2024년 09월 06일

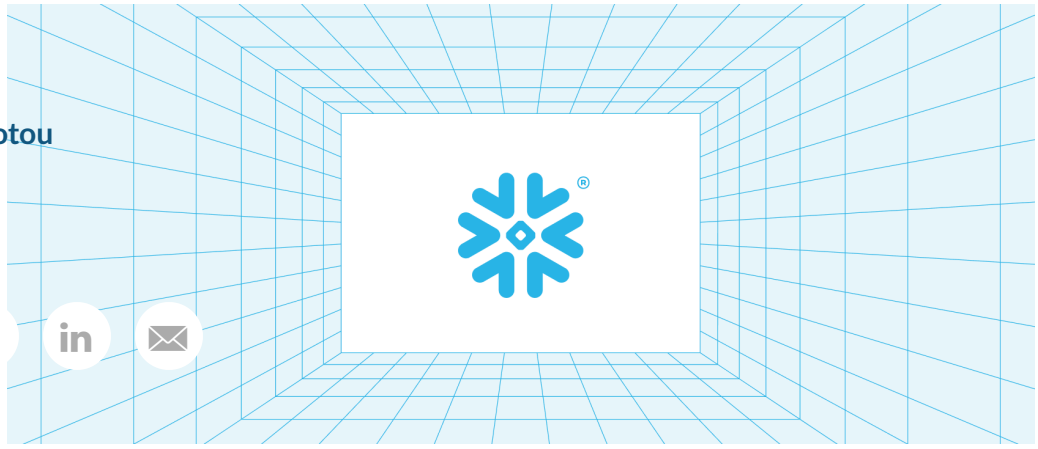
AUTHOR

Ioannis Papapanagiotou

SHARE

Snowflake's Elastic Cloud Services

Core Platform



Snowflake's Data Cloud is powered by an advanced data platform provided as Software-as-a-Service (SaaS). Snowflake combines a completely new SQL query engine with an innovative architecture natively designed for the cloud to enable data storage, processing, and analytic solutions that are fast and easy to use. Snowflake's cloud-first architecture runs completely on public cloud providers without the burden of maintaining on-premise servers and can run on any public cloud, creating a cloud-agnostic platform. The following picture showcases the high-level architecture of Snowflake. You can learn more about our architecture in the Key Concepts & Architecture documentation or in the Snowflake Elastic Data Warehouse research paper.

AUTHOR

Ioannis Papapanagiotou

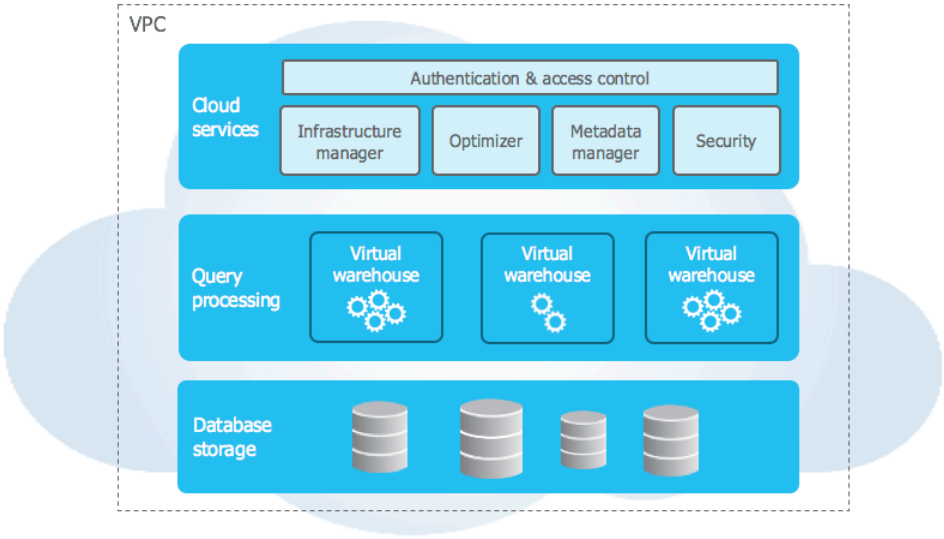


Figure 1. Snowflake Technical Architecture

In this blog, we are going to focus on the management of the Cloud Services layer, also called *Global Services (GS)*. At a high level, the Cloud Services layer is a collection of services that coordinates activities across Snowflake. These services tie together all of the different components of Snowflake to process user requests, from login to query dispatch. The Cloud Services layer also runs on compute instances provisioned by Snowflake from the cloud provider. Services managed in this layer include authentication, access control, infrastructure management, metadata management, query parsing, compilation, and optimization, etc.

