Starling: A Scalable Query Engine on Cloud Functions

Matthew Perron
MIT CSAIL
mperron@csail.mit.edu

David DeWitt
MIT CSAIL
david.dewitt@outlook.com

Raul Castro Fernandez
University of Chicago
raulcf@uchicago.edu

Samuel Madden
MIT CSAIL
madden@csail.mit.edu

ABSTRACT

Much like on-premises systems, the natural choice for running database analytics workloads in the cloud is to provision a cluster of nodes to run a database instance. However, analytics workloads are often bursty or low volume, leaving clusters idle much of the time, meaning customers pay for compute resources even when underutilized. The ability of cloud function services, such as AWS Lambda or Azure Functions, to run small, fine granularity tasks make them appear to be a natural choice for query processing in such settings. But implementing an analytics system on cloud functions comes with its own set of challenges. These include managing hundreds of tiny stateless resource-constrained workers, handling stragglers, and shuffling data through opaque cloud services. In this paper we present Starling, a query execution engine built on cloud function services that employs a number of techniques to mitigate these challenges, providing interactive query latency at a lower total cost than provisioned systems with low-to-moderate utilization. In particular, on a 1TB TPC-H dataset in cloud storage, Starling is less expensive than the best provisioned systems for workloads when queries arrive 1 minute apart or more. Starling also has lower latency than competing systems reading from cloud object stores and can scale to larger datasets.

CCS CONCEPTS

• **Information systems** → *Relational parallel and distributed DBMSs*; *Online analytical processing engines*; • **Computer systems organization** → *Cloud computing*;

1 INTRODUCTION

Modern organizations are increasingly turning to cloud providers to run their data services, including database analytics workloads with services like Amazon Redshift or Azure SQL Data Warehouse. These cloud systems are more elastic than on-premises alternatives and avoid their upfront costs. Yet these systems still require users to make complex provisioning decisions. But because many analytics workloads are unpredictable and ad-hoc, provisioning well is difficult, resulting in over-provisioned hardware. Although some cloud services provide "elastic" features that allow compute nodes to be added or removed dynamically, this scaling can take minutes, making it impractical on a per-query basis. Further, many cloud database systems require data to be explicitly loaded into proprietary formats on local disks to perform well. For workloads composed of one-off queries or ETL queries, loading data results in an unacceptable increase in query latency. Furthermore, cloud storage tends to be an order of magnitude cheaper than other storage services. As a result, several systems, including Presto [25] and Athena [1], are purpose built for executing queries directly on cloud storage. Other systems like Redshift [7] have special mechanisms for reading from cloud storage.

In contrast to current offerings, an ideal system would avoid pre-provisioning, charge users query-by-query, and be performance competitive. It would also avoid loading data into proprietary formats, and let users tune to cost and performance needs on a query-by-query basis. Although achieving all of these goals perfectly and simultaneously is not possible, so called "serverless" cloud function services, like AWS Lambda [13] and Azure Functions [14], offer a tantalizing promise that suggests they may be able to get close. In particular, these services allow arbitrary numbers

System	Does not require loading	Pay by query	Tunable performance
Amazon Athena	✓	✓	✗
Snowflake	✗	✓*	✓
Presto	✓	✗	✓
Amazon Redshift	✗	✗	✓
Redshift Spectrum	✓	✗	✓
Google BigQuery	✓	✓	✗
Azure SQL DW	✓	✗	✓
Starling	✓	✓	✓

Table 1: Comparison of cloud analytics databases

of small tasks to be invoked with very low startup times (typically a few milliseconds) and offer virtually unlimited parallelism. Users are charged only for the execution time used, typically at the granularity of 1 second or less. Using such tasks, one could invoke many small parallel jobs to scan, join, and aggregate tables in raw cloud storage using well known techniques from parallel databases to implement a SQL query processing system.

However, using function services to support ad-hoc analytical workloads comes with its own set of hurdles. First, workers or functions have limited memory, execution time limits, and networking restrictions that prevent sending data directly from one instance to another. Thus, in order to support shuffles, function services require other methods of moving data between instances. In addition, instances are typically stateless, which is at odds with stateful analytical queries that need to shuffle or aggregate data. Finally, latency of individual workers can be unpredictable, leading to *stragglers* taking much longer to run than other workers on data of similar scale; this is particularly true when workers use proprietary, closed-source, and otherwise opaque cloud storage services to exchange state, as these services often yield variable and unpredictable latencies.

To explore the promise of function services for database analytics, we built Starling, a query execution engine that runs on serverless platforms. Starling leverages the benefits of cloud services while mitigating the above challenges. To achieve high resource utilization, Starling maps tasks to function invocations so users pay for only the compute resources their query actually uses. The number of invocations can grow and shrink as needed during each query execution. Starling takes advantage of the on-demand elasticity of cloud object storage services, such as Amazon S3[9] to shuffle data. It materializes intermediate results in a format optimized to reduce cost while achieving high aggregate throughput. To mitigate stragglers, Starling uses a tuned model to detect straggling requests and mitigates their impact on query latency. Finally, Starling provides opportunities to optimize

queries for cost or latency by adjusting the number of invocations at each stage. Tuning queries to cost or performance is desirable for users who run ad-hoc workloads.

With these optimizations, Starling achieves query latency comparable to provisioned systems while decreasing cost for workloads with moderate query volume.

We begin by exploring the properties of available tools for analytical workloads, and show that Starling fits a point in the design space that has so far remained unaddressed.

