2024년 09월 06일

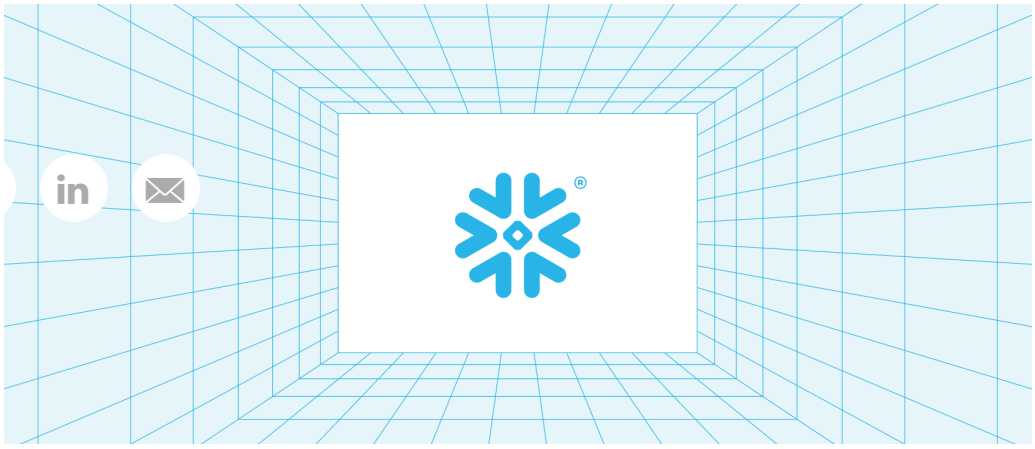
AUTHOR

Ata E Husain Bohra

Automated Query Retries: Improving Snowflake's fault tolerance

Core Platform

SHARE



Snowflake operates a complex system of services, these systems run at a massive scale, requiring running software on several thousands of compute instances and millions of processor cores. Failures at such a scale are inevitable.

In most cases, a query received by the Snowflake Service gets executed flawlessly, generating the desired results. However, in rare cases, query execution can fail due to a number of reasons: intermittent failures due to faulty compute instances, intermittent network failures between various component services, underlying cloud provider service disruptions (compute and/or storage), user errors (query syntax errors, for instance), overloaded instances, and/or software regressions (recent software updates).

In this blog, we intend to focus on a novel infrastructure engineered to improve our customers' experience by hiding complexity and attendant failures by retrying failed queries when safe to do so. That is, the operations being retried are idempotent. Given not all query failures are worth retrying (user error, for instance); the infrastructure is designed to handle specific failures caused due to following reasons:

Intermittent software and/or hardware failures — *out-of-memory* errors, instance failures, network timeouts etc.

Potential software regressions due to recent updates to Snowflake software.

AUTHOR

Ata E Husain Bohra

Query Retry Infrastructure Deep Dive

The infrastructure consists of four components. These are our Metadata management, Query Retry Selector, Instance Selector, and the Request Dispatcher. Below we describe what each component does.

Metadata Management

In order to retry an operation, one of the essential requirements is to preserve the input request state which is used to re-execute the operation. The input state can be preserved either in volatile memory ('Random Access Memory) of the compute instance that will execute the failed request and/or could be persisted in a durable datastore. The durable datastore could be local and/or remote to the compute instance executing the failed request. When designing the Query Retry infrastructure we choose to preserve the input request state in a strongly consistent and highly durable network connected datastore; this allows us the flexibility to retry operation on a different compute instance(s) instead of retrying on the same instance where it has failed. For every customer query, the framework persists metadata information essential to (re)execute the request, it also stores a counter tracking number of times a given query has been retried so far, referred to as '*retry-attempt-counter*', hereafter.

Query Retry Selector

An automated query retry is a powerful tool. It abstracts intermittent hardware and/or software failure issues from the customers. However, it needs to be implemented with great care as retrying a non-idempotent query could compromise data integrity. The *Query Retry Selector* module is responsible for determining if an operation being retried is idempotent.

Plugin-based Architecture: Two main design considerations for the selector module are:

1. Based on the complexity of the query, multiple checks need to be performed before a failed query can be declared eligible for retries.

2. Provide extensibility, flexibility, and isolation of core checks and custom logic. An inability to provide such guarantees would slow down the development of new features due to tight dependencies on teams owning infrastructure to make required changes.

To satisfy the above design considerations, we choose to implement the selector module modeling a '*plugin-based architecture*'. The module is a collection of two types of filters: *core system filters* and *custom plugin filters*. The infrastructure team designed, implemented, and maintains *core filters*, whereas, specific rules can be implemented as *plugin-filters*. A *plugin filter* can be added/removed/updated with little or no effect on the rest of the *core filters* or other *plugin-filters.* This architecture allows better collaboration among various Snowflake teams. New filters can be added/removed by the team designing and implementing new features with limited or no involvement from the team owning Query Retry infrastructure.

AUTHOR

Ata E Husain Bohra

At every new engineering feature needs to be accompanied by a design document detailing the proposed architecture, important implementation details, dependencies, testing strategies, etc. One of the core components of the design document is to ensure that the proposed feature/change implements a new Query Retry Selector *filter*, updates an existing *filter(s)*, or that existing *filters* are sufficient.