The virtual machines in the service layer are all provisioned from a public cloud provider such as Amazon Web Services, Microsoft Azure, or Google Cloud. The service infrastructure is divided into two categories: *foreground (FG)* and *background (BG)* instances. The FGs are responsible for handling customer queries (including compilation and query lifecycle management). The BGs are responsible for running a variety of background services including instance health management, updating load balancer configurations, and the orchestration tasks themselves.

FGs are partitioned into clusters, each of which serves a dedicated set of customers. Customer queries are routed to the correct cluster based on load balancer configurations that are dynamically updated. Similarly, the BG instances are partitioned into separate sets of instances (dedicated clusters for provisioning services, core background services, compute tasks, etc.).

Elastic Cloud Services

The compute layer of Snowflake, exposed to customers as virtual warehouses, handles query execution and is explicitly scaled and

configured by the customer. Warehouses come in T-shirt sizes (S, M, L, etc.) corresponding to the amount of compute power in the warehouse. Dynamically sized multi-cluster warehouses are also available where Snowflake can optimize the number of clusters of the warehouse. On the contrary, the Cloud Services layer is transparent to the end-user and manages metadata, transaction routing, as well as the creation and management of the compute layer. Our goal is to provide the Cloud Services layer the ability to scale elastically to handle any workload.

AUTHOR

Ioannis Papapanagiotou

Many customers and accounts can be routed to a multi-tenant cluster of nodes in the GS layer, and there will often be great disparities in incoming query volume throughout the day. As Snowflake deployments scale, the number of active GS instances has increased: there are more customer queries to run, more files to be cleaned up, more metadata to be purged, etc. A Snowflake deployment is a distinct Virtual Private Cloud (VPC) that contains the major parts of Snowflake software including a scalable GS tier. With the increased number of GS instances, we started to run into issues around how to manage our fleet at scale.

Our team is responsible for the GS instance and cluster lifecycle, health management and self-healing automation, topology and account/service placement, traffic control and resource management (autoscaling, throttling, and placement), as well as the aforementioned BG framework. Our North Star is to have a stable service, resilient to failure that transparently manages dynamic resource utilization in a cost-efficient manner.

Below we cover some of the work the team has accomplished in automating the GS infrastructure.

GS Cluster Manager

We perform the majority of GS administration activities through a single service that we call the *GS Cluster Manager*. ECS is responsible for

Code upgrade/downgrade of Snowflake deployments;

Creation, update, and cleanup of GS clusters;

Enforcement of mapping manifests and constraints of accounts to clusters and accounts to version;

Interaction with our Cloud Provisioning Service to generate and terminate cloud instances across all Cloud providers, manage every

instance in their lifecycle, and perform corrective actions in case of unhealthy instances.

The following figure showcases how we create an ECS-managed cluster. The first step is to create a GS cluster and register a package with the latest Snowflake release. Then we declaratively allocate a customer to the cluster and allow the Cluster Manager to converge the actual system to the declared topology. Once the customer is ready to execute a query workload, we scale out the number of instances in the cluster to accommodate the needs for the corresponding workload and update the topology on our reverse proxy (currently we leverage Nginx).

AUTHOR

Ioannis Papapanagiotou

Figure 2. Creating and Registering a GS cluster

Most of these admin actions happen through privileged SQL queries such as

```
SELECT
SYSTEM$REGISTER_GS_PACKAGE('release_5.1.0', ...);
```

We have declared to the GS Cluster Manager that queries for the account SNOWFLAKE must run on cluster SNOWFLAKE_QUERY and must run on version 5.1.0. The GS Cluster Manager will then ensure the deployment state converges to meet the following:

The GS Cluster Manager can look up metadata on the designated GsCluster and sees, for example, that SNOWFLAKE_QUERY may have the following attributes: *base_size=3, server_type=SF_HIGH_MEMORY,* etc (where SF_HIGH_MEMORY is an example name of a Cloud instance type)

The GS Cluster Manager knows that the account SNOWFLAKE is set to run on that cluster and it needs to run the 5.1.0 release. So it can provision 3 new SF_HIGH_MEMORY (this server type maps to a Cloud instance) instances running the 5.1.0 binary

When the 3 instances exist and are ready, the GS Cluster Manager can use them to form a new cluster and route all queries issued by SNOWFLAKE account to that cluster.

