Towards Resource-adaptive Query Execution in Cloud Native Databases

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

Jun Hyuk Chang
junhyukc@uchicago.edu
University of Chicago

Riki Otaki
rotaki@uchicago.edu
University of Chicago

Zhe Heng Eng
zhehengeng@uchicago.edu
University of Chicago

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

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

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

ABSTRACT

Modern cloud environments, characterized as resource-dynamic with new developments, see prevalence in ephemeral resources. Such resources can be unstable in resource availability, experiencing both anticipated and unforeseen terminations during utilization. Their prices, although attractive, can be fluctuating over time. The presence and prevalence of ephemeral resources in cloud environments pose a challenge to current cloud-native databases and workloads, which requires a rethink of design principles and necessitates the new primitives: query preemption, resource arbitration, and cost tolerance. In this paper, we design Ratchet, a resource-adaptive query execution framework, to realize the identified primitives. Ratchet enables adaptive query suspension and resumption at various granularities, resource arbitration for complex and heterogeneous workloads, and a fine-grained pricing model to utilize dynamic cloud resources without the risk of unexpectedly high prices. We also explore emerging directions to inspire future research.

1 INTRODUCTION

Data-intensive systems are migrating towards cloud-native architectures that offer low-latency, consistent, and pay-as-you-go query answering [23], exemplified by cloud-native databases. This extends beyond being merely a deployment trend; it represents a critical point prompting a reconsideration of the core architectural decisions that form the foundation of these systems, which is driven by one of the key factors, the increasing prevalence of ephemeral cloud resources.

First, ephemeral cloud resources are dynamic in availability with potential terminations. Spot instances, offering short-lived computing infrastructure, have been prevalent for a while. Such resources may experience some terminations when resource providers need them back due to limited availability or other constraints. New developments are amplifying the bursty capacity. For instance, serverless platforms offer applications the convenience of employing lightweight Virtual Machines (VMs) that have limited run-time, memory capacity, and addressable addresses [19]. Recent efforts from the database community have shown how to develop a query processing framework on top of a serverless platform [32]. Another emerging cloud paradigm is "zero-carbon clouds" [8], where data centers can be completely ephemeral as they are largely powered by renewable sources that are not always stable for continuous supply. Hence, cloud-native databases designed for zero-carbon clouds must effectively ready themselves to handle the transient nature of resources.

Second, the prices of ephemeral cloud resources can be attractive but significantly fluctuate over time. Although the prices of ephemeral cloud resources are typically appealing during off-peak times compared with classic reserved and on-demand ones [2], such prices have been reported to skyrocket dramatically, reaching 200 to 400 times the normal rate during periods of peak demand [34]. Thus, even if users usually expect low latency, there is an increased demand for economically viable solutions, provided they do not significantly compromise performance. A growing number of users are beginning to favor cost-conscious options that may result in slightly increased latency or stale results [1].

Therefore, the prevalence of ephemeral cloud resources poses a new opportunity for systems designers to reduce the operational costs of database systems, yet the dynamic and fluctuating nature of such resources is often incompatible with current cloud-native database designs and workloads. From the perspective of design principles, the convention of databases pre-reserving what are assumed to be stable resources to maintain low latency has become less applicable in these resource-dynamic environments; furthermore, charging users through a pay-as-you-go model may not always be practical, considering the ephemerality of resources and the user's desire to evade peak times for money efficiency. From the perspective of workloads, with the recent proliferation of data-driven applications, many organizations have increasingly complex and heterogeneous workloads, including both long-running and short-running queries [17, 26], where long-running queries may occupy resources for extended periods and are not readily suited for ephemeral resources. This can lead to significant delays for shorter-running queries that might have otherwise been completed promptly with sufficient resources. Many of these long-running queries can often be progressive as well, where an iterative loop repeatedly refines a result until the desired completion criterion is met – these are the so-called iterative and progressive queries [24]. To keep allocating limited and dynamic resources to queries that have already achieved significant progress may, under certain conditions, curtail the efficient utilization of these resources for the rest of the workload.

Considering the trend towards ephemeral cloud resources and the challenges they introduce, we believe it is beneficial to rethink the design principles of cloud-native databases, and thus, we propose three primitives:

Primitive 1: Query Preemption. The feasibility of permitting queries to constantly consume ephemeral cloud resources is limited because (1) the resources are dynamic and infeasible to be reserved and sustained for a prolonged period, and (2) the attempt to continuously allocate ephemeral resources to long-running queries, particularly when they yield diminishing returns, can be counterproductive. Thus, such long-running queries ought to be paused when necessary or beneficial. This primitive fosters more efficient and flexible query execution by transforming a single long-running query into a sequence of shorter ones. This is also advantageous in scenarios where partial results are acceptable or meaningful, such as approximate query processing or deep learning training.

