HIVE-19821: Distributed HiveServer2

Problem

HiveServer is a centralized service that provides query compilation, planning, and execution for all connected Hive sessions. HS2 deployments often hit OOM issues due to a number of factors:

- Queries that scan a large number of partitions have to pull a lot of metadata into memory
  - For example, a query reading thousands of partitions requires loading thousands of partitions into memory
- Very large queries can take up a lot of heap space, especially during query parsing

Hive often has to deal with a “big metadata” problem, where the amount of metadata required to run a query is often much larger than expected. This metadata can come in the form of partition metadata, column stats, permissions, etc.

Today, Hive users work around these issues by (1) increasing the amount of memory allocated to the HS2 process, or (2) adding multiple HS2 instances and configuring a load balancer to split load between the instances.

Proposed Solution: Distributed HiveServer2

Distributed HS2 proposes to do all query parsing, compilation, planning, and execution coordination inside a dedicated container. This should significantly decrease memory pressure on HS2 and allow HS2 to scale to a larger number of concurrent users.

Each Hive session will have a dedicated container that will run all query parsing and execution for that session.

This does not solve all the memory issues mentioned in the problem statement; the biggest benefit is **isolation**. A user cannot submit a query that brings down an entire HS2 instance. A crashed HS2 instance affects all users and forces them to re-submit their queries. Very often, this means re-starting queries that were already running. With Distributed HiveServer2 poorly written queries, or queries that require lots of metadata can only cause their own session to crash, rather than all sessions running on the HiveServer2 instance.

Benefits

Isolation: Different user sessions are isolated from each other; one rogue query that fetches a million partitions can't bring down an entire HS2 instance and fail all other running queries

Scalability: A single HS2 instance should be able to handle many more user sessions / connections than before; users don't have to continuously increase the heap size of HS2 either

Alternatives

Load Balancers

The typical way to scale HS2 is to spawn multiple HS2 instances and configure a load balancer to distribute load across the multiple HS2 instances. This provides scalability, but does not completely solve the issues mentioned above.

There is no way to accurately predict how much memory is required to parse, plan, and execute a Hive query. This makes it difficult to know when HS2 instances will go down. A query may fetch zero or one-thousand partitions, it’s difficult to know without first parsing the query.

This makes round-robin scheduling problematic because a single Hive query could easily push a HS2 instance overs its limit.

Furthermore, when a HS2 instance does crash, if users re-submit their queries the load will be split amongst all other HS2 instances, but there is no guarantee that this increased load will lead

to other HS2 instances (e.g. cascading failures). In fact, a rogue query that say does a select star and loads thousands of partitions, could potentially crash all HS2 instances if it is retried indefinitely.

HiveServer2 Cluster

A single HS2 “master” could manage a cluster of “slave” HS2 instances. The master instance accepts all incoming connections and spawns a process on a slave instance for each Hive session. The slave process then handles all query compilation, etc. for its associated session.

This has the benefit that it keeps the isolation between the execution engine Application Master (e.g. Spark Driver or Tez AM) and Hive query parsing and planning. See section on “Drawbacks and Risks” for a more detailed analysis on this.

The work done here could be thought of as a step towards this design.

Admission Control

Building an admission control system for this issue is difficult because its very difficult to know how much memory a Hive query will require without first compiling it. This leads to a catch-22, there is no way to know if admitting a query will crash HS2 without first compiling it, but compiling it may cause the crash due to excessive metadata requirements.

Design

The main idea is to create a new `IDriver` called `RemoteProcessDriver` that wraps an `IDriver` (such as `Driver`) and runs all of its methods in a remote process.

Implementation

Hive

Phase 1

This feature is configurable at a per-session level via the config:
`hive.server2.enable.container.service`

This requires the following code changes:

- `RemoteProcessClient`: A client API for interacting with an `IDriver` that is running inside a remote process, since the API focuses on executing an `IDriver` remotely, the API is very similar to the `IDriver` interface

- `RemoteProcessDriver`: An implementation of the `IDriver` interface that uses the `RemoteProcessClient` and `RemoteProcessLauncher` to run all of its methods in a remote process
- `RemoteProcessLauncher`: Launches a remote process that will run the remote `IDriver`

Hive on Spark

For HoS this just requires moving all query compilation, planning, etc. inside the application master for the corresponding Hive session.

Phase 1

The first version of the Hive on Spark integration will focus on getting simple queries to work. This requires the following code changes:

- `SparkRemoteProcessClient`: Implementation of `RemoteProcessClient` that uses a `RemoteProcessHiveSparkClient` to interact with the remote process
- `SparkRemoteProcessLauncher`: Implementation of `RemoteProcessLauncher` that uses `SparkSession` and `SparkSessionManager` to launch the remote process aka the remote driver
- `RemoteProcessHiveSparkClient`: Similar to `HiveSparkClient`, defines an API for running `IDriver` methods inside the `RemoteDriver`
- `RemoteProcessHiveSparkClientImpl`: Implementation of `RemoteProcessHiveSparkClient` that uses the `AbstractSparkClient.ClientProtocol` to run the IDriver methods
- `SparkWorkSubmitter`: An interface that controls how a `SparkWork` object should be submitted, necessary as `SparkWork` submission (done inside `SparkTask`) is different when the remote process service is enabled vs. disabled
- `SparkMemoryAndCoresFetcher`: An interface that controls how `SetSparkReducerParallelism` fetches the memory and cores of the Spark session; necessary as this logic is different when the remote process service is enabled vs. disabled
- `RemoteProcessDriverExecutor`: Runs an `IDriver` inside a remote process (e.g. the `RemoteDriver`); necessary because the `spark-client` module has no dependency on the `ql` module

The `AbstractSparkClient.ClientProtocol` and `RemoteDriver.DriverProtocol` classes were modified to support the new communications mechanisms required for these classes. Essentially, they need to support sending requests for running `IDriver` methods in the remote process.

Future Work

- Lazily load the SparkContext in RemoteDriver: Right now RemoteDriver creates the SparkContext immediately, but it instead it should be loaded lazily

Usage

In its current form, this feature will be most useful for users who are running long ETL scripts. Scripts that run a pipeline of Hive queries, create and load multiple tables, run multiple select queries, etc. It will be less useful for users who simply login to Hive just to perform a few metadata operations (e.g. altering some partitions). For these types of “lighter” workloads, there is no need to spawn a separate container to run them.

Ideally, this will be most beneficial for “production” clusters where users are typically running complex workloads rather than ad-hoc jobs.

Drawbacks and Risks

There are a number of potential risks to this design:

- Coupling of query parsing and planning with execution engine AMs
    - For HoS, the Spark driver often requires a large amount of memory, especially at scale; however, its not always the case that query parsing will require an equivalent amount of memory
    - Holding onto all the Spark driver memory just for query parsing and planning may cause a waste of resources
        - Today, this may not really be a big issue given that once a Spark driver is spawned, its resources do not get released until the Hive session is closed
    - The “HiveServer2 Cluster” mentioned in the “Alternatives” section provides a way to solve this issue

Future Work

Furthermore, the following enhancements will be added post phase 1:

- Security integration
    - Hive will need a way to authenticate with the metastore from a arbitrary YARN container, along with other services such as Sentry or Ranger
- Operation Logging integration
- HS2 Web UI support
- CliDriver integration

More complex enhancements can be considered as well:

- Lazily spawn the remote process:
    - The current design requires all queries in a Hive session to be run inside the remote process
    - If the remote process was only spawned when a select query is executed, this may save some resources; there may be no need to even run simple DDL operations in the remote process at all
    - This might require coordination of state across HS2 and the remote process