The most important point is that GS Cluster Manager is **declarative**. The only task human operators have is to declare what is the correct state of the world. Operators do not prescribe iterative steps to accomplish the goal, and therefore do not have to be exposed to the complexities of actually making things happen. For example, our operators don't have to know how to obtain more instances, don't have to know to clean up excess instances, and don't have to know when one instance is down, etc. All these responsibilities are fully automated by the GS Cluster Manager.

AUTHOR

Ioannis Papapanagiotou

Automatic and safe code upgrade mechanism

At Snowflake, we roll out a new code version on a weekly cadence. We have fully automated the process to minimize the opportunity for human error. Code upgrade is orchestrated in a safe way, e.g. if a customer is running a query on a GS instance, then that GS instance cannot be shut down until all its queries finish execution, as the customer would otherwise see a query interruption. We manage online upgrades with a rollover process. First, new GS instances are provisioned with the new software version installed, and some initialization work such as cache warming is done. You can find more information about Snowflake's caching. We then update the Nginx topology to route new queries to the new instances. Since customer queries can run for hours, we keep the older instances in the cluster running the previous version until the workload terminates. Once the old version instances in the cluster have completed their work, the instances are removed from the cluster and the upgrade is complete.

Figure 3. GS cluster version upgrade

Fast Automatic & Safe Rollback

Rolling out a new code version is inherently risky. We have high-quality standards in our testing ecosystem. However, we do not know if we have inadvertently shipped a new bug or regression until the issue is seen in

production. We have two ways to roll back. The fast rollback is in case of a catastrophic issue related to the release. During releases, we have a grace period in which both the old and new software versions run, with the old set kept idle and out of topology. This allows us to perform instantaneous rollback by updating the Nginx topology to map requests to the old, stable software version. Figure 5 showcases the fast rollback path.

AUTHOR

Ioannis Papapanagiotou

Figure 4. GS cluster version targeted account rollback

We can also perform a targeted rollback. Not all bugs hit customers equally, and many customers have unique workloads. For example, a bug may be released that affects only a handful of customers due to their particular workload, but the other thousands of accounts may be able to go forward with the new code release. It is clearly unnecessary to do full rollback for all customers because of a single customer-facing issue, but we also do not want to provide any customer a subpar experience; thus we support rollback on a per-account basis by explicitly mapping only affected accounts to the old software version.

GS Instance Lifecycle and Cluster Pools

At Cloud scale, instances often become unhealthy and need to be replaced to maintain high service availability. More broadly, each instance has a lifecycle beginning with its provisioning and ending with its release back to the cloud provider. Sick instances are a natural phenomenon and our current way of handling them is to automatically quiesce, quarantine, and then replace them to maintain a high level of availability. However, it's difficult to know exactly what caused an instance to enter a bad health status and we often need to look at logs to diagnose issues. Therefore, we have a holding mechanism to retain unhealthy instances for diagnostics purposes. To recycle instances after software upgrades, replace unhealthy instances with healthy ones, and retain instances encountering unusual health issues, we maintain a few "*instance pools*". The movement between the pools is automated as part of the GS lifecycle and is based on a state machine that we will describe below. The pools are as follows:

Quarantine Pool: instances that have entered some sort of negative state and need to be removed from their clusters and restarted;

Free Pool: healthy instances that can be quickly swapped into clusters if any active instances enter a negative state;

Graveyard Pool: instances that have lived through the whole lifecycle and are actively being released back to the cloud provider.

Holding Pool: instances retained but no longer actively managed by ECS so that they can be analyzed without disruption.

