Starling: A Scalable Query Engine on Cloud Functions

Motivation

- Modern Analytical workloads require certain key features from Databases
  - Does not require loading of data
  - Pay by query
  - Tunable performance
- Existing Cloud native databases do not provide all the features required by modern analytical workloads

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

Why Cloud Functions?

- Can read directly from Cloud Storage
- Low startup time and billed on per-invocation basis
- Many functions can be invoked in parallel (tunable parallelism/performance)

Challenges with Cloud Functions?

- Analytical queries can run for hours, but cloud function execution is limited to a few minutes
- Cloud functions execute in resource constrained environments
- Analytical queries require shuffling data, but cloud function do not allow communication between function invocations

Design : Starling Architecture

- Coordinator
- Cloud Function Service
- Workers
- Storage

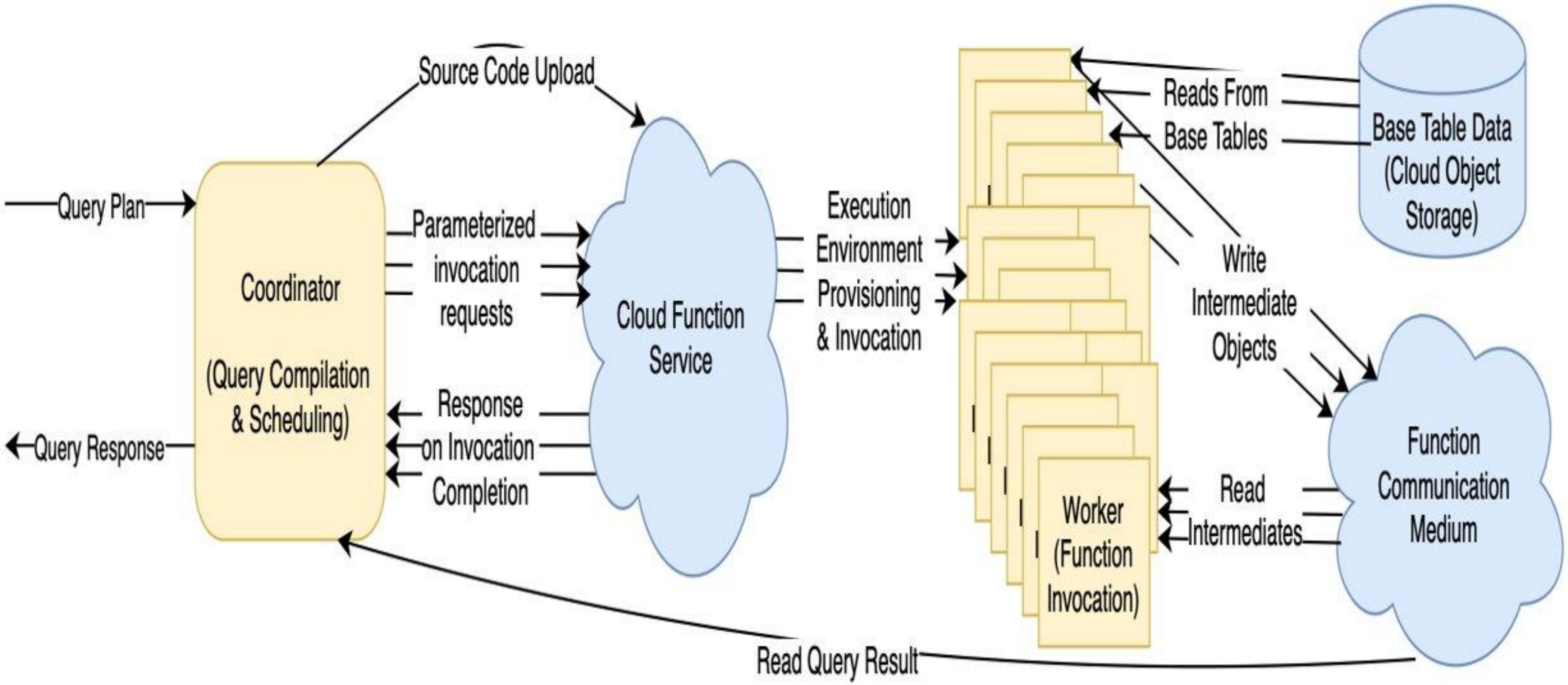

Figure 1: Query Execution in Starling. Opaque cloud components in blue, Starling components in yellow

