, the pipeline-level strategy results in a greater volume of intermediate data compared to the process-level strategy. This discrepancy arises from the fact that the pipeline-level strategy exclusively initiates suspension upon the completion of a pipeline. To illustrate, consider Q8: when the suspension is requested at around 50% of its execution time, the pipeline engages with a hash-join operator is being processed; thus, the pipeline-level strategy can only suspend Q8 when this pipeline is processed completely, thereby necessitating the retention of the entire hash table for the join. However, the process-level strategy exhibits a swifter response to suspension requests, resulting in the persistence of intermediate data involving only the partial hash table.

Since the pipeline-level strategy only suspends a query when one of its pipelines is finalized, the time a suspension is requested is not always the moment when the suspension process commences, and we measure the time difference accordingly in Fig. 9. Specifically, Q21 has the minimum delay on different datasets due to the granularity of its query plan, which can be quantified by the number of pipelines involved. This is mainly because Q21 has more pipelines than

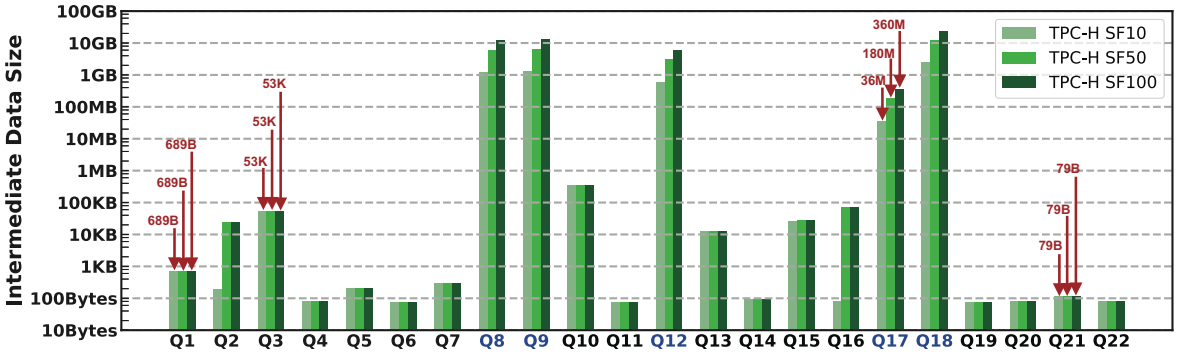

Fig. 8: Persisted intermediate data size of queries in TPC-H when suspending them at around 50% execution time using the pipeline-level strategy. The bars marked with blue labels (i.e., Q8, Q9, Q12, Q17, and Q18) represent queries that generate larger intermediate data because they are suspended upon completion of the pipeline ending with join-like operators.

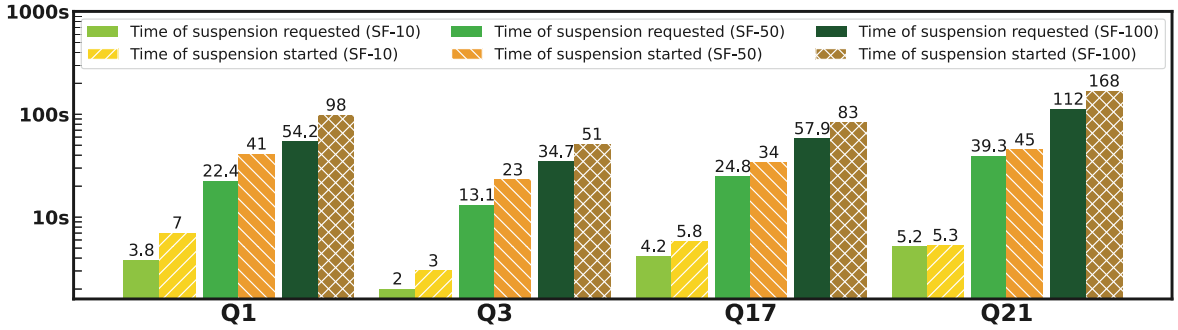

Fig. 9: Time lag incurred when suspensions are requested under the pipeline-level strategy.

Q1, Q3, and Q17, which provides more feasible suspension points and thereby allows the query to be suspended closer to the suspension point. Our results quantitatively confirm that when suspension occurs, the pipeline-level strategy typically generates significantly less intermediate data for persistence than the process-level strategy. This reduced intermediate data can be attributed to the fact that pipeline-level strategy does not necessitate capturing all state and context data for resumption. However, the pipeline-level strategy may introduce extra delays in responding to suspension requests, particularly when the query plan comprises fewer pipelines or is dominated by a single heavyweight pipeline.