2 MOTIVATION AND DESIGN

Starling seeks to provide a balance of performance, flexibility, and low cost-per-query that current systems do not provide for some important classes of workloads. Below we describe the current landscape of systems and describe the promise and challenges of cloud functions for query processing. We follow with a brief description of the architecture of Starling, and conclude by discussing why we chose Amazon AWS to for our Starling implementation.

2.1 Landscape of Cloud Analytic Databases

The proliferation of cloud analytic databases has led to a rich ecosystem of offerings with varying features and pricing models. Table 1 shows an overview of the design space. The rows in the table correspond to some of the most popular analytic databases. The columns are as follows:

• **Does not require loading** Some systems need to load data from its original format, e.g. CSV, ORC, Parquet, etc., to an internal format that permits executing queries with high performance. This loading step is a barrier to users who want to run ad-hoc queries on raw files in cheap cloud object storage. With the rise of enterprise data lakes, having analytic data stored in raw formats in inexpensive cloud object stores is increasingly the norm. While many systems have methods of reading data from external sources like cloud storage, this is typically a tacked-on option that results in significant performance degradation compared to data stored on local disks in native formats.

• **Pay-per-Query** Provisioned systems start a cluster that sits in the cloud waiting for queries to process. Whether the system is idle or not, the cloud vendor charges for the underlying virtual machines (plus some fixed cost for the data analytics service). In a *pay-per-query* model, users are only charged based on the queries they run. Such a model can be dramatically cheaper if queries are issued sporadically or unpredictably. We evaluate specifically when pay-per-query is more cost-efficient than Starling in our experiments. While Snowflake does not have a pay-per-query model per se (hence the asterisk in the table), it allows users to automatically shut down clusters during periods of inactivity, and resume processing when new queries arrive.
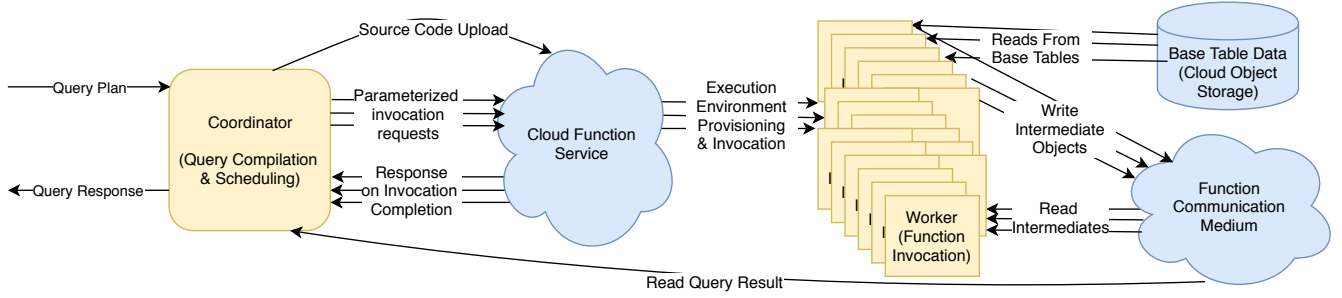
Figure 1: Query Execution in Starling. Opaque cloud components in blue, Starling components in yellow

• **Tunable Performance** In cloud settings, both the response time and cost of executing a query depends on the amount of resources that are provisioned. When systems permit scaling resources to tune performance and cost in response to data volumes, then we say these systems are elastic. When systems are inelastic or do not allow tuning, queries may fail to execute or take longer than users require.

Starling is a system for analytic users who run a low to moderate query volumes on data in cloud object storage. It i) does not require loading; ii) charges only for the queries that are executed; and iii) permits users to trade off cost and performance, and adjust parallelism on a per-query basis.

No existing system offers all three options. Cloud functions are the building block allowing us to achieve these goals.

2.2 Cloud Functions

Cloud function services allow users to run applications without managing or provisioning servers. Users upload application code or executables to the service. In response to events or direct user invocation, the function service provisions an execution environment and runs the user-provided code. For our purposes, the key advantage of cloud functions are i) they can read directly from cloud storage, ii) they have a very low startup time and are billed on a per-invocation basis and iii) many of them can be invoked in parallel. These properties translate directly to the desired features in Starling (no loading, pay-per-query, and tunable parallelism/performance.)

Despite their high-level appeal, analytical workloads are not a natural fit for cloud functions for several reasons. First, analytic queries can run for hours, but cloud function execution is limited to a few minutes. Moreover, cloud functions execute in resource-constrained containers; e.g., 1 core per function and at most 3 GB RAM is typical. Second, analytic queries require shuffling data to compute joins, but cloud functions do not allow communication between invocations. A string of recent research has shown cloud functions are useful for a number of data parallel workloads [11, 17, 18, 21], but for workloads with more complex communication patterns like analytical workloads, the limitations of cloud functions have been identified as show-stoppers [20, 26]. In this

paper we develop mechanisms to work around these shortcomings and deliver a performance and price-competitive data analytics system built on cloud functions.

2.3 Starling Architecture

Starling is a query execution engine. Users submit planned queries to the system and receive back query results. Figure 1 shows Starling's architecture. As seen in the figure, users submit queries to a small coordinator that compiles the query and uploads it to a cloud function service. The coordinator then schedules the tasks by invoking them through the function service. The function service is responsible for provisioning execution environments for workers that perform the task of query execution. Workers read base table data from inexpensive cloud object storage services. Because functions are stateless, they exchange state through a communication medium, e.g., shared storage. When all tasks complete the worker reads the result from the function communication medium and returns it to the user.