Primitive 2: Resource Arbitration. Given the ephemeral resources that fluctuate in availability and monetary cost, "how many resources a query needs?", a question answered by existing resource reservation approaches, no longer holds substantial value. Instead, the emergent question prompted by this primitive is: "is it worth allocating resources to a particular query?". Answering this question necessitates a sophisticated mechanism that can adaptively determine if, when, how much, and for how long a query should be allocated resources. This decision should consider multiple factors, such as user needs, available resources, and the progress of each query at various times.

Primitive 3: Cost Tolerance. Users have diverse monetary cost tolerances. Low latency with high cost may not be the best option for all of them, particularly those who prioritize cost-efficiency over speed. However, waiting until the resources become stable or the end of peak periods to avoid high costs is not an ideal solution either. Hence, a more fine-grained pricing model becomes vital, which not only allows users to trigger query execution whenever sufficient resources are available but also provides the opportunity to circumvent unanticipated spikes in resource prices.

To realize the identified primitives, we design Ratchet, a novel resource-adaptive query execution framework for cloud-native databases that can be readily deployed to the environment with ephemeral cloud resources. We further describe three prototypical and exemplary components in Ratchet. Specifically, we design (1) an adaptive query execution framework, which enables query suspension and resumption at various granularities, (2) a resource arbitration mechanism, responsible for determining resources allocation for query suspension and resumption during runtime, (3) a cost model that provides users with a more granular set of cloud resources utilization and pricing options. Furthermore, we discuss some promising research directions inspired by Ratchet.

2 RATCHET

We elaborate on the three proposed primitives.

2.1 Query Preemption

Query preemption entails the ability to pause a running query in a partially complete state, thereby releasing resources for other jobs that could use them more efficiently. The preempted query can be resumed later when its continuation is considered beneficial. Thus, for efficient query preemption, we present an adaptive query suspension and resumption that holds the potential to substantially improve the flexibility of query execution in cloud-native databases, especially with ephemeral resources.

The original concept of query suspension and resumption is proposed to create a query suspension plan for a pull-based model, involving a combination of persisting current state and reverting to previous checkpoints based on the overhead of state serialization to disk [7]. This method presents certain constraints, such as both the suspended and resumed queries requiring the same resources and database state and a query plan being executed as a single thread. We revisit this concept and believe that a novel solution is necessary to accommodate the new trends and realize the proposed primitives.

We take an initial step by designing a novel adaptive query suspension and resumption framework by integrating different strategies across various granularities. The redo strategy and process-level strategy represent two opposing points on the spectrum of query suspension and resumption. The redo strategy, which is grounded in recovery mechanisms [27, 29], re-executes a query that was prematurely terminated before attaining its final results; whereas the process-level strategy allows a query to be suspended and resumed at any given time by capturing and persisting all the intermediate data of query execution and the context data of the process where the query is associate with. In between the redo and process-level strategy, three strategies can facilitate suspending and resuming queries at the data batch, operator, and pipeline levels. Specifically, the data batch-level strategy places checkpoints between data batches during query execution, analogous to the established checkpoint mechanism in streaming-style systems [3]. These checkpoints serve as resumption points in cases of unexpected termination. The operator-level strategy, inspired by the early query suspension and resumption mechanism [7], enables query suspensions to occur when some operators are completed during query execution. The pipeline-level strategy enables the query suspension and resumption for multi-threads pipeline-driven query execution and provides more flexibility (e.g., different resource configurations) when resuming queries.

Each aforementioned suspension and resumption strategy possesses distinct characteristics, and we proposed five metrics for a clear portrait from various perspectives:

- ***Agility*** is the speed at which the suspension can be triggered, i.e., how quickly the suspension process can start after the suspension request is initiated
- ***Capacity*** measures the amount of intermediate data and states needed to be persisted during query suspension.
- ***Adaptivity*** designates whether the strategy can adaptively utilize the available resources for query resumption, regardless of whether the available resources surpass or fall short of the resources during the suspension of the query.
- ***Complexity*** evaluates the development efforts to achieve such a strategy, e.g., whether modifying the existing data systems or importing additional toolkits is necessary.
- ***Preservability*** indicates the progress a strategy can preserve when potential resource terminations happen, i.e., how much processing progress of a query can be retained when employing the suspension and resumption strategy to handle potential terminations during query execution.

	Agility	*Capacity*	*Adaptivity*	*Complexity*	*Preservability*
Redo	Terminate at anytime	No intermediate data & state	Redo queries with available resources	No additional efforts	Lost all progress
Data Batch-Level Suspend & Resume	Suspend until some data batch processed	Checkpoints after data-batch processing	Resumption with available resources	Partition data and determine checkpoints	Lost progress since last persisted data batch
Operator-Level Suspend & Resume	Suspend until some operator complete	Persisted intermediate data & state of operator	Resumption needs same resources as suspension	Modified data systems	Lost progress since last persisted operator
Pipeline-Level Suspend & Resume	Suspend until some pipeline complete	Persisted intermediate data & state of pipeline	Resumption with available resources	Modified data systems	Lost progress since last persisted pipeline
Process-Level Suspend & Resume	Suspend anytime at process level	Persisted intermediate data & state of process	Resumption needs same resources as suspension	Import additional toolkit	Preserved all progress