B. Analysis of Suspension and Resumption Strategy Selection

We analyze Riveter's selection of suspension and resumption strategies by examining scenarios in which termination time windows and termination probabilities are configured with different settings. In this evaluation, we predefine the termination probability P and the termination time window $[T_s, T_e]$. The termination probability P dictates the likelihood of a query being terminated during its execution. Should a termination occur, the exact timing of termination within the window $[T_s, T_e]$ is determined by a uniform distribution with the cumulative distribution 1% at T_s to 100% at T_e. In this section, we employ the notation *X-Y%* to represent the termination time window. This notation specifies the interval for the termination time window, beginning at approximately $X\%$ and ending at $Y\%$ of the total execution time. We run and analyze all the TPC-H queries using SF-100 datasets. All the results in §IV-B are averaged over ten independent runs.

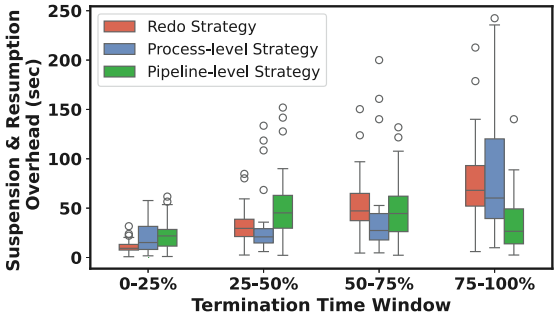

Fig. 10: Suspension and resumption overheads of TPC-H queries on the SF-100 dataset using three strategies under scenarios where termination probability is 100%.

We first investigate scenarios characterized by certain terminations, where the termination probability is 100%. Fig. 10 presents the overheads of suspension and resumption across all 22 TPC-H queries using the SF-100 dataset, utilizing three strategies under a range of termination time windows, yet maintaining a termination probability of 100%, as depicted through box plots. For Fig. 10, we deactivate the cost model and the mechanism for adaptive strategy selection within Riveter, thereby compelling Riveter to employ a predetermined strategy.

We measure the overhead as the difference between the normal execution time and the execution time that includes a single suspension and resumption using the specific strategy. Fig. 10 illustrates that for queries utilizing the redo strategy, overhead increases as the termination time windows shift from the initial 0–25% interval to the final 75–100% interval. This trend arises because the overhead under the redo strategy corresponds to the termination point (the execution time before termination is wasted), making the overhead increments directly proportional to the termination time windows and specific termination points. Similarly, overheads for queries applying the process-level strategy exhibit a gradual increase as termination time windows progress from 0–25% to 50–75%, with a marked rise in the 75–100% window. This pattern can be attributed primarily to the growth in intermediate data sizes as execution advances (referenced in Fig. 7), which results in extended serialization and persistence durations. Consequently, this not only requires additional time for suspension and resumption but also heightens the likelihood of failing to complete the suspension before termination. Overheads associated with the pipeline-level strategies exhibit a different pattern, showing a gradual increase from the $0-25\%$ to the $50-75\%$ termination time windows, but demonstrating a decrease from the $50-75\%$ to the $75-100\%$. This is due to the nature of the pipeline-level strategy, where feasible suspension points are pipeline breakers. A number of TPC-H queries are characterized by dominating pipelines, which results in pipeline breakers being predominantly situated near the beginning and end of the query execution process. Therefore, some TPC-H queries cannot be suspended timely when the termination time windows are located at the $50-75\%$ and the $50-75\%$ of query execution time, thereby, increasing the overhead. Fig. 10 presents the trade-off among the strategies, confirms our argument that adaptive query execution is necessary when terminations may happen, and highlights the advantages of Riveter.

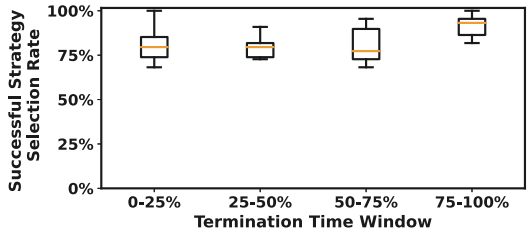

Fig. 11: Successful strategy selection rate under scenarios where termination probability is 100%.

To show the effectiveness of Riveter's strategy selection process, we activate its strategy selection mechanism and record the number of successful selections across all TPC-H queries on the SF-100 dataset over ten independent runs, as depicted in Fig. 11. A strategy selection is deemed successful if the execution of a query, under the strategy chosen by Riveter, is completed in the shortest time, even in the event of a termination. Fig. 11 demonstrates that Riveter often selects the best approach in different scenarios over ten runs.

We also explore scenarios where termination probabilities are varied, including 30%, 50%, 70%, and 90%, rather than being absolute at 100%. Below, we describe examples for queries Q1, Q3, Q17, and Q21 under different termination probabilities and time windows, as illustrated in Table III. The expected number of terminations is calculated using the termination probabilities and the total number of independent runs. For example, with our ten independent runs and a termination probability of 30%, the expected number of terminations would be 3.