Starling's design requires a few properties of underlying cloud services. First, Starling needs to launch hundreds of function invocations at once and in parallel from cold start. Second, Starling relies on relatively inexpensive and high throughput methods of exchanging data between tasks, such as object storage services. Its performance depends on individual parallel workers being able to each achieve high throughput despite other workers executing concurrently.

2.4 Choosing a Cloud Function Service

Google Cloud Functions [19], Azure Functions [14], and AWS Lambda [13], each provides a function service. Starling's architecture can be implemented in any of these platforms, however, certain platforms offer features that are more amenable to the kind of workloads Starling is designed to support. In particular, Google Cloud Functions and Azure functions have restrictions on the languages that can be used, or the rate at which functions can be invoked, or both, that will impact parallel query processing performance, while AWS Lambda does not have such restrictions.

For these reasons we chose to build Starling on AWS Lambda. This restricted us to using AWS services to exchange intermediate data. In the next section we discuss why we chose S3 [9], AWS's cloud object storage offering, for both base table storage and as the communication medium.

3 MANAGING DATA IN STARLING

Storage is an important component of any data management system. Starling provides interactive query performance on raw data stored in cloud object storage. Starling does not manage base table data, but must interact with it efficiently, as we describe below in Section 3.1. Because cloud functions are stateless, Starling must also manage intermediate state during query execution as described in Section 3.2.

3.1 Base Table Storage

Starling executes queries over data sitting in S3. Starling's design is agnostic to base table formats, but common choices are CSV, ORC, and Parquet. Starling requires only that rows of a specified schema can be parsed from the source objects in S3; however, for the best performance base table data should be stored as objects of a few hundred MB.

Open source columnar formats like ORC [12] help Starling to achieve good performance as they allow reading a subset of columns rather than the whole row. Skipping unneeded columns decreases latency of tasks and the cost of running workers. ORC also includes indexes and basic statistics that allow users to skip portions of the input increasing performance. Therefore, in our evaluation in Section 6 we use raw data stored in ORC.

3.2 Intermediate State

As cloud functions are stateless and have no method of communicating directly, Starling relies on AWS services to shuffle data. A medium for exchanging data between function invocations should have low cost, high throughput, low latency, and scale transparently. We considered several options for exchanging intermediate state before choosing Amazon's cloud object storage service, S3. Using virtual machines or a streaming system like Amazon Kinesis [6] both require users to provision capacity ahead of time and thus are not a suitable choice. We also considered using queue services like Amazon SQS [10], but these limit message sizes (to 256KB in the case of SQS) and require encoding data as text, making it cumbersome and computationally costly for large shuffles. NoSQL services like DynamoDB [3] have very low latency but unacceptably high cost for large shuffles. While S3 has high latency compared to some of these alternatives, this can be mitigated as we describe in Section 3.3. We describe the most important properties of S3 for Starling below.

S3 Properties: S3 [9] is an AWS's object storage service. Users write binary objects of arbitrary size to the service into "buckets" with a named "key". S3 is a write-once system, meaning that no appends or updates to objects are allowed, only replacement. Users can scale read and write throughput by ensuring keys are spread across "prefixes", defined as the first few characters in the key [15]. Users interact with the service through a REST interface. Reads from S3 can fetch the entire object or a range of bytes. S3 charges users by the amount of data stored at a rate $0.23 per GB per month, and a cost of $0.0004 per thousand GET requests of any size, and $0.005 per thousand PUTs (prices as of July 2019). Unlike standard file systems, S3 does not guarantee read-after-write consistency, which complicates query processing as we describe in Section 3.3.1. S3 provides atomic reads and writes, ensuring that readers never see data from two separate writes in the same read.

Sharing intermediates: Starling uses S3 to pass intermediate data between function invocations. Workers write their outputs as a single object in S3 with a predetermined key. Because the object is written at a known location, readers can poll the object key until the object appears. For query processing, S3 has the additional advantage of being persistent, meaning workers can begin sending data before destination workers have started executing. This saves a significant amount of execution time in AWS Lambda.

Using S3, one-to-many communication is both inexpensive and straightforward. Producer tasks write an object to S3, making it visible to all readers that need it. All-to-all communication, as in a shuffle, is more difficult to achieve at low cost. As recent work has demonstrated, writing an object per partition for large shuffle incurs unacceptably high cost [26]. Starling ameliorates this problem by writing a single partitioned file to S3 and having consumer tasks read only the relevant portion of each object output by the producers. We describe this process in more detail in Section 4.2. More details of the on-disk format of Starling's intermediate state may be found in the extended paper [24].

3.3 Mitigating High Storage Latency

While S3 has very high aggregate throughput, it has much higher latency than other shuffling options. A 256KB read has median latency of 14ms. If workers perform single-threaded blocking reads to S3, they may sit idle while waiting for responses, increasing query latency and thus increasing invocation runtime costs. In order to mitigate this latency each task performs several reads in parallel from S3. Fortunately, most tasks in Starling must perform many reads, making this trivial to parallelize. Columnar formats like ORC are broken into column segments, and in shuffles, tasks must

make many reads to fetch partition data from objects written by input tasks. Parallelizing reads helps keep tasks busy, spending less time idle and more time on query processing. As we increase the number of parallel reads that the invocation performs, we achieve higher effective throughput until we achieve 160MBps at 16 parallel reads, after which adding more parallelism does not improve throughput.