AUTHOR

Ioannis Papapanagiotou

Figure 5. State Machine of GS Lifecycle

In our state machine, all instances are provisioned by the Cloud Provisioning Service (CPS). CPS abstracts fetching instances across multiple public Cloud providers (AWS, GCP, Azure). The last state of our state machine is the *Graveyard*. After that, we release the instance(s) back to the Cloud provider. You will observe that we also have one state, called *Quarantine*. Quarantine is where GS instances are sent to self-resolve any pending tasks assigned to them. For the holding pool, we allow any non-Graveyard instance to be sent to Holding to provide flexibility to the operator when needing to debug something that was not in a working cluster. In the case of a software upgrade, we will move the older version instance to the Quarantine pool. Instances move from Free Pool to Quarantine when they've been marked terminal. This usually happens when they fail to provision or fail to start within a parameterized time limit, or when ECS decides that the free pool is over-provisioned.

Figure 6. GS cluster instance recycling

At scale, we have observed that not all instances will be healthy and functional. Some instances can enter a bad state (JVM Garbage Collection death spiral, full disk, broken disk, corrupt file system, too many file descriptors, etc.) and at which point they become dysfunctional and need to be rotated out. However, these unhealthy instances may still be running customer queries, some of which may take hours to complete, so even rotating them becomes a waiting game similar to the code upgrade scenario. Multiply this by the ever-expanding number of Snowflake deployments, some of which are capable of running thousands of GS instances. Furthermore, not all unhealthy States indicate the instance has an inherent issue. Many issues may be intermittent, in this case, it becomes easier to restart the GS server process than to procure a new node from a cloud provider.

AUTHOR

Ioannis Papapanagiotou

Figure 7. Sick GS instance isolation

Partitioning Customers into GS Clusters

Without any kind of cluster partitioning, we would encounter a number of performance and reliability issues. For example, if there were no cluster-level locality and any customer-issued query could be served by any GS instance, we would not be able to make good use of caching. As another example, without partitioning, the blast radius of instance failure increases, as the number of accounts that may encounter delays due to having their queries routed to a failed instance would be unbounded. Similarly, the blast radius of a multiply-retried pathological query potentially extends to the entire Snowflake deployment.

As described earlier, we map accounts to clusters to combat such issues. We support single-tenant clusters (only one account mapped to the cluster) and multi-tenant clusters (several accounts sharing a cluster). Multitenancy improves resource utilization and efficiency for small accounts with bursty workloads, as such an account using an entire cluster on its own would be idling most of the time. Larger accounts with more consistent, sustained workloads may be more suited to a single-tenant cluster configuration where underutilization is less of a concern and we can derive even greater benefits from cache locality. ECS actualizes the cluster partitioning, automating cluster mapping management at scale. Without ECS manifesting these cluster mappings,

the work would have to be done manually by human operators which is unacceptable for several reasons:

human errors: all instances which comprise a cluster should be homogeneous in code version, server type, encryption level, etc. When everything is manual, human errors inevitably crop up and we started seeing heterogeneous clusters causing all sorts of issues

configuration errors: related to the above, for example, customers which opted for a higher encryption setting should never be allowed to run queries on a normal cluster.

efficiency: if we ever want to, for example, isolate a single pathogenic customer into a dedicated cluster, there are many manual steps that must be done (find spare cluster and hope no other accounts are mapped to it, if there are no spare clusters then shuffle accounts around, or at worst case find extra nodes, provision nodes if there is no spare capacity, form the cluster taking care it is a homogeneous cluster, remember which code version the isolated customer is supposed to run, and so on)

scale: we are simply serving more accounts and running more clusters than we can staff with human operators.

Conclusion

In this blog post, we introduced our Elastic Cloud Services layer which serves as the brain of delivering Snowflake as a service. We covered how we manage large and small clusters, automatic upgrades, fast rollback strategies, customer account partitioning, and our state diagram of the instance lifecycle. In a follow-up post, we will discuss how we improve elasticity with auto-scaling and dynamic throttling of our Service layer.

As the workloads Snowflake serves are growing extremely quickly, we have a lot of work to do to make our infrastructure more capable and efficient. We manage large fleets of cloud instances and face challenging and exciting problems in building robust and elastic distributed systems. If you found the above exciting, please get in touch with us! We would love to hear from you!

AUTHOR

Ioannis Papapanagiotou

Snowflake Inc.

AUTHOR

Ioannis Papapanagiotou

SHARE

SUBSCRIBE NOW