Query Q1 serves as a representative example of queries that heavily rely on one of a few pipelines. For Q1, we configure a termination time window $[81s, 108s]$ (i.e., $75-100\%$ of expected execution time), with a 30% probability of termination. Once Q1 approaches this time window, Riveter assesses the cost associated with different suspension strategies. The cost of the redo strategy is notably high in this scenario. This is primarily because Q1 is nearing the completion of its execution, and any termination at this point would result in the loss of all progress, necessitating a complete restart. Consequently, this would lead to a significant increase in execution time. Meanwhile, the estimated cost of the process-level strategy is considerably higher than that of the pipeline-level strategy. This is attributed to the much larger size of intermediate data that must be persisted during the suspension period. Thus, Riveter recommends the pipeline-level strategy for Q1.

In the case of Query Q3, we analyze the strategy selection of Riveter when there is a potential for early termination during query execution. In this case, the first pipeline of Q3 reaches completion quite early in the process. Riveter, after performing its calculations, selects the redo strategy for suspension, primarily due to its lower cost compared to the other two strategies. This choice is bolstered by the fact that the termination time window occurs at the outset of query execution, resulting in a relatively modest cost for the redo strategy since the cost of the redo strategy is mainly associated with the query re-execution instead of persisting any intermediate data. During three independent runs, one termination event occurred, necessitating the need to rerun the query execution and thereby increasing the redo strategy's average running time.

In the case of Query Q17, the probability of query termination is relatively high. Consequently, suspending the query becomes a more reasonable choice than redoing it entirely. Riveter, taking this into account, opts for the process-level strategy over the pipeline-level strategy. The latter, which involves persisting relatively large amounts of data, introduces additional overhead. Furthermore, Riveter's estimation suggests that the next pipeline of Q3 will fully cover the termination time window, rendering the redo strategy more expensive due to the high termination probability.

Query Q21 is one of the most complex queries in TPC-H. We configure a termination time window at the middle phase of query execution, paired with a high termination probability, which suggests a higher expected number of terminations. According to Riveter's estimation, the size of intermediate states maintained for persistence during suspension, using a process-level strategy, substantially exceeds that of the pipeline-

Query	Configuration	Selected Strategy	Execution Time	Execution Time with Suspension
Q1	Termination Probability: 30%, Termination Time Window: 75-100%	Pipeline-level	108s	109.8s
Q3	Termination Probability: 50%, Termination Time Window: 0-25%	Redo	78.6s	82.4s
Q17	Termination Probability: 70%, Termination Time Window: 50-75%	Process-level	116.1s	132.9s
Q21	Termination Probability: 90%, Termination Time Window: 25-50%	Pipeline-level	242.4s	298s

TABLE III: Comparison of normal execution time versus execution time using Riveter's selected strategy

level strategy, which can introduce considerable overhead into the query execution process. Additionally, the termination time window initiates at 70 seconds. This factor diminishes the attractiveness of employing the redo strategy, given that termination has a 90% probability of occurring, resetting the progress and significantly extending the overall query execution time. Therefore, Riveter recommends adopting the pipeline-level strategy, as the execution time with a single suspension and resumption event introduces only negligible overhead. Furthermore, the observed increase in average execution time when including suspension and resumption, compared to normal execution time, is attributable to one independent run where suspension fails to complete before reaching the termination point. This failure necessitates restarting the query and significantly increases the overhead.

Our evaluation reveals that there is no universally superior suspension and resumption strategy and underscores the necessity of judicious selection of when/how to suspend a query.

C. Accuracy Estimation and Runtime of Cost Model

The cost model plays a crucial role in the Riveter framework, aiding in determining the optimal strategy based on latency estimations for various strategies. As detailed in §III-C, assessing the latency of the process-level strategy necessitates estimating the size of intermediate data; thus, we investigate the accuracy of this estimation within Riveter. We collect data from 200 query executions and employ a regression-based approach to fit the curve, and subsequently, leverage this curve to estimate the size of intermediate data when applying the process-level strategy in Riveter to suspended queries. The results of our estimations are presented in Table IV, offering valuable estimates to the cost model for identifying the strategy with the least latency.

Query	Dataset	Regression-based Estimate	Optimizer-based Estimate	Ground truth
Q1	SF-50	2.25GB	15.7GB	2.3GB
Q1	SF-100	4.75GB	31.3GB	4.3GB
Q3	SF-50	3GB	2.36×10^{15} GB	2.3GB
Q3	SF-100	5.2GB	1.89×10^{16} GB	4.2GB
Q17	SF-50	2.7GB	1.01×10^{17} GB	2.1GB
Q17	SF-100	4.5GB	8.05×10^{17} GB	5GB
Q21	SF-50	11.7GB	5.05×10^{15} GB	14GB
Q21	SF-100	24.3GB	4.05×10^{16} GB	28GB

TABLE IV: Estimation analysis of cost model when queries are suspended at around 50% using process-level strategy.