Table 1: Analysis of suspension & resumption strategies

We further analyze the strategies using the proposed metrics in Table 1. The redo strategy permits the termination of a query at any time without persisting any intermediate state for query resumption, which implies no processing progress will be retained. It re-runs the query using currently available resources when a predicated or unforeseen termination happens, thus further development or changes are unnecessary. The process-level suspension and resumption strategy, typically relying on additional tools for operation at the process level, facilitates the suspension of queries at any given moment. This strategy can pause the query and preserve the current processing progress, retaining all intermediate data and contextual states of the process. However, the practice of "storing everything" presents two significant downsides. First, the volume of persisted data can be exceedingly large, potentially leading to extra latency. Second, it restricts the resumption of processes and queries to the same resource configurations (e.g., the number of hardware threads and the allocated memory size) that were in use at the time of suspension. The data batch-level strategy involves the suspension of query execution after the processing of one or multiple data batches. This strategy aims to retain intermediate data in a checkpoint-like manner following the completion of each data batch processing stage so that the query progress can be preserved at the batch level if a suspension is triggered and finished before terminations occur. The operator-level strategy enables individual physical query operators to execute lightweight checkpointing based on their specific semantics and should facilitate the coordination of checkpoints among operators through a dependency management mechanism. This strategy permits suspension during query execution, maintaining progress at the level of query operators. It also necessitates additional development of the query execution engine and other components in cloud-native databases. The pipeline-level strategy offers a different approach by preserving the intermediate data and states of each pipeline in the query plan for resumption, which implies that suspension and data persistence can only occur once certain pipelines have concluded. This approach can reduce the volume of data that needs to be stored but only suspend queries at specific points. Furthermore, this strategy also demands data systems alterations, as it modifies how queries are executed in normal pipeline-driven execution engines.

When dealing with complex and heterogeneous workloads in cloud environments with ephemeral resources, the adaptive suspension and resumption of queries at different granularities become crucial. Ratchet achieves this objective by utilizing different query suspension and resumption strategies. We describe the functionality of Ratchet and detail two representative strategies: pipeline-level and process-level strategy.

2.1.1 Query Preemption Principles.

Ratchet determines query suspension and resumption strategies based on two principles: (1) query suspension with an approximate time window and (2) query resumption with different resources.

Query suspension with approximation. A prevalent assumption in query suspension is the fixed and predetermined time point for suspension. However, this assumption may not always be valid in many real-world applications. For instance, accurately predicting the exact timing of resource availability in zero-carbon clouds or serverless computing is often challenging. Instead, it is more reasonable to have an approximate awareness of when the resource will become unavailable. Hence, the query can be suspended, preserving its processing progress before potential termination occurs.

Query resumption with available resources Some existing approaches for query resumption are hindered by a key constraint: they must resume queries with the identical resource configuration as when they were suspended. This constraint substantially curtails the flexibility in query resumption. For example, resuming a suspended query demanding the same resource level as at the point of suspension is not always achievable when the currently available resources prove to be inadequate. Within the multi-tenant databases where query scheduling is frequent and vital, this not only impedes the query resumption process but also results in less efficient resource utilization, leading to an overall increase in latency for the tenants. Thus, it is crucial and beneficial to maximize all accessible resources for query resumption or migration.

The principles of Ratchet are practical as they align with two emerging trends. Firstly, while it is impossible to anticipate query suspensions in cloud environments with absolute certainty, they can often be predictable with a rough timeframe. Secondly, in typical resource-dynamic environments, it is not uncommon for the

resource configuration at the time of query suspension to differ from that at the time of resumption.

Ratchet can take as input a termination time window and a probability, jointly determining a potential termination point, along with resource configurations for resumption. Specifically, Ratchet accommodates the termination point through a time window $[t_s, t_e]$ with a predefined probability p or a tailored probability function, $\pi(t_s, t_e)$, where π can be substituted by a probability distribution or an equivalent function. Thus, the query suspension and resumption are invoked by:

```
db.execute_suspend(query, suspend_loc, t_s, t_e, p)
db.execute_suspend(query, suspend_loc, t_s, t_e, π)
db.execute_resume(query, resume_loc, resource_conf)
```

Here, p and π refer to a termination probability and a probability function, respectively. *suspend_loc* represents the location for persisting intermediate data and state during a suspension. The persisted intermediate data located in *resume_loc* and a specific resource configuration *resource_conf* are indicated so that Ratchet can adopt it for query resumption.

2.1.2 *Pipeline-level Suspension & Resumption.*

Many modern database systems exploit many cores to execute queries, which is best exemplified by multi-threads pipeline-driven query execution [13, 22]. This execution mode divides a query plan into several pipelines based on pipeline breakers, with each pipeline being executed using multiple threads to enhance parallelism. Each thread maintains a local state to store intermediate data of this thread for a pipeline. Each pipeline owns a global state to merge all the thread-level intermediate data when the pipeline is finalized and generate processed results for the subsequent pipeline. This centralized-style approach positions pipeline breakers as natural points for suspension and resumption. Essentially, persisting the global state of a pipeline is equivalent to capturing the pipeline's intermediate data, thereby providing a safe point at which to suspend queries without losing substantial progress. Likewise, it is convenient to resume a query using the persisted global state.