3.3.1 Mitigating Object Visibility Latency. As noted above, S3 does not guarantee read-after-write consistency in some cases. As a result, an object recently written to S3 may not become visible to tasks in a subsequent stage for several seconds or more. Although infrequent, any delay in object visibility slows down all reading workers and can have detrimental impact on query latency because a shuffle is an all-to-all communication. Furthermore, tasks reading the object continue to incur delays waiting for it to become readable. Starling mitigates this risk by writing the same object to two different keys in S3. We call this optimization "doublewrite". Consumers attempt to read the first key, and if it is not available, try the second one. This strategy makes query performance more predictable by reducing the risk that a single visibility issue slows down all consumers.

4 QUERY EXECUTION

The goal of Starling's query execution engine is to achieve interactive performance at low cost. To run queries on AWS Lambda, Starling's coordinator uses a single JSON file describing a physical query plan as input. The plan contains dependencies between stages, and the number of tasks within each stage. The coordinator monitors task completion and starts new stages once dependencies are completed. The coordinator generates C++ source code for the query, and compiles it into a single executable that is then packaged with necessary dependencies, compressed in an archive and uploaded to AWS Lambda. Each task's input and output object names are determined before each stage begins execution. Each task executes as much of the query as possible without communicating with other workers. For instance, Starling's workers may perform multiple joins if tables are partitioned on the same key. The coordinator then invokes query tasks until query completion. Below we describe how relational operators are implemented, how shuffles are performed, and how tasks are scheduled to balance performance and cost.

4.1 Relational Operator Implementation

After reading data from S3, workers in Starling execute using data-centric operations, where operators are implemented as a series of nested loops, rather than executing an operator-at-a-time. Query compilation allows for type specialization and achieves very good performance for analytics use cases systems [22]. Essentially each task contains a set of nested
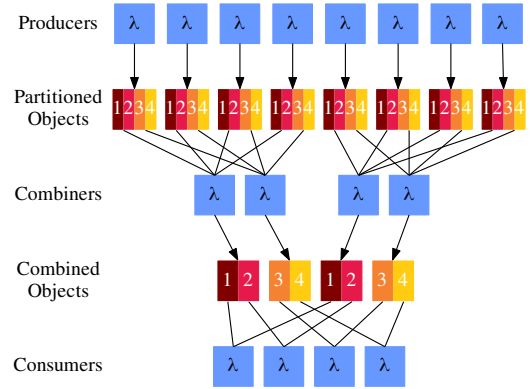
Figure 2: Starling multistage shuffle, function executions in blue, S3 Objects in shades of red showing partitions. Lines are reads and arrows are writes

loops each performing necessary relational operations. This pipelining of operations within workers contributes to Starling's low query latency. The tasks write the materialized output of its operations as a single object to S3.

Below we describe how relational operators are implemented in Starling.

Base Table Scans. While Starling does not manage base table data directly, it still must be able to scan data from base tables quickly. It does this by reading portions of input files in parallel. If there is a projection, Starling reads only the necessary columns from the base table, if possible given the file format (e.g., in ORC or Parquet).

Joins. Starling supports both broadcast joins and partitioned hash joins. In the case of broadcast joins, each input task for the inner relation writes a single object to S3. On the outer relation, tasks read all data from inner relation and their own subset of the outer relation to perform the join.

Partitioned hash joins require a shuffle. Tasks scan both relations and partition their data on the join key and write partitioned files. If possible, this partitioning is pipelined in a single task with other operations. Afterwards, a set of join tasks is started to perform the join. These join tasks create a hash table for their partition of one relation and then scan the other relation, probing into the hash table. As shuffling efficiently is critical for performing partitioned joins, we describe in detail how shuffles are performed in Section 4.2.

Aggregation. Starling performs aggregation in two steps. Tasks that perform the final operation before aggregation each generate a set of partial aggregates and output an object to S3. To complete the query, a final task reduces these partial aggregates into a final aggregate. When necessary, Starling first performs a shuffle, partitioning on the `group by` key before generating partial aggregates.

4.2 Shuffling

Starling uses a partitioned intermediate format allowing tasks performing a shuffle to read only relevant partitions from input files. In a standard shuffle, each consumer must read from every output, an all-to-all communication. Since the workers in Starling are small, each task must read a large number of small objects from the storage service. In addition to impacting query latency, many reads may incur an unacceptably high cost since object storage services charge by request. For example, with 512 producer tasks and 128 consumer tasks, the S3 cost for this shuffle is only 5.7 cents at current S3 pricing, but for a larger shuffle of 5120 producers and 1280 consumers, the cost increases to more than $5.

To address this issue, we use 2 strategies for partitioned hash joins. For small joins we use a standard shuffle as described above where every consumer reads output from every producer task. The number of reads is $2sr$ where s is the number of producers and r is the number of consumers.