Furthermore, we conduct a robustness test using optimizer-based estimation that relies on estimated cardinality and data types associated with the input data table columns. As illustrated in Table IV, the optimizer-based estimation, with its independence from historical or real-time data, produces significantly inaccurate estimates compared to the ground-truth values. In the test, we configure Riveter to utilize the optimizer-based estimate for strategy selection in the case of Q17 (Table III). The optimizer-based estimation tends to overestimate intermediate data sizes and the associated overhead introduced by their persistence. Given termination probability and termination time window, this results in Riveter selecting the pipeline-level strategy that is a sub-optimal strategy, as shown in Fig. 12. Since the suspension of pipeline-level strategy only occurs when reaching pipeline breakers, the suspension of pipeline-level strategy for Q17 is deferred until a pipeline is processed completely (e.g., Fig. 9) and thereby overlaps with the termination time window. In three experimental runs, this lag leads to two terminations occurring before the pipeline-level suspension is completed.

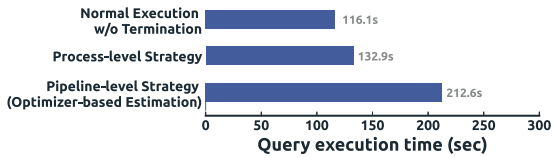

Fig. 12: Query suspension strategy selection for Q17 based on optimizer-based estimation.

In our implementation of Riveter, the cost model is launched as a dedicated process and efficiently communicates with the process where the query is executed through allocated shared memory. We measure the running time of the cost model, including data ingestion for estimation, latency estimation for the process-level strategy, and evaluation of latency for other strategies to determine the optimal strategy. Specifically, Riveter runs query Q1, Q3, Q17, and Q21 on the SF-100 dataset and suspends them at around 50% of their execution time. We capture the running time of the cost model when it is triggered for optimal strategy selection for this suspension request. As tabulated in Table V, the running time of the cost model for Q1, Q3, and Q21 is negligible. The cost model for Q17 brings a relatively higher overhead to the query execution. This is primarily due to determining the size of larger intermediate data of Q17 during suspension, a critical step in the cost model, which requires more time to complete and introduces additional overhead to the cost model.

Query	Running Time of Cost Model	Overall Execution Time (no suspension)
Q1	0.012s	108.4s
Q3	0.018s	69.3s
Q17	5.85s	115.8s
Q21	0.02s	223.6s

TABLE V: Running time of cost model in Riveter for various queries on the SF-100 dataset

V. Related Work

In this section, we identify some key domains where Riveter can effectively contribute, and compare it with established relevant techniques within those domains.

A. Recovery, Checkpoint, and Suspension

Recovery, checkpoint, and suspension can be traced from a similar lineage in databases. ARIES [26], a canonical algorithm for database recovery, enables features like steal and no-force buffer management, partial rollbacks, and fuzzy checkpoints. Disk-based databases often use ARIES or similar methods for recovery [27]. However, in-memory databases typically avoid ARIES-style recovery to boost performance, prioritizing minimal I/O overhead by accessing persistent storage primarily for recovery purposes. Although many disk-resident database systems implement ARIES-alike mechanisms for recovery purposes, in-memory database systems avoid using ARIES-style for performance reasons [19].

The checkpoint mechanism is vital for improving recovery processes, creating a reference point for persistently capturing the database's current state. Unlike disk-based systems, main-memory databases use an asynchronous method to move transaction updates from the log to a checkpoint archive, speeding up recovery and freeing log space [28]. Main-memory databases typically generate consistent snapshots instead of individual log records, representing the database's state at a specific time [29]. The snapshot-based approaches enable in-memory databases [30] and modern disk-resident databases [31] to reload the latest snapshot after a system crash quickly.

In light of checkpoint mechanisms, Chandramouli et al. introduced a pull-based query execution approach that operates at the operator level and focuses on query suspension and resumption [21]. This approach, which can be integrated into Riveter, differs from the existing suspension and resumption strategy in Riveter. The comparison is illustrated in Table VI. Graefe et al. [32] and Antonopoulos et al. [33] explored the possibilities of implementing pause and resume functionality for index operations in commercial database systems. Similar to query suspension, the query preemption mechanism can pause or checkpoint some ongoing jobs in favor of others due to specific objectives. SaGe [34] focuses on SPARQL queries and allows the queries to be suspended by the Web server after a fixed period and resumed upon client request. Rotary [14] is a general resource arbitration framework, but one of its implementations, Rotary-AQP can checkpoint some long-running AQP jobs during execution and resume it using the persisted state when it is beneficial to do so.

Riveter distinguishes itself from the aforementioned works in two main aspects. First, Riveter is under the assumption that potential resource terminations are either predictable or reasonably estimated, facilitating more informed decision-making in Riveter. Second, Riveter is designed as an adaptive framework that integrates different query suspension and resumption strategies. It can select the most appropriate strategy according to the proposed cost model and then execute the strategy for query suspension and resumption at various levels, ensuring the sustained progress of queries even in scenarios with fluctuating resources.