The pipeline-level strategy can be implemented by extending the existing pipeline-driven query execution engines [31, 33], such as serializing/deserializing the intermediate data of a pipeline, and evaluating the dependency of the resumed pipeline in a query.

2.1.3 *Process-level Suspension & Resumption.*

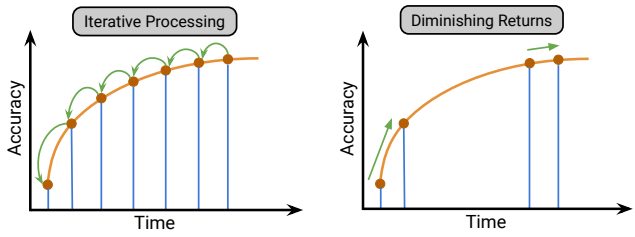
When suspending a query, the process-level strategy suspends the running process where the database and query are active and subsequently checkpoints the entire state of the process to non-volatile storage such as hard disk. This checkpointed data can be used to restore the process and resume the query from the point of suspension.

Nevertheless, this strategy can lead to an exceptionally large volume of checkpointed data, as it not only stores the state of the query and database but also captures all information required to restore the process, such as process context and system data. Moreover, this strategy is typically unable to alter the resource configuration upon process resumption since the context information, e.g., the number of threads and the amount of allocated memory, are also captured during the process suspension. The implementation and execution of the process-level strategy typically necessitate the management of operating system processes.

2.2 Resource Arbitration

Resource arbitration is a novel adaptive scheduling paradigm that can continuously preempt and reallocate resources. It is critical in Ratchet since query preemption is only worthwhile if the benefits of reallocating the preempted resources exceed the overhead of preemption and the loss associated with the preempted query, which is analogous to context switching. In comparison to classic scheduling methods, resource arbitration needs to consider various factors, such as cloud resources availability, characteristics of different queries (e.g., query completion criteria), the progress of each query, and specified users' needs.

Due to the ephemeral availability and fluctuating prices of cloud resources, resource arbitration for queries happens in an iterative way, resulting in the queries being processed iteratively either by design or inherently, as illustrated in Figure 1a. Specifically, Ratchet evaluates ongoing and preempted queries at each iteration, such as assessing the overhead associated with suspending an ongoing query and estimating the resource consumption for resuming a suspended query. These evaluations could inform the strategy for query preemption, Ratchet would opt for the redo strategy if the predicted suspension is expected to occur just moments after the execution has started or the combined cost of persisting the intermediate data and the performance overhead induced by suspension and resumption surpasses the potential loss of progress and the expense of re-running the query. Conversely, Ratchet should favor the pipeline-level suspension and resumption strategy if the estimated suspension point is following the completion of a specific pipeline or the cost of preserving the intermediate data is lower than the potential loss of progress in the current pipeline. Ratchet can also select the appropriate strategy even when the available resources for resuming the query differ from those at the time of suspension.

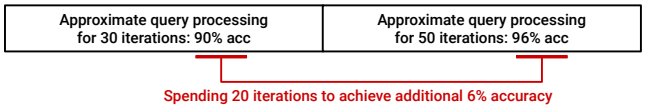


(a) Iterative processing **(b) Diminishing returns**

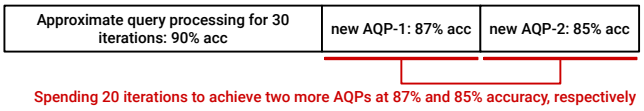
Figure 1: Iterative queries with diminishing returns

Ratchet is also designed to handle complex and heterogeneous workloads. For instance, one feature of resource arbitration is the capability to manage queries that demonstrate diminishing returns, as depicted in Figure 1b. For such queries, the query progress improvement between starting and ending points is not linear. Specifically, the first several iterations can already provide a rough idea of the final result, whereas the last iteration only contributes marginal improvements.

Thus, in certain scenarios, it is valuable to keep refining the query results by allocating sufficient resources continuously, as plotted in Figure 2a. However, in other scenarios, it may be beneficial to early-suspend queries exhibiting diminishing returns and reallocate resources to queries that promise more significant progress in a shorter timeframe. For instance, as depicted in Figure 2b, reallocating the resources to two new AQP jobs could elevate their accuracies to 87% and 85%, but only marginally improves the accuracy of the current AQP job from 90% to 96%.



(a) Keep allocating resources to the job with diminished return



(b) Re-allocating resources to new jobs

Figure 2: Motivational cases for resource arbitration

2.3 Cost Tolerance

Ratchet's cost tolerance model offers users a more granular set of cloud resource utilization and corresponding pricing options. For further elucidation, we provide an example depicted in Figure 3.