Since request costs become unacceptably high for joins with many input tasks, we can trade off compute time for object storage request costs by implementing a multi-stage shuffle. We introduce a stage of combining tasks between the producers and consumers. Each task in the combining stage reads a contiguous subset of partitions from a subset of input objects and produces a single combined output, with the same partitioned file format. Because these combining tasks read contiguous partitions, they still perform only two reads per input. Finally, the consumer tasks read outputs written by relevant combiners. Because each combining task reads a subset of partitions, read outputs need only read a subset of the outputs of these combining tasks. A diagram of this approach is shown in Figure 2. The number of S3 reads in a multistage shuffle is $2(s/p + r/f)$ where p is the fraction of partitions each combiner reads and f is the fraction of files each combiner reads. The number of combining tasks is $1/(pf)$. Figure 2 shows a multi stage shuffle with $p = 1/2$ and $f = 1/2$, where each combining task reads half of the partitions from half of the input files. With 5120 and 1280 consumer tasks and $p = 1/20$ and $f = 1/64$, the S3 read cost is just $0.073 compared to more than $5 with a standard shuffle. The additional write cost of these combiners is negligible. Each of the 1280 combiners makes two additional writes, costing an additional $0.00128.

In a multistage shuffle, we can create any number of combining tasks, but we typically choose the same number of combining tasks as receiving tasks.

4.3 Assigning Tasks to Workers

The primary way that cost and performance is managed in Starling is by controlling the number of tasks per stage. Typically having more tasks per stage results in lower latency but higher cost because of the overhead of exchanging intermediate state between workers. However, this is a delicate balancing act. With too many workers overheads may overwhelm any potential performance gains. But with too few tasks, resource constrained workers may run out of memory. The space in between these two extremes allows users of Starling to trade off cost and performance by tuning the number of workers per stage.

For large queries, we sometimes need to execute more tasks than the maximum available parallel function invocations, as Amazon imposes a limit on the total tasks that can be executed at once. In July 2019 the limit on parallel AWS Lambda function invocations was 1,000. This can be increased by contacting Amazon. For our experiments we set maximum number of parallel invocations to 5,000. The coordinator sets a maximum limit on the number of parallel tasks per stage, Once the limit is reached, Starling waits for a task to finish before scheduling a new task, until all stages are complete.

Because Starling does not currently have a query optimizer, we expose user-configured parameters necessary to tune for cost and performance, including the shuffling strategy and the number of tasks per stage.

4.4 Pipelining

Instead of waiting for all tasks in a stage to be complete before starting consuming stages, a strategy for decreasing the latency of queries is to start consuming stages when a large fraction of producer inputs is available. This allows workers to start reading the available inputs and decreases overall query latency by mitigating the impact of some stragglers. However, this comes with additional risks and costs. If a task in the producer stage straggles during a standard shuffle, it causes all reading tasks to sit idle, significantly increasing the cost of executing this stage. Thus, turning on pipelining typically results in lower query latencies, but comes with additional cost. Users who desire the least expensive query execution possible should disable pipelining. One way to reducing this cost is by eliminating as many stragglers as possible with straggler mitigation techniques.

5 STRAGGLERS

Starling relies on S3 for both reading base table data as well as for exchanging intermediate state between function invocations. As stages must wait for their inputs to be available before doing the computational work of query processing, stragglers in any of these requests can have significant impact on query latency. Unfortunately, S3 requests often suffer from poor tail latency, with a small fraction of reads and writes taking much longer to complete. Thus, straggler mitigation is critical in making Starling performance competitive
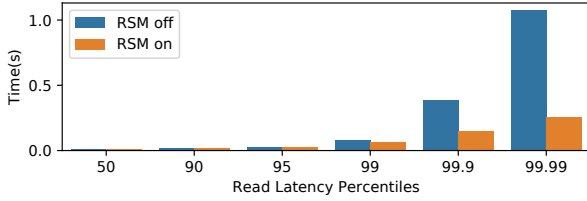
Figure 3: Read latency percentiles for 256KB reads to S3 from AWS Lambda. Comparing RSM off and on

with provisioned systems. One of the chief challenges of mitigating stragglers is that these services are outside the control of Starling, are closed source, and have opaque operation. Therefore, we base our optimizations on the power of two choices. [23], a well-known theoretical framework for using randomization and duplicate tasks to improve performance in unpredictable distributed systems, which systems like MapReduce and Spark also employ to good effect.

5.1 Read Straggler Mitigation

A single query in Starling may perform hundreds of thousands of S3 GET requests. Some of these requests experience significant delays, as we show in Figure 3. We mitigate these stragglers by observing how long a request takes compared to its expected completion time. Workers detect when a request is taking longer than expected, open a new connection to S3, and retry the request. Workers use a simple model to determine when queries should arrive based on observed latency and throughput of S3 requests as well as the throughput of AWS Lambda Invocations. Starling's model for expected query response time is given by: $r = l + (b/tc)$, where r is the expected response time, b is the number of bytes requested, and c is the number of concurrent readers. The tunable parameters of the model l and t, correspond to latency and throughput of AWS Lambda invocations reading from S3. We measure these as 15ms and 150MBps respectively.

If S3 fails to respond to a request within a fixed factor of the expected time, Starling sends a duplicate request, accepts whichever response returns first, and closes the other connection. While we do not have insight into the design of S3 and thus cannot determine the source of these stragglers, we find that this strategy mitigates most read stragglers and significantly improves query latency.