B. Database Migration

Database migration facilitates the transfer of databases and analytics workloads from one execution environment to another, often requiring the database to stop and restart, similar to pausing and resuming queries. One state-of-the-art method, live database migration, utilizes virtualization [35] to migrate database services with minimal interruption and negligible performance impact [36], [37]. Live database migration involves two main methods: iterative copying [36], [38] and dual execution [39], [40]. Iterative copying, akin to the methodology employed by Riveter, pauses active transactions on the source database during migration, transferring the database cache and ongoing transaction states to start the destination with a hot cache [41], [42]. On the other hand, dual execution, introduced by Zephyr and improved in systems like MgCrab [43] and Remus [40], allows transactions to run concurrently on both source and destination databases, reducing service disruptions and downtime to a minimum. Slacker [44] and Madeus [45] are middleware solutions that synchronize databases without altering them directly.

State migration is crucial in streaming databases for reconfiguring stateful operators, often employing a "stop-and-restart" method, a kind of redo strategy. This approach involves halting program execution to securely transfer state, then restarting the job once state redistribution is complete, leveraging existing fault-tolerance mechanisms [46], [47]. Typically, only a few operators need state migration, allowing the rest to operate uninterrupted. Fault-tolerance checkpoints can be utilized to facilitate the process of state migration [48], [49], [50].

Riveter exhibits orthogonality to live or state migration, augmenting their capabilities within resource-dynamic environments. Riveter facilitates query migration as opposed to full database migration, employing various suspension and resumption strategies. This approach improves the feasibility of live migration in scenarios where resources are ephemeral or fluctuating. Migrating queries proves to be less resource-intensive than relocating entire database states, primarily due to the smaller intermediate states.

C. Query Scheduling

Query scheduling is not uncommon in data systems deployed within cloud environments, particularly for resource allocation among multiple users within multi-tenant databases [51]. These databases need to ensure that each tenant meets their Service

	Query Execution Model	Timing of Suspensions	Single or Multiple Threads
Chandramouli et al.[21]	pull-based (or iterator) model	suspends at operators with low memory usage	works for single-thread query execution
Pipeline-level strategy	push-based (or morsel-driven) model	suspends at pipeline-breakers	works for multi-threads query execution

TABLE VI: Comparison between pipeline-level strategy and reference [21]

Level Objectives (SLOs) by receiving adequate resources for request handling. Addressing this involves mechanism like resource isolation [52], [53] and intelligent tenant placement [54], [55]. Yet, this responsibility becomes challenging with the increase of long-running queries, which risk exhausting cloud resources. Current methods—allocating substantial resources to these queries or repositioning them—either hasten resource saturation or increase query latency. Recent research endeavors have explored the possibility of preempting certain long-running jobs during their execution in favor of prioritizing others [14].

In classic approaches, cloud resources are presumed to be persistent and stable. In contrast, Riveter operates under the assumption that these resources are ephemeral and subject to change over time. Furthermore, Riveter offers substantial enhancements to existing approaches. This enhancement stems from the application of suspension and resumption techniques, which allow a long-running query to be segmented into a sequence of smaller tasks. This segmentation, in turn, enables more efficient query scheduling and resource allocation.

VI. Discussion

In this section, we discuss implementation choices, share the insights gained, and highlight open questions.

More Suspension and Resumption Strategies. Riveter serves as an adaptable query suspension and resumption framework capable of accommodating a wide range of strategies. In our implementation, we present three representative strategies: redo, pipeline-level, and process-level strategy, to demonstrate Riveter's functionality and performance. Nevertheless, Riveter is versatile and can support additional strategies while adapting to more complex scenarios. For example, it is valuable to implement a data-level strategy that can partition input datasets and execute queries in batch mode, particularly when developing a suspension-oriented query execution engine proves to be complicated. Alternatively, an operator-level strategy can be implemented to offer finer-grained suspension capabilities for scenarios involving iterator-based query execution.

Multiple Suspensions during Query Execution. While the proposed Riveter can be extended to accommodate scenarios with multiple suspension events, our performance evaluation of Riveter mainly focuses on scenarios that involve a single suspension and resumption event. This choice is motivated by the fact that each suspension event can be treated as an independent occurrence, and latency increases proportionally with the number of suspensions. As part of our future work, we will also explore more complex scenarios involving multiple queries, where the individual suspension and resumption of queries may interact with one another.

Persistence and Serialization Overhead. One insight we learned from our implementation and evaluation is that Riveter requires significant time to serialize and persist intermediate data upon suspension, leading to considerable overhead to query execution. Therefore, devising a strategy to minimize the intermediate data serialization and persistence emerges as a crucial enhancement. For example, data can be sorted before execution, and Riveter can leverage this pre-sorted data layout while continuously tracking the watermark during execution. The watermark can serve as intermediate data, diminishing intermediate data size and reducing the overhead.