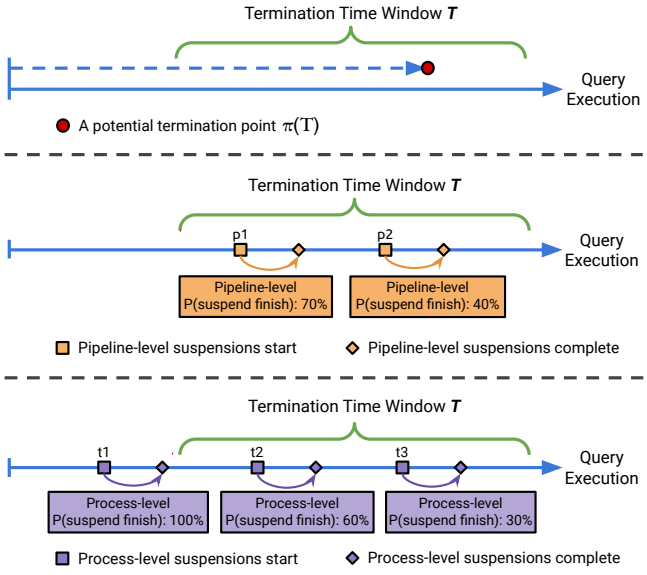


Figure 3: Query execution with potential termination using redo, pipeline-level, and process-level strategy

Consider a query q that will suspend for a potential termination point that is determined by a probability distribution π over a time window T, using three strategies: redo strategy, pipeline-level strategy, and process-level strategy.

As demonstrated in Figure 3, the redo strategy re-executes the query q when a termination occurs at $\pi(T)$, which introduces potential latency $\pi(T)$. The pipeline-level strategy enables query suspension upon the completion of each pipeline. Thus, the pipeline-level strategy can successfully suspend q at p_1 and p_2 with associated probabilities 70% and 40%, i.e., $P(p_1 + d_q^{p_2} < \pi(T)) = 70\%$ and $P(p_2 + d_q^{t_3} < \pi(T)) = 40\%$. The process-level strategy allows for the suspension of query q at multiple time points such as t_1, t_2, t_3. Suspending q at t_1 ensures a 100% probability of successful suspension and persistence of intermediate data since $P(t_1 + d_q^{t_1} < \pi(T)) = 100\%$ where $d_q^{t_1}$ is the duration of persisting the intermediate data of q at t_1 and $\pi(T)$ is the termination point based on the time window T and the distribution π. However, if the process-level strategy attempts to suspend q at t_2 or t_3, there exists a risk of complete suspension only occurring after reaching the termination point, thereby resulting in the loss of progress. Specifically, the risk of suspending at t_2 and t_3 are 60% or 30%, respectively, that is, $P(t_2 + d_q^{t_2} < \pi(T)) = 60\%$ or $P(t_3 + d_q^{t_3} < \pi(T)) = 30\%$.

A schedule can be presented with the execution options and allowed to indicate whether it accepts the suspensions at various points, e.g., t_1 or p_2. For instance, it is reasonable that a more competitive price may be available to users willing to suspend their queries at all potential points since it enhances the probability of securing available resources in the cloud environments and avoiding periods of peak utilization.

3 RESEARCH DIRECTIONS

Suspension-Oriented Databases. For environments where disruptions or terminations could be regular, a system would need suspendable and resumable queries as a first-class citizen. This necessitates a thorough revamp of numerous layers and components within existing data systems, thus opening a broad range of research opportunities.

Reshaping classic operators for query suspension and resumption is a promising research direction. Considering sort operators or sort-based operators, such as sort-merge join, maintaining cursors for sorted data is appealing, which could be an effective means of tracking the progress of query processing and reducing the cost associated with suspension and resumption. It is also challenging to redesign transaction management. For instance, introducing a new transaction status such as "suspend" or adding a new step when executing a transaction needs to be carefully considered, with the imperative of maintaining the ACID properties. Suspendable query optimization also presents a significant opportunity as it demands a balance between efficient query execution and responsive suspension. Each phase in canonical query optimization can be reconstructed to take suspension and resumption into account. For example, a suspension-oriented optimizer could transform the original plan into a suspendable one, generate an execution plan that considers potential suspension at each operator, and estimate the plan's cost by adding the underlying overhead of suspension and resumption.

Pay-As-You-Suspend. The prevalent pay-as-you-go pricing models in cloud environments typically factor in resource usage, data freshness, latency, etc. There is an inherent trade-off among these factors [1], suggesting that in certain scenarios, it is reasonable to compromise on latency to decrease costs or enhance the quality of results. Incorporating query suspension and resumption could redefine this pricing model, transitioning towards a more flexible *pay-as-you-suspend*.

Pricing models in cloud-native applications are usually influenced by various factors, with peak time resource utilization being a particularly significant yet often overlooked determinant [34]. With query suspension and resumption, users are offered more execution options to circumvent peak times, thereby reducing overall costs. Query suspension and resumption can also help the interactive query latency [32] for serverless computing to further minimize costs by identifying the most suitable suspension points while sustaining acceptable interactive latency. This allows users to obtain resources at more competitive prices without significantly compromising service quality in cloud environments.