Data Management in Starling

- Starling needs to work efficiently with raw data for competitive performance
- Base Tables and Intermediate State are both stored in Amazon S3
- Data shuffling requires all-to-all communication which has a high cost in S3
- One of the ways Starling mitigates this is by
    - enabling producers to write a single partitioned file
    - consumers read only the relevant partitions

Storage Latency Mitigation

- S3 has high aggregate throughput but much higher latency than other shuffling options
- Tasks perform several reads in parallel as opposed to performing blocking reads
- S3 does not guarantee read-after-write consistency
- Recently written objects to S3 by Producers may not be readily visible to Consumers
- Starling mitigates this risk by writing the same object to two different keys in S3
- Reduces the risk that a single visibility issue slows down all consumers

Query Execution : Relational Operator Implementation

- Operators implemented as a series of nested loops
  - enables pipelining of operations
- **Broadcast Joins :**
  - Input task for inner relation writes a single object to S3
  - Join tasks read inner relation and their subset of outer relation to perform join
- **Partitioned Hash Joins :**
  - Input task writes partitioned file (partitioned on join key) to S3 for both relations
  - Join tasks perform hash join on this partitioned data
  - These joins would require shuffling

Query Execution : Shuffling

- Standard shuffle requires all-to-all communication
- For small joins, starling performs 2sr reads
- For large joins, these many reads are unacceptable
- Starling uses multistage shuffle by introducing combiners
- This brings down the number of reads to 2(s/p + r/f)
- Cost for additional writes by combiners is negligible

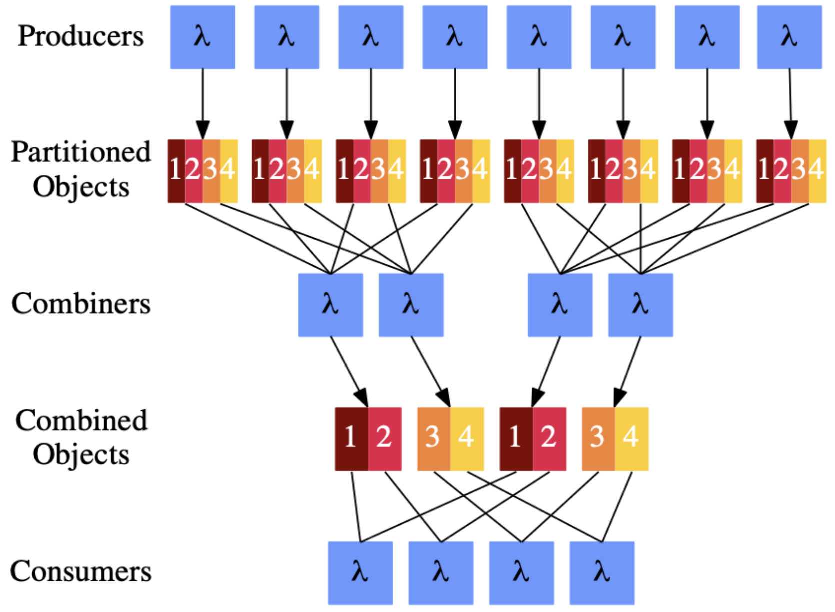

Figure 2: Starling multistage shuffle, function executions in blue, S3 Objects in shades of red showing partitions. Lines are reads and arrows are writes

Query Execution : Assigning Tasks and Pipelining

Assigning Tasks

- Trade-off between performance and cost
- Starling exposes this as user configured parameters

Pipelining

- Starling uses pipelining between stages to reduce query latency
- Consumer stages begin when a large fraction of producer inputs is available

Stragglers

- S3 requests often suffer from poor tail latency
- Tasks in intermediate stages can Straggle
- Causes dependent tasks to stall
- To counter this, starling implements read and write straggler mitigation techniques

Stragglers : Read Straggler Mitigation

- Observe how long a request takes compared to its expected completion time
- Expected query response time : r = l + (b/tc)
- If S3 fails to respond to a request within a fixed factor of the expected time, Starling sends a duplicate request
- It accepts whichever response returns first, and closes the other connection