Query Re-optimization and Isolation. Riveter assumes that query plans remain the same when suspending and resuming queries. However, it presents a challenging yet beneficial avenue to explore the possibility of altering the query plan after suspension but before resumption. This leads to the question of how Riveter should determine the strategy for query suspension and the nature of intermediate data persistence to maximize the benefits of query re-optimization. Furthermore, Riveter excludes the data modifications during suspension for query isolation. An outstanding question remains regarding the methodologies to ensure isolation amidst data modifications. While snapshot isolation emerges as a viable strategy, further investigation is warranted to broaden the scope of potential solutions.

VII. Conclusion

This paper argues that resources are increasingly fluctuating and ephemeral in modern computing infrastructure, which necessitates the development of an adaptive query execution mechanism for these scenarios, as exemplified by modern cloud-native databases. We present Riveter, a framework designed and developed for adaptive query execution. It offers a spectrum of strategies for suspending queries and resuming them when deemed necessary or advantageous; it also confirms that no single suspension and resumption technique is dominant and careful selection of when/how to suspend a query is needed. To determine the strategies for different queries, we have developed an algorithm based on a cost model tailored to the characteristics of these suspension strategies. Our experimental evaluation, leveraging the TPC-H benchmark, explores the intermediate data persistence within different strategies, analyzes strategy selection guided by a cost model for various queries, and examines the estimation accuracy and runtime performance associated with the employed cost model. The findings underscore that the efficacy of query suspension and resumption strategies is contingent upon multiple factors. Thus, cloud-native databases need to select strategies that align with their specific operational contexts.

Acknowledgment

This work was in part supported by NSF Award IIS-2048088 and a Google Data Analytics and Insights (DANI) Award. The authors would like to thank the anonymous reviewers for their insightful comments.

REFERENCES

[1] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *ACM Symposium on Cloud Computing (SoCC)*, 2012, p. 7.

[2] J. Wang, T. Li, H. Song, X. Yang, W. Zhou, F. Li, B. Yan, Q. Wu, Y. Liang, C. Ying, Y. Wang, B. Chen, C. Cai, Y. Ruan, X. Weng, S. Chen, L. Yin, C. Yang, X. Cai, H. Xing, N. Yu, X. Chen, D. Huang, and J. Sun, "Polardb-imci: A cloud-native HTAP database system at alibaba," *Proceedings of the ACM on Management of Data*, vol. 1, no. 2, pp. 199:1–199:25, 2023.

[3] "Amazon ec2 spot instances," https://aws.amazon.com/ec2/spot, 2023, accessed: 2023-09-10.

[4] Q. Zhang, P. A. Bernstein, D. S. Berger, B. Chandramouli, V. Liu, and B. T. Loo, "Compucache: Remote computable caching using spot vms," in *Conference on Innovative Data Systems Research (CIDR)*, 2022.

[5] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. N. Tantawi, and C. Krintz, "See spot run: Using spot instances for mapreduce workflows," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2010.

[6] R. Liu, J. H. Chang, R. Otaki, Z. H. Eng, A. J. Elmore, M. J. Franklin, and S. Krishnan, "Towards resource-adaptive query execution in cloud native databases," in *Conference on Innovative Data Systems Research (CIDR)*, 2024.

[7] H. Shafiei, A. Khonsari, and P. Mousavi, "Serverless computing: A survey of opportunities, challenges, and applications," *ACM Computing Survey*, vol. 54, no. 11s, pp. 239:1–239:32, 2022.

[8] O. Poppe, Q. Guo, W. Lang, P. Arora, M. Oslake, S. Xu, and A. Kalhan, "Moneyball: Proactive auto-scaling in microsoft azure SQL database serverless," *VLDB Endowment*, vol. 15, no. 6, pp. 1279–1287, 2022.

[9] A. A. Chien, "Driving the cloud to true zero carbon," *Communications of the ACM*, vol. 64, no. 2, p. 5, 2021.

[10] V. R. Narasayya and S. Chaudhuri, "Multi-tenant cloud data services: State-of-the-art, challenges and opportunities," in *ACM International Conference on Management of Data (SIGMOD)*, 2022, pp. 2465–2473.

[11] "Amazon ec2 spot instances pricing," https://aws.amazon.com/ec2/spot/pricing, 2023, accessed: 2023-09-10.

[12] Z. Luo, L. Niu, V. Korukanti, Y. Sun, M. Basmanova, Y. He, B. Wang, D. Agrawal, H. Luo, C. Tang, A. Singh, Y. Li, P. Du, G. Baliga, and M. Fu, "From batch processing to real time analytics: Running presto® at scale," in *IEEE International Conference on Data Engineering (ICDE)*, 2022, pp. 1598–1609.