To show this, we evaluate Starling's read mitigation strategy using a microbenchmark, performing thousands of reads to S3 from Lambda. In Figure 3 we show the tail latencies of the request time for a set of 256KB reads with and without read straggler mitigation (RSM) enabled. While the straggler mitigation mechanism is not perfect and sometimes requests take as long as 2.5 seconds, the long tail of requests seen without this mitigation is cut short, helping queries to
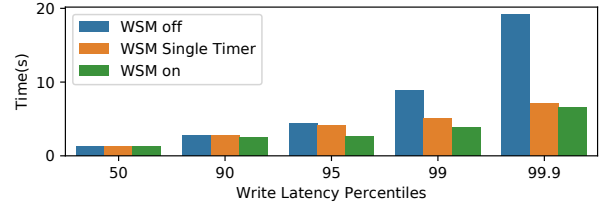
Figure 4: Write latency percentiles for 100MB writes to S3 from AWS Lambda. Comparing WSM off, with a single timeout, and fully on

run faster. At the 99.99th percentile, latency is more than a second without RSM and .25 seconds with RSM.

Although duplicate requests result in an additional expense, this pays off in saved function invocation time. An additional read request needs to save just 8 milliseconds of invocation time to pay for itself. While the read straggler mitigation mechanism is only triggered in 0.3% of cases (160 times for 52,000 reads), in this experiment the mitigation saves nearly 95 seconds of compute time at an additional read cost the equivalent of only 1.3 seconds, making read straggler mitigation a cost saving measure as well.

5.2 Write Straggler Mitigation

While most queries perform several orders of magnitude fewer writes to S3 than reads, usually two per function invocation, write requests tend to be much larger, up to several hundred MB. Median latency for these large write requests may be several seconds. Write stragglers must be handled differently. While read requests are small and their responses are large, the inverse is true for writes. Furthermore, we observed that most stragglers on write are not due to slow data transmission rates to S3, but in the S3 service processing requests and sending responses. That is, write requests are sent to the S3 service quickly, but replies from S3 may be delayed for unknown reasons. By using a strategy similar to RSM, Starling may react slowly to cases where data is written to S3 quickly but S3 is sluggish in its response. Therefore, we use an additional model to predict response times for writes *once the request has completed sending*. The expected response time is the same as the RSM model but requires a different set of parameters since the internal throughput of the S3 service appears to be much higher than the throughput from a single function invocation. When either of these models for response time indicate that a straggler has occurred, a second write request is started on a new connection.

We determine how well the write straggler mitigation (WSM) mechanism works by running another microbenchmark. Figure 4 shows a comparison of the tail latency of many 100MB writes to S3. We compare the tail latency of response times without write straggler mitigation, with only a single timeout (the same as read straggler mitigation) and

with the full write straggler mitigation including a second timeout set after the client finishes sending its request. Without write straggler mitigation, the longest writes take more than 20 seconds. While the single timeout, reduces these worst performing reads to about 18 seconds, the full write straggler mitigation brings the tail latency down to about ten seconds. As writes are much less frequent than reads in Starling, this long tail is encountered much less frequently. At the 99th percentile writes take almost 9 seconds without WSM, 5 seconds with only one timeout, and 3.8 seconds with full write straggler mitigation.

To pay for itself, each additional write would need to save 102 milliseconds of compute time. In this experiment, the full write straggler mitigation is invoked in 31% of writes, 3138 of 10240 writes, costing the equivalent of 314 seconds of compute, while saving the equivalent of 2100 seconds.

Since all writes have to complete for following stages to read them, WSM is a critical part of Starling achieving low latency while also saving compute time.

6 EVALUATION

In our evaluation we seek to answer the following questions in corresponding sections:

- How does Starling's operational cost compare to alternatives as query workloads change? (Section 6.2)
- How performant is Starling compared to alternatives? (Section 6.3)
- How well does Starling scale to larger datasets? (Section 6.4)
- How does Starling compare to current cloud interactive pay-by-query services? (Section 6.5)

Additional experiments can be found in the extended paper [24].

6.1 Experimental Setup

We execute our experiments on a scale factor 1,000 (1TB) TPC-H [16] dataset for most experiments, and scale factor 10,000 (10TB) for the scaling experiment. Each uncompressed table is broken into files of size at most 1GB, then encoded using Apache ORC [12], a standard columnar format. The ORC files use Snappy compression [27] to compress columns. The resulting ORC files are uploaded to a single Amazon S3 [9] bucket. The size before compression is about 1TB and the size after conversion to ORC is 317GB. This dataset has no skew. We compare to other systems on a subset of TPC-H queries: all queries except 11, 21, and 22. We exclude these queries as Starling is not yet support features required to execute them. We do not have reason to suspect that the addition of these queries would substantially change our results. For each configuration we execute each query sequentially, and report the median latency of three executions.

We describe each of the systems and configurations that we compare against below.

Amazon Redshift. Amazon Redshift [7] is a data warehouse product sold by Amazon Web Services(AWS). Users provision a cluster with a fixed number of nodes, though nodes can be added later. Users must first pre-load data before it can be queried. Redshift offers two large node types: A "dense compute" node type dc2.8xlarge with 32 threads, 244GB of DRAM and 2.56TB of SSD storage, and a "dense storage" node type ds2.8xlarge with the same CPU and memory, but with 16TB of HDD storage. Redshift also allows users to read data from external tables in S3 with a feature called Spectrum. Instead of performing reads directly from the user's cluster, Spectrum spawns short lived workers, much like AWS Lambda invocations, to filter base table data from S3 and send it to the cluster to complete the query.

Redshift allows users to divide tables among nodes on a distribution key and sort on a key by defining these keys in the schema. Doing so significantly decreases query latency when tables are distributed on join keys and sorted, as nodes can perform local merge joins without shuffling data.