Fine-grained and Interactive Debugging. The idea of pausing data processing for debugging purposes is intuitively attractive. One relevant work proposed a mechanism to pass messages between operator-oriented actors for pausing query execution to examine specific variables without persisting the intermediate data for suspension and resumption [21].

The query suspension and resumption introduce an enhanced, more fine-grained opportunity for debugging, particularly for long-running queries that are often challenging to troubleshoot using only final results. For example, suppose a suspension occurs at the pipeline or operator level, and the related intermediate data is serialized and persisted; this allows developers, including data engineers and scientists, to perform not only a broad verification of aggregated results but also some tuple-by-tuple checks for unobtrusive errors using the serialized intermediate data. An additional advantage of query suspension and resumption is the opportunity for interactive debugging with dynamic data. As the intermediate data are persisted, it opens up the possibility for developers to manipulate this data, such as replacing key variables or aggregated results. This capability enables proactive evaluation of the remaining segments of the query plan or execution and further allows to conduct of trial runs for swift debugging.

4 RELATED WORK

Query Scheduling. It is an integral part of the data systems deployed in cloud environments. A prime example is multi-tenant databases that permit constrained resources to cater to multiple database tenants simultaneously [14]. One of the most critical duties in multi-tenant databases is ensuring that each tenant has adequate resources to manage requests within a specific timeframe, often referred to as a Service Level Objective (SLO). Existing approaches include resource isolation, which reserves a fixed or minimal amount of required resources [30], and intelligent tenant placement [25]. However, it becomes progressively challenging to manage this duty as long-running queries become increasingly prevalent since they can saturate the virtualized resources. For example, existing methods for scheduling long-running queries typically either attempt to reserve substantial resources for these queries, potentially hastening resource saturation, or try to reposition the queries, which consequently increases their latency. Recent work [24] can preempt some long-running jobs during execution in favor of others.

The established methods always consider the cloud resources sustaining and stable, while Ratchet assumes resources are ephemeral and dynamic. Ratchet can also significantly improve existing solutions. This is because, when suspension and resumption are employed, a long-running query can be viewed as a series of smaller tasks, thereby facilitating more efficient query scheduling and resource management.

Database Migration. Cloud-native database systems are designed and built to leverage the cloud's elasticity, automation, and managed services [23]. Although scaling out cloud database services is appealing [5], database migration, which helps move users' database and analytics workloads within cloud environments, is inevitable. The state-of-the-art approach, live database migration, aims to migrate database services with minimal service interruption and negligible performance impact [36]. It primarily adopts two strategies: iterative copying [10, 12] and dual execution [15, 20]. Iterative copying, as in a similar vein to Ratchet, typically suspends all active transactions on the source during migration and iteratively transfers the database cache and the state of active transactions, thereby enabling the migration destination to commence with a hot cache [11, 35]. State migration is also common in streaming databases due to its critical role in the reconfiguration of stateful operators. One straightforward method is the "stop-and-restart", akin to the redo strategy. This method involves temporarily halting program execution, safely transferring the state during the computational pause, and restarting the job once state redistribution is complete, which commonly leverages existing fault-tolerance mechanisms within the systems [4, 6]. For many reconfiguration scenarios, only a small set of operators need to undergo state migration; operators not engaged in the migration can continue without disruption, and fault-tolerance checkpoints can be employed for state migration [16, 28]. This method can be optimized by subdividing the state and migrating the partitions [18].

Ratchet is orthogonal to live or state migration and enhances them in environments with ephemeral cloud resources. For instance, Ratchet allows query migration rather than full database migration by using the pipeline-level suspension and resumption strategy. This offers a better chance for live migration when the resources are ephemeral since the migration of queries is far less resource-intensive than relocating the full database states due to smaller intermediate states for serialization and transfer.

High Availability & Fault Tolerance. High availability refers to the database systems' ability to avoid loss of service by minimizing downtime, which is desirable for cloud infrastructure. State replication is a commonly used technique for achieving high availability [9], which typically copies all the pages of memory that change on the active host and transmits them to the backup host at each checkpoint. With modern hardware, such as RDMA (Remote Direct Memory Access), replication-based high availability can be accomplished by directly updating records on remote backup servers, bypassing the need for remote CPU involvement [38]. The recent advancement in Write-ahead Log (WAL) can provide tuple-level

granularity data version control in replication protocols, such as Paxos, for achieving high availability [39]. Ensuring fault tolerance is essential for cloud-native databases as well. To achieve this, various methods have been proposed, such as providing checkpoints at stateful operators, which combines the results from multiple data sources [3]. Another method involves partitioning the query execution graph into a collection of tasks, persisting the result of each task as it is executed [37].