[13] P. Garefalakis, K. Karanasos, P. R. Pietzuch, A. Suresh, and S. Rao, "Medea: scheduling of long running applications in shared production clusters," in *European Conference on Computer Systems (EuroSys)*, 2018, pp. 4:1–4:13.

[14] R. Liu, A. J. Elmore, M. J. Franklin, and S. Krishnan, "Rotary: A resource arbitration framework for progressive iterative analytics," in *IEEE International Conference on Data Engineering (ICDE)*, 2023, pp. 2140–2153.

[15] "Duckdb is an in-process sql olap database management system," https://github.com/duckdb/duckdb, 2023, accessed: 2023-09-05.

[16] "Criu: A project to implement checkpoint/restore functionality for linux," https://github.com/checkpoint-restore/criu, 2023, accessed: 2023-07-04.

[17] "Tpc-h benchmark," https://www.tpc.org/tpch/default5.asp, 2023, accessed: 2023-09-11.

[18] C. Mohan, D. Haderle, B. G. Lindsay, H. Pirahesh, and P. M. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems*, vol. 17, no. 1, pp. 94–162, 1992.

[19] A. Magalhães, J. M. Monteiro, and A. Brayner, "Main memory database recovery: A survey," *ACM Computing Survey*, vol. 54, no. 2, pp. 46:1–46:36, 2022.

[20] T. Härder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Survey*, vol. 15, no. 4, pp. 287–317, 1983.

[21] B. Chandramouli, C. N. Bond, S. Babu, and J. Yang, "Query suspend and resume," in *ACM International Conference on Management of Data (SIGMOD)*, 2007, pp. 557–568.

[22] A. Kumar, Z. Wang, S. Ni, and C. Li, "Amber: A debuggable dataflow system based on the actor model," *VLDB Endowment*, vol. 13, no. 5, pp. 740–753, 2020.

[23] S. A. M. Tajalli, S. Z. Tajalli, M. Homayounzadeh, and M. H. Khooban, "Zero-carbon power-to-hydrogen integrated residential system over a hybrid cloud framework," *IEEE Transactions on Cloud Computing*, vol. 11, no. 3, pp. 3099–3110, 2023.

[24] "Cost-based optimizer," https://docs.databricks.com/en/optimizations/cbo.html, 2023, accessed: 2023-11-06.

[25] "Apache parquet," https://parquet.incubator.apache.org, 2023, accessed: 2023-09-29.

[26] C. Mohan and F. E. Levine, "ARIES/IM: an efficient and high concurrency index management method using write-ahead logging," in *ACM International Conference on Management of Data (SIGMOD)*, 1992, pp. 371–380.

[27] C. Mohan and I. Narang, "ARIES/CSA: A method for database recovery in client-server architectures," in *ACM International Conference on Management of Data (SIGMOD)*, 1994, pp. 55–66.

[28] K. Salem and H. Garcia-Molina, "Checkpointing memory-resident databases," in *IEEE International Conference on Data Engineering (ICDE)*, 1989, pp. 452–462.

[29] K. Ren, T. Diamond, D. J. Abadi, and A. Thomson, "Low-overhead asynchronous checkpointing in main-memory database systems," in *ACM International Conference on Management of Data (SIGMOD)*, 2016, pp. 1539–1551.

[30] L. Lee, S. Xie, Y. Ma, and S. Chen, "Index checkpoints for instant recovery in in-memory database systems," *VLDB Endowment*, vol. 15, no. 8, pp. 1671–1683, 2022.

[31] M. Haubenschild, C. Sauer, T. Neumann, and V. Leis, "Rethinking logging, checkpoints, and recovery for high-performance storage engines," in *ACM International Conference on Management of Data (SIGMOD)*, 2020, pp. 877–892.

[32] G. Graefe, W. Guy, and H. A. Kuno, "'pause and resume' functionality for index operations," in *International Workshop on Self Managing Database Systems (SMDB@ICDE)*. IEEE, 2011, pp. 28–33.

[33] P. Antonopoulos, H. Kodavalla, A. Tran, N. Upreti, C. Shah, and M. Sztajno, "Resumable online index rebuild in SQL server," *VLDB Endowment*, vol. 10, no. 12, pp. 1742–1753, 2017.

[34] T. Minier, H. Skaf-Molli, and P. Molli, "Sage: Web preemption for public SPARQL query services," in *The World Wide Web Conference (WWW)*, 2019, pp. 1268–1278.

[35] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI*, 2005.

[36] S. Das, S. Nishimura, D. Agrawal, and A. El Abbadi, "Live database migration for elasticity in a multitenant database for cloud platforms," *UCSB Computer Science Technical Report*, 2010.

[37] E. Wu, S. Madden, Y. Zhang, E. Jones, and C. Curino, "Relational cloud: The case for a database service," *MIT CSAIL Technical Report*, 2010.