We compare against four configurations of Redshift reading from local data, and one reading from S3 using Spectrum. With local data we have two clusters, one with "Dense compute" nodes with local SSDs and one with "Dense Storage" with 16TB of HDD storage each, abbreviated dc and ds respectively. We also compare two different schemas, one with distribution keys and sorting keys defined, and one without where data is equally partitioned among nodes without respect to their join key and sort order. The configurations on the dense compute cluster with distribution keys we call redshift-dc-dk and with a without distribution keys we call redshift-dc-dd. Likewise the dense storage configurations we call redshift-ds-dk and redshift-ds-dd for distribution keys and default distribution, respectively.

In addition, we compare with a four node dc2.8xlarge cluster using the Spectrum feature to read all base table from S3. We call this configuration spectrum.

We report the cost of running Redshift's nodes using on-demand pricing [8] for all configurations and add S3 scans costs for the Spectrum configuration.

Presto. Like Starling, Presto is a SQL execution engine designed to execute on data on "in situ", on raw storage, rather than having users load data into specialized formats. We use a cluster of r4.8xlarge nodes to execute queries query. Each node has 32 threads and 244GB of DRAM. We used Amazon EMR [5] 5.24.1 to set up Presto [25] 0.219. Since we could have set this cluster up ourselves without using EMR we report only the cost of running EC2 [4] virtual machines, and not the additional cost of Amazon EMR. We enable spilling
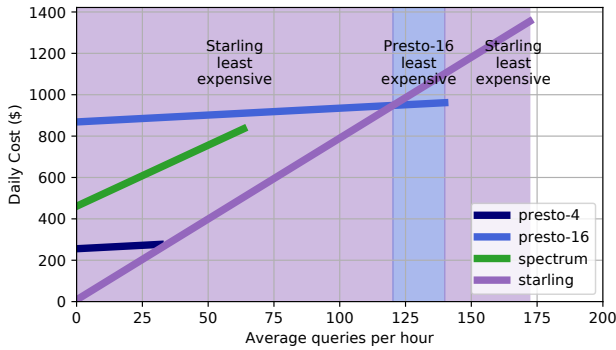
Figure 5: Daily cost with increasing queries of Starling and configurations with data stored in S3

to disk and query optimization. Before running queries, we collect statistics on all tables to assist the query optimizer.

We report performance and cost numbers on both a 5-node cluster (4 workers and 1 master) and a 17-node cluster, with 16 workers. We call these configurations `presto-4` and `presto-16` respectively.

Amazon Athena. Amazon Athena [1] is a managed query service based on Presto Version 0.172 [2]. Users define a schema for data sitting in S3 buckets, then execute queries against this data without provisioning hardware. Users execute queries on Athena with a REST interface, when the query completes, results are written to an S3 object. For our results below, we report the run-time of each query as measured by the Athena Service.

Users are charged for the number of bytes in S3 that their query scans. Like Starling, Athena has near zero cost when idle, with the exception of S3 costs for data storage. Unlike Starling, however, Athena provides users with no control over the degree of parallelism used in executing queries, and, as we will show, occasionally is unable to allocate sufficient resources to execute queries over large data files at all.

Starling. Unless otherwise noted, we configure Starling with all performance optimizations enabled including doublewrite (see Section 3.3.1) and pipelining (Section 4.4). Because Starling does not have a query optimizer, we set the number of tasks per stage manually as follows. First each input object is assigned a task. We then choose the number of tasks for join stages by sweeping over several options. We choose a configuration that is neither the fastest nor least expensive, but offers a good tradeoff of cost and performance. For the scale factor 1,000 dataset we disable multi-stage shuffling, but enable it for large joins in the scale factor 10,000 experiment. Starling's query plans use join orders generated from the Redshift optimizer with the `redshift-dc-dd` configuration, e.g. without partitioned data. Unless indicated, configurations are fixed over experiments.
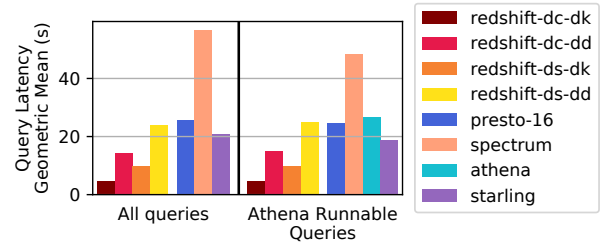
Figure 6: Geometric mean of latency on 1TB dataset

6.2 Cost of Operation

We conduct experiments to demonstrate that at moderate query rates, Starling is less expensive than alternatives.

We compare against systems that, like Starling, read data directly from cloud object storage. A comparison against systems that load data into local storage may be found in the extended paper [24]. Figure 5 compares two Presto clusters of differing size, Redshift Spectrum, reading from S3, and Starling. Each of these systems incurs more cost as more queries are executed. In the case of Starling this cost comes from reading base table and intermediate data from S3, as well as AWS Lambda costs. At about 120 queries per hour Starling's cost increases beyond `presto-16`. However, as `presto-16` is not as performant as Starling, with more than 153 queries per hour Starling is the only system able to keep up. No configuration reading base tables from S3 can execute more than 189 queries per hour back-to-back. While `presto-4` is cost competitive with Starling at about 33 queries per hour, this is its maximum query throughput when executing queries back-to-back.

In summary, Starling is the least expensive system of all configurations when query volumes are moderate.