Ratchet share a similar intuition; however, a distinguishing difference is that suspensions in Ratchet are either known or estimable, enabling informed decision-making in Ratchet. Nevertheless, Ratchet can be viewed as a unique resilience strategy for high availability and fault tolerance queries. It achieves this by suspending queries at various levels: data batch, operator, pipeline, and process, thereby guaranteeing query progress even in unstable environments.

5 CONCLUSION

Cloud environments with dynamic and ephemeral resources, coupled with complex and heterogeneous workloads, as well as the growing emphasis on cost-efficiency among users, present new primitives to cloud-native databases: query preemption, resource-arbitration, and cost tolerance. Making an initial step, we design Ratchet, a resource-adaptive query execution framework designed to adaptively suspend and resume queries at various granularities, arbitrating resources for the queries and offering a broader spectrum of pricing options. We also explore several promising yet challenging future research directions. Although we are in the early stages of this exciting journey, this promising field of study holds immense potential to revolutionize the efficiency and flexibility of cloud-native databases in resource-dynamic environments.

ACKNOWLEDGMENT

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

REFERENCES

[1] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Jim Chen, Min Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Adam Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Maya Meng, Prashant Mishra, Jay Patel, Rajesh Sr, Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Justin Tang, Jun Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Joey Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divy Agrawal, Jeffrey F. Naughton, Sujata Kosalge, and Hakan Hacigümüs. 2021. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. *VLDB Endowment* 14, 12 (2021), 2986–2998.

[2] Amazon EC2 Spot Instances Pricing 2023. Amazon EC2 Spot Instances Pricing. https://aws.amazon.com/ec2/spot/pricing/. Accessed: 2023-07-05.

[3] Apache Spark Streaming Checkpointing 2023. Apache Spark Streaming Checkpointing. https://spark.apache.org/docs/latest/streaming-programming-guide.html. Accessed: 2023-07-04.

[4] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *ACM International Conference on Management of Data (SIGMOD)*. 601–613.

[5] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. 2011. Hyder - A Transactional Record Manager for Shared Flash. In *Conference on Innovative Data Systems Research (CIDR)*. 9–20.

[6] Paris Carbone, Stephan Ewen, Gyula Fóra, Seif Haridi, Stefan Richter, and Kostas Tzoumas. 2017. State Management in Apache Flink®: Consistent Stateful Distributed Stream Processing. *VLDB Endowment* 10, 12 (2017), 1718–1729.

[7] Badrish Chandramouli, Christopher N. Bond, Shivnath Babu, and Jun Yang. 2007. Query suspend and resume. In *ACM International Conference on Management of Data (SIGMOD)*. 557–568.

[8] Andrew A. Chien. 2021. Driving the Cloud to True Zero Carbon. *Commun. ACM* 64, 2 (2021), 5.

[9] Brendan Cully, Geoffrey Lefebvre, Dutch T. Meyer, Mike Feeley, Norman C. Hutchinson, and Andrew Warfield. 2008. Remus: High Availability via Asynchronous Virtual Machine Replication. In *USENIX Symposium on Networked Systems Design & Implementation (NSDI)*. 161.

[10] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2013. ElasTraS: An elastic, scalable, and self-managing transactional database for the cloud. *ACM Transactions on Database Systems* 38, 1 (2013), 5.

[11] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2011. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud using Live Data Migration. *VLDB Endowment* 4, 8 (2011), 494–505.

[12] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. 2010. Live Database Migration for Elasticity in a Multitenant Database for Cloud Platforms. *UCSB Computer Science Technical Report* (2010).

[13] Kayhan Dursun, Carsten Binnig, Ugur Çetintemel, Garret Swart, and Weiwei Gong. 2019. A Morsel-Driven Query Execution Engine for Heterogeneous Multi-Cores. *VLDB Endowment* 12, 12 (2019), 2218–2229.

[14] Aaron J. Elmore, Carlo Curino, Divyakant Agrawal, and Amr El Abbadi. 2013. Towards Database Virtualization for Database as a Service. *VLDB Endowment* 6, 11 (2013), 1194–1195.

[15] Aaron J. Elmore, Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2011. Zephyr: live migration in shared nothing databases for elastic cloud platforms. In *ACM International Conference on Management of Data (SIGMOD)*. 301–312.

[16] Raul Castro Fernandez, Matteo Migliavacca, Evangelia Kalyvianaki, and Peter R. Pietzuch. 2013. Integrating scale out and fault tolerance in stream processing using operator state management. In *ACM International Conference on Management of Data (SIGMOD)*. 725–736.

[17] Panagiotis Garefalakis, Konstantinos Karanasos, Peter R. Pietzuch, Arun Suresh, and Sriram Rao. 2018. Medea: scheduling of long running applications in shared production clusters. In *European Conference on Computer Systems (EuroSys)*. 4:1–4:13.

[18] Moritz Hoffmann, Andrea Lattuada, Frank McSherry, Vasiliki Kalavri, John Liagouris, and Timothy Roscoe. 2019. Megaphone: Latency-conscious state migration for distributed streaming dataflows. *VLDB Endowment* 12, 9 (2019), 1002–1015.