[38] S. Das, D. Agrawal, and A. E. Abbadi, "Elastras: An elastic, scalable, and self-managing transactional database for the cloud," *ACM Transactions on Database Systems*, vol. 38, no. 1, p. 5, 2013.

[39] A. J. Elmore, S. Das, D. Agrawal, and A. E. Abbadi, "Zephyr: live migration in shared nothing databases for elastic cloud platforms," in *ACM International Conference on Management of Data (SIGMOD)*, 2011, pp. 301–312.

[40] J. Kang, L. Cai, F. Li, X. Zhou, W. Cao, S. Cai, and D. Shao, "Remus: Efficient live migration for distributed databases with snapshot isolation," in *ACM International Conference on Management of Data (SIGMOD)*, 2022, pp. 2232–2245.

[41] S. Das, S. Nishimura, D. Agrawal, and A. E. Abbadi, "Albatross: Lightweight elasticity in shared storage databases for the cloud using live data migration," *VLDB Endowment*, vol. 4, no. 8, pp. 494–505, 2011.

[42] X. Wei, S. Shen, R. Chen, and H. Chen, "Replication-driven live reconfiguration for fast distributed transaction processing," in *USENIX Annual Technical Conference (USENIX ATC)*, 2017, pp. 335–347.

[43] Y. Lin, S. Pi, M. Liao, C. Tsai, A. J. Elmore, and S. Wu, "Mgcrab: Transaction crabbing for live migration in deterministic database systems," *VLDB Endowment*, vol. 12, no. 5, pp. 597–610, 2019.

[44] S. K. Barker, Y. Chi, H. J. Moon, H. Hacigümüs, and P. J. Shenoy, ""cut me some slack": latency-aware live migration for databases," in *International Conference on Extending Database Technology (EDBT)*, 2012, pp. 432–443.

[45] T. Mishima and Y. Fujiwara, "Madeus: Database live migration middleware under heavy workloads for cloud environment," in *ACM International Conference on Management of Data (SIGMOD)*, 2015, pp. 315–329.

[46] P. Carbone, S. Ewen, G. Fóra, S. Haridi, S. Richter, and K. Tzoumas, "State management in apache flink®: Consistent stateful distributed stream processing," *VLDB Endowment*, vol. 10, no. 12, pp. 1718–1729, 2017.
[47] M. Armbrust, T. Das, J. Torres, B. Yavuz, S. Zhu, R. Xin, A. Ghodsi, I. Stoica, and M. Zaharia, "Structured streaming: A declarative API for real-time applications in apache spark," in *ACM International Conference on Management of Data (SIGMOD)*, 2018, pp. 601–613.
[48] R. C. Fernandez, M. Migliavacca, E. Kalyvianaki, and P. R. Pietzuch, "Integrating scale out and fault tolerance in stream processing using operator state management," in *ACM International Conference on Management of Data (SIGMOD)*, 2013, pp. 725–736.
[49] L. Mai, K. Zeng, R. Potharaju, L. Xu, S. Suh, S. Venkataraman, P. Costa, T. Kim, S. Muthukrishnan, V. Kuppa, S. Dhulipalla, and S. Rao, "Chi: A scalable and programmable control plane for distributed stream processing systems," *VLDB Endowment*, vol. 11, no. 10, pp. 1303–1316, 2018.
[50] M. Hoffmann, A. Lattuada, F. McSherry, V. Kalavri, J. Liagouris, and T. Roscoe, "Megaphone: Latency-conscious state migration for distributed streaming dataflows," *VLDB Endowment*, vol. 12, no. 9, pp. 1002–1015, 2019.
[51] A. J. Elmore, C. Curino, D. Agrawal, and A. E. Abbadi, "Towards database virtualization for database as a service," *VLDB Endowment*, vol. 6, no. 11, pp. 1194–1195, 2013.
[52] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri, "SQLVM: performance isolation in multi-tenant relational database-as-a-service," in *Conference on Innovative Data Systems Research (CIDR)*, 2013.
[53] R. Liu, D. Wong, D. Lange, P. Larsson, V. Jethava, and Q. Zheng, "Accelerating container-based deep learning hyperparameter optimization workloads," in *Workshop on Data Management for End-To-End Machine Learning (DEEM@SIGMOD)*, 2022, pp. 6:1–6:10.
[54] Z. Liu, H. Hacigümüs, H. J. Moon, Y. Chi, and W. Hsiung, "PMAX: tenant placement in multitenant databases for profit maximization," in *International Conference on Extending Database Technology (EDBT)*, 2013, pp. 442–453.
[55] R. Liu, S. Krishnan, A. J. Elmore, and M. J. Franklin, "Understanding and optimizing packed neural network training for hyper-parameter tuning," in *Workshop on Data Management for End-To-End Machine Learning (DEEM@SIGMOD)*, 2021, pp. 3:1–3:11.