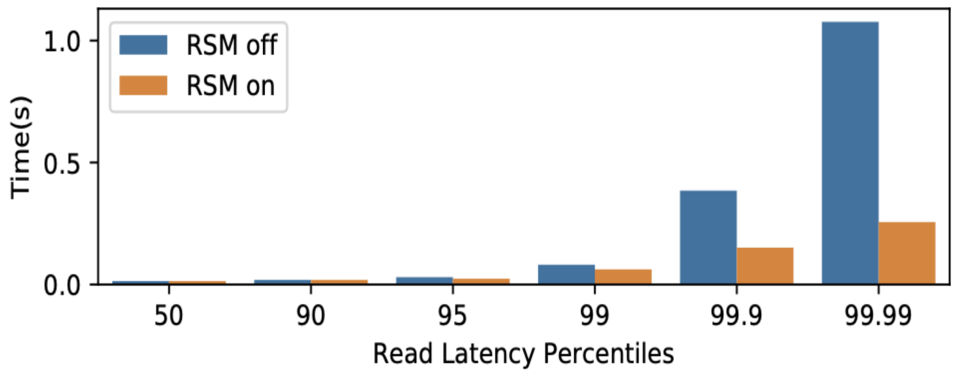

Figure 3: Read latency percentiles for 256KB reads to S3 from AWS Lambda. Comparing RSM off and on

Stragglers : Write Straggler Mitigation

- In most cases, requests sent to S3 quickly, but response from S3 may be delayed
- Using a strategy similar to RSM, Starling may react slowly to such cases
- Additional model to predict response times for writes once request has completed sending
- Second write request is started on a new connection if a straggler occurred as per these models

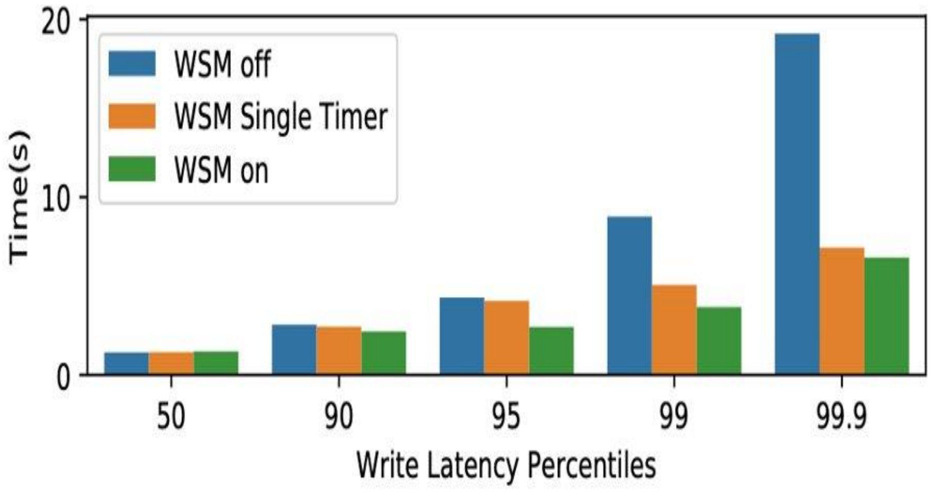

Figure 4: Write latency percentiles for 100MB writes to S3 from AWS Lambda. Comparing WSM off, with a single timeout, and fully on

Evaluation : Experimental Setup

- 1,000 (1TB) TPC-H [16] dataset for most experiments, and scale factor 10,000 (10TB) for the scaling experiment
- Systems Compared against
  - Amazon Redshift
    - dc – dense compute
    - ds – dense storage
    - dk – Distribution key and ordering enabled
    - dd – no distribution key and ordering
  - Redshift with Spectrum
  - Presto-4 with 4 workers
  - Presto-16 with 16 workers
  - Amazon Athena