[19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, João Carreira, Karl Krauth, Neeraja Jayant Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. 2019. Cloud Programming Simplified: A Berkeley View on Serverless Computing. *CoRR* abs/1902.03383 (2019).

[20] Junbin Kang, Le Cai, Feifei Li, Xingxuan Zhou, Wei Cao, Songlu Cai, and Daming Shao. 2022. Remus: Efficient Live Migration for Distributed Databases with Snapshot Isolation. In *ACM International Conference on Management of Data (SIGMOD)*. 2232–2245.

[21] Avinash Kumar, Zuozhi Wang, Shengquan Ni, and Chen Li. 2020. Amber: A Debuggable Dataflow System Based on the Actor Model. *VLDB Endowment* 13, 5 (2020), 740–753.

[22] Viktor Leis, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *ACM International Conference on Management of Data (SIGMOD)*. 743–754.

[23] Feifei Li. 2019. Cloud native database systems at Alibaba: Opportunities and Challenges. *VLDB Endowment* 12, 12 (2019), 2263–2272.

[24] Rui Liu, Aaron J. Elmore, Michael J. Franklin, and Sanjay Krishnan. 2023. Rotary: A Resource Arbitration Framework for Progressive Iterative Analytics. In *IEEE International Conference on Data Engineering (ICDE)*. 2140–2153.

[25] Ziyang Liu, Hakan Hacigümüs, Hyun Jin Moon, Yun Chi, and Wang-Pin Hsiung. 2013. PMAX: tenant placement in multitenant databases for profit maximization. In *International Conference on Extending Database Technology (EDBT)*. 442–453.

[26] Zhenxiao Luo, Lu Niu, Venki Korukanti, Yutian Sun, Masha Basmanova, Yi He, Beinan Wang, Devesh Agrawal, Hao Luo, Chunxu Tang, Ashish Singh, Yao Li, Peng Du, Girish Baliga, and Maosong Fu. 2022. From Batch Processing to Real Time Analytics: Running Presto® at Scale. In *IEEE International Conference on Data Engineering (ICDE)*. 1598–1609.

[27] Arlino Magalhães, José Maria Monteiro, and Angelo Brayner. 2022. Main Memory Database Recovery: A Survey. *ACM Computing Survey* 54, 2 (2022), 46:1–46:36.

[28] Luo Mai, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, Saravanam Muthukrishnan, Vamsi Kuppa, Sudheer Dhulipalla, and Sriram Rao. 2018. Chi: A Scalable and Programmable Control Plane for Distributed Stream Processing Systems. *VLDB Endowment* 11, 10 (2018), 1303–1316.

[29] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems* 17, 1 (1992), 94–162.

[30] Vivek R. Narasayya, Sudipto Das, Manoj Syamala, Badrish Chandramouli, and Surajit Chaudhuri. 2013. SQLVM: Performance Isolation in Multi-Tenant Relational Database-as-a-Service. In *Conference on Innovative Data Systems Research (CIDR)*.

[31] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *Conference on Innovative Data Systems Research (CIDR)*.

[32] Matthew Perron, Raul Castro Fernandez, David J. DeWitt, and Samuel Madden. 2020. Starling: A Scalable Query Engine on Cloud Functions. In *ACM International Conference on Management of Data (SIGMOD)*. 131–141.

[33] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *ACM International Conference on Management of Data (SIGMOD)*. 1981–1984.

[34] Hossein Shafiei, Ahmad Khonsari, and Payam Mousavi. 2022. Serverless Computing: A Survey of Opportunities, Challenges, and Applications. *ACM Computing Survey* 54, 11s (2022), 239:1–239:32.

[35] Xingda Wei, Sijie Shen, Rong Chen, and Haibo Chen. 2017. Replication-driven Live Reconfiguration for Fast Distributed Transaction Processing. In *USENIX Annual Technical Conference (USENIX ATC)*. 335–347.

[36] Eugene Wu, Samuel Madden, Yang Zhang, Evan Jones, and Carlo Curino. 2010. Relational cloud: The case for a database service. *MIT CSAIL Technical Report* (2010).

[37] Zhicheng Yin, Jin Sun, Ming Li, Jaliya Ekanayake, Haibo Lin, Marc T. Friedman, José A. Blakeley, Clemens A. Szyperski, and Nikhil R. Devanur. 2018. Bubble Execution: Resource-aware Reliable Analytics at Cloud Scale. *VLDB Endowment* 11, 7 (2018), 746–758.

[38] Erfan Zamanian, Xiangyao Yu, Michael Stonebraker, and Tim Kraska. 2019. Rethinking Database High Availability with RDMA Networks. *VLDB Endowment* 12, 11 (2019), 1637–1650.

[39] Jianjun Zheng, Qian Lin, Jiatao Xu, Cheng Wei, Chuwei Zeng, Pingan Yang, and Yunfan Zhang. 2017. PaxosStore: High-availability Storage Made Practical in WeChat. *VLDB Endowment* 10, 12 (2017), 1730–1741.