6.3 Query Latency

While low cost is a benefit, users with ad-hoc query workloads also want interactive performance. Figure 6 compares the geometric mean of the latency of queries in the workload. Athena did not complete four of the queries, 2, 8, 9, and 15 reporting either a resource exhausted error, or because Athena was missing some SQL functionality Therefore, we also include a separate plot of the geometric mean of the subset of queries Athena completed. A full comparison of Starling to Athena follows in Section 6.5. These four queries are slower than average and thus removing them decreases the geometric mean for most systems. While Starling is just over four times slower than `redshift-dc-dk`, it does not have the advantage of pre-partitioned base tables nor sorted data. When compared to a configuration without this advantage `redshift-dc-dd` on the same cluster without pre-sorting or pre-partitioning, Starling's latency is less than 50% slower than Redshift after loading data into its local disks. Starling
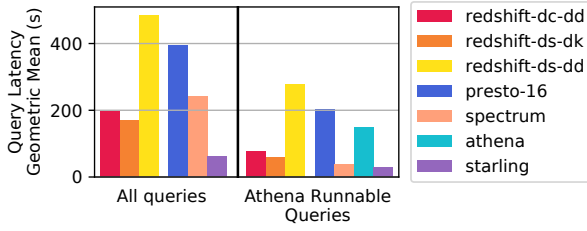
Figure 7: Geometric mean of latency on 10TB dataset

does not preclude sorting or pre-partitioning. For queries that do simple scans and aggregations like Q1 and Q6, Spectrum, which uses stateless workers like Starling for base table scans, achieves very low latency, even compared to Redshift configurations with local data. For queries with expensive joins like Q9 Starling has latency similar to Redshift configurations with default distribution (configurations with dd). This is likely because of extra cost of shuffling data when base tables are not partitioned on the join key.

For users with repeated workloads that require low query latency and are cost insensitive, a provisioned system with pre-loaded local data and tuned schema is still the best choice. But for ad-hoc analytics on data in cloud object storage, Starling has the lowest query latency. Against systems with tables pre-loaded, sorted, and stored locally, it has the lowest cost for query inter arrival times more than 60 seconds, and against systems data in S3, Starling is less expensive when query inter-arrival times are more than 30 seconds.

6.4 Scalability

In this section we show that Starling scales better than provisioned systems, without requiring an expensive reprovisioning step. We generated a scale factor 10,000 TPC-H dataset(10 TB before compression) and executed a subset of 12 queries from our query set, chosen for their range in the size of input data and the number of joins . To scale to the 10TB dataset, Starling increases the number of workers performing large joins, and uses multi-stage shuffles to mitigate large S3 read costs. We kept the configuration of other systems the same. We summarize the results in Figure 7. In this case, each provisioned system has query latency at least 2.7 times larger than Starling. Starling has the lowest latency of any compared system in 8 of 12 queries. The next fastest configuration, `redshift-ds-dk` has lower latency than Starling's for 1 of the 12 queries. `redshift-dc-dk`, the best performing config on the 1TB dataset, runs out of disk space when loading the 10TB dataset because of the additional indices it builds during the data loading stage. Thus, we could not compare against `redshift-dc-dk` for this experiment. Of course, adding more resources to the provisioned systems for this larger dataset could allow them to execute with lower latency. This experiment demonstrates the challenges

of provisioning a system for an ad-hoc query workload with arbitrary amounts of data. For workloads where the size of data inputs varies widely or is unknown ahead of time, the rapid elasticity of Starling gives it an advantage over provisioned systems. While provisioned systems would have to provision additional resources to handle more load, Starling scales on a query-by-query basis and thus is able to be more flexible to changes in input data size.

A discussion of the cost per query in the 10TB dataset can be found in the extended paper [24].

6.5 Pay-per-query Services

Fully managed query services, like Amazon Athena [1], come closest to realizing the goals of Starling. Users simply submit queries and charged on a query-by-query basis. Unfortunately, Athena is not a panacea for ad-hoc query workloads. Of the 19 queries on the 1TB dataset experiment Athena could not complete four. As seen in Figure 6, for the queries that were completed by Athena, query latency was more than 50% longer than with Starling.

Despite these challenges, Athena has cost per query competitive with Starling. Excluding the queries Athena was unable to execute, Athena has slightly higher cost per query than Starling at the highest query rates, $0.287 compared to Starling's $0.0256. Because Starling has a small cost for the coordinator and Athena does not, Athena is cheaper at query rates lower than one query every 350 seconds.

Athena does not scale to larger datasets. On the larger dataset, Athena completed only 5 out of 12 queries. On those completed, query latency was 5 times higher than Starling's. Finally, Athena costs more than twice as much per query. For users who need interactive performance, the inability to tune for performance in Athena makes it a nonstarter.

7 CONCLUSION

In this paper we presented Starling, a query execution engine for data analytics built on cloud function services. Starling fills a gap in user's desire for an analytics system that provides interactive query latency at low cost per query for ad-hoc workloads while being tunable to performance and cost objectives. Starling achieves these goals by harnessing the rapid scaling and fine granularity that cloud function services provide. Starling overcomes the challenges of managing hundreds of stateless workers that must exchange data through opaque cloud storage. Starling's optimizations make it cost and performance competitive even with provisioned systems, and allow it to easily scale from small to large datasets and back without making explicit provisioning decisions. An extended version of this paper may be found on Arxiv. [24]