Dynamic control: The Query Retry Infrastructure was designed and built to allow dynamic fine-grained control to (dis)allow query retries in the production deployments without software updates. One of the *core filters* implemented allows controlling query retry behavior based on operation command type (refer to **SQL Command Reference** for details) and there are several *custom filter* implementations allowing dynamic control of query retry behavior for new Snowflake features. The dynamic control is accomplished by altering corresponding *configuration parameters.*

Instance Selector

The *Instance Selector* module is responsible for selecting compute instance to retry a query. The module design ensures the same compute instance never gets repeated during a query (re)execution lifecycle. Below we describe the rules governing compute instance selection per query-retry attempt.

First Retry Attempt

Observing query failure patterns on production deployments, a substantial number of queries fail when run on a faulty compute instance.

Retrying the query on another healthy instance running the same software version would most likely yield success. Hence, the first retry attempt is done on another compute instance running **the same Snowflake software version** as the first compute instance and the '*retry-attempt-counter*' gets incremented in this process.

Second Retry Attempt

AUTHOR

Ata E Husain Bohra

Snowflake seamlessly auto-updates software versions without **any** service downtime. Snowflake runs hundreds of hours of functional correctness tests supplemented with performance and load tests before releasing a new software version. Despite these precautions, at times changes to Snowflake software may cause customer jobs to fail. If a query fails after the first *retry attempt*, the second retry attempt is done on a compute instance running a **previously released Snowflake software version**.

If a query fails on a second retry attempt as well, then the query is deemed as a failure.

Request Dispatcher

The *Request Dispatcher* module is responsible to dispatch retry requests to an instance selected by the Instance *Selector* module. The module implementation ensures safe and reliable message delivery guarantees.

Figure-1 below shows interactions of all above-described infrastructure components when attempting to retry a customer query within a given compute instance. Further, it depicts a query execution lifecycle if it gets (re)executed across multiple compute instances.

Figure 1: Snowflake Query Execution Lifecycle

Engineering Challenges

Exactly Once Semantics

The Snowflake Data Cloud is a collection of services interacting with each other using a RESTful interface. On a large scale, distributed systems failures are inevitable: services may encounter compute

instances and/or transient network failures. As described in the *Request Dispatcher* section, the compute instance where the query failed, the request dispatcher will retry the request to another compute instance to re-execute the operations. Note that various failure scenarios such as timeouts due to intermittent network failures resulting in the request potentially not being received by the destination instance or compute instance failures before or after receiving/sending the request. The infrastructure design guarantees that a given customer query getting retried is executed successfully **exactly once** under all circumstances.

AUTHOR

Ata E Husain Bohra

Return result to the client

Snowflake exposes endpoints for customer applications to connect and submit queries. In most cases the compute instance accepting the request is able to successfully generate the desired result. However, when query retry is used, the compute instance executing (or executed) the request *could* be different from the instance where the customer submitted the request. The framework designed a routing layer to redirect customer requests to fetch results from the appropriate compute instance or has executed the request, ensuring the integrity of the results being returned.

Debugging failed queries

One of the first principles for designing a good production system is to incorporate enough debug information to analyze failures in production quickly and easily. It can be argued that auto-retying of queries adds extra complexity when debugging failures. The infrastructure is engineered to ease triaging failures by *chaining* query-execution attempts across multiple compute instances. Engineers are able to construct a detailed query-execution lifecycle timeline with all the events of interest.

Test infrastructure support

To maintain **high product quality**, one of the *production readiness* criteria for a new feature is to implement efficient test techniques. This ranges from thorough unit, integration, and load testing. The Query Retry infrastructure provides a novel fault-injection mechanism enabling a test to force-fail any query and follows query-retry execution flow. For load testing, the framework allows specifying % or total queries that need to be retried, validating the system's ability to handle production workload traffic.

Retrying Queries at Production Scale

Snowflake service platform serves on average **more than a billion customer queries per day**. In order to determine the effectiveness of the infrastructure, we analyzed production data over a period of ***90 days*** using the below equation:

E = (S *100) / T

where:

T: total number of queries that failed and was retried by the data cloud platform.

S: total number of queries that failed, retried and ran successfully on the first/second retry attempt.

E: % of retried queries that ran successfully.

In absence of any software regressions due to software updates, in a ste st the framework executed ~**55%** of failed queries successfully. The lion's share of which got successfully executed on the *first retry-attempt* protecting customers against *faulty* compute instances and/or intermittent network failures. Despite extensive testing before releasing any Snowflake software version, in some rare instances, software regressions did fail customer queries. The framework was able to execute **~97%** failed queries successfully on the *second retry-attempt*, hence, significantly improving the customer user experiences.

AUTHOR

Ata E Husain Bohra

Figure-2: Query Retry Success % (90 days production data)

Figure-3: Query Retry Attempt Effectiveness % (90 days production data)

Future Outlook

The QueryRetry Infrastructure is designed, implemented and is serving customers well in the production. We have a strong roadmap ahead of us and we intend to leverage the existing infrastructure as the foundation to solve interesting engineering challenges aimed towards continuously improving the customer user experiences such as dynamically controlling the number of retry-attempts and/or instance selection based on failure types, etc.

If you are passionate about building complex large-scale distributed systems for Snowflake Data Cloud, please check out our Careers page for open positions.

AUTHOR

Ata E Husain Bohra

SHARE

SHARE