Evaluation：Cost of Operation

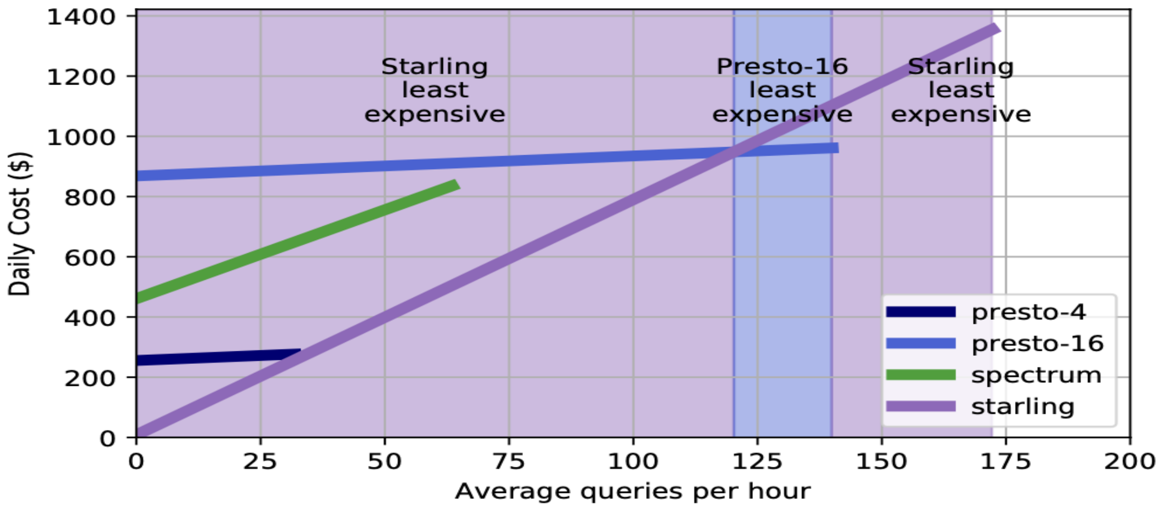

Figure 5: Daily cost with increasing queries of Starling and configurations with data stored in S3

- Starling is the least expensive system of all configurations when query volumes are moderate

Evaluation：Query Latency

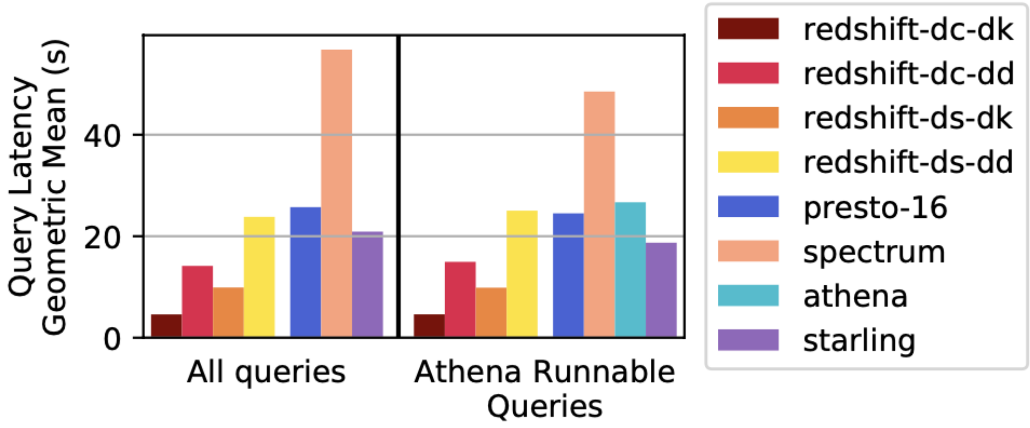

Figure 6: Geometric mean of latency on 1TB dataset

- For repeated workloads that are cost insensitive, a provisioned system with pre-loaded local data and tuned schema is still the best choice
- But for ad-hoc analytics, Starling has the lowest query latency

Evaluation : Scalability

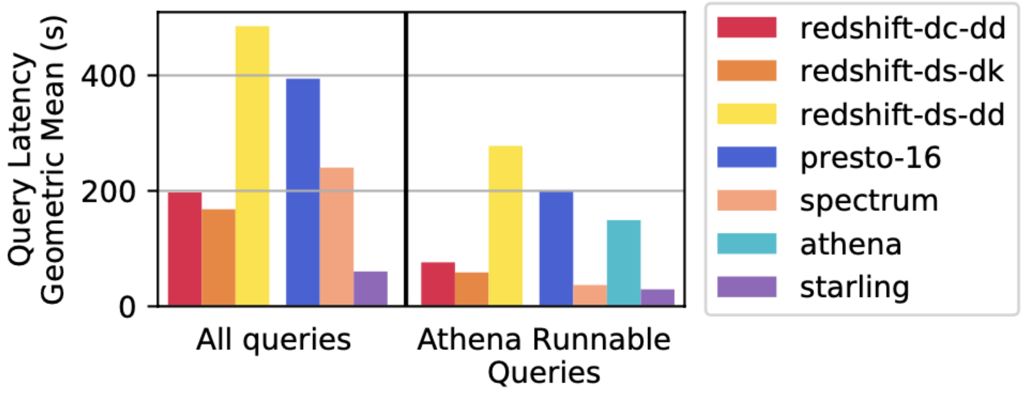

Figure 7: Geometric mean of latency on 10TB dataset

- Starling scales on a query-by-query basis and thus is able to be more flexible to changes in input data size as compared to other provisioned systems

Evaluation : Pay-per-query Services

- Athena provides a similar model and is cost per query competitive with Starling
- However, it is not suitable for ad-hoc query workloads
  - Many queries do not run
  - The ones which do run have higher latency
  - Doesn't scale well for larger datasets

WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON