Short Description:

This article aims to provide some best practices to follow when deploying the Apache Phoenix QueryServer (PQS) with an Apache HBase cluster

Article

Introduction

Apache Phoenix is a relational database "layer" built on top of Apache HBase, a NoSQL key-value data store. Until recently, the only provided entry-point for users to Phoenix was by their JDBC driver. This implicitly limited client applications to JVM-based languages.

The Phoenix QueryServer was built using a sub-project of the Apache Calcite project named Avatica. Avatica is a JDBC-over-HTTP library. It includes both an HTTP server and a JDBC driver. The HTTP server has its own wire API which can be leveraged by clients in any language communicate with the backend JDBC server.

Deployment

One of the goals of the Phoenix QueryServer was to follow typical deployment strategies of HTTP servers. The primary adopted goal is that the QueryServer aims to be stateless. While this is not entirely the case as QueryServer does maintain internal state, the server and Java client are written in such a way that any missing state in the PQS instance can be automatically recovered by the client.

Because PQS is designed to be stateless, it also scales horizontally. The general deployment recommendation is that PQS is installed on every node running an Apache HBase RegionServer. This enables users to control Phoenix throughput by increasing or decreasing the number of nodes which PQS is deployed on, just as one would do with HBase. The resource consumption of the PQS is minor when idle, so deploying it everywhere will have little impact.

There are, however, considerations which might impact placement of PQS on every RegionServer node. For example, not all RegionServer nodes in a cluster may be accessible to external clients, partially or fully, for security reasons. It's also possible that clients are running outside of the cluster's perimeter. In this case, specific nodes might be designated as externally facing, again for security reasons, and the PQS instances would have to be limited to those nodes.

High Availability

Like deployments of HTTP servers, the use of an HTTP load balancer can also be used to achieve high availability. There are two main approaches to take with load balancers, each with their own pros and cons.

A generic load-balancer

As mentioned earlier, the provided Java client to Avatica is capable of recovering from any missing server-side state. This is the basis which allows PQS to be used behind any generic HTTP load balancer, such as HAProxy, Nginx, or the Apache HTTP Server (httpd).

To efficiently implement this approach, it is strongly recommended to enable the appropriate configuration for the load balancer which will always route a client to a given server as long as the distribution of servers does not change, commonly known as "sticky sessions". This configuration will ensure that clients do not spend an excessive amount of time recreating server-side state.

HAProxy example

PQS has been tested using HAProxy 1.5.14. The following is an example configuration file for HAProxy.

```
global
  maxconn 256
defaults
  mode http
  timeout connect 5000ms
  timeout client 60000ms
  timeout server 60000ms
frontend http-in
  bind  *:8888
  default_backend servers
backend servers
  balance source
  server queryserver1 server1:8765 check
  server queryserver2 server2:8765 check
```

Load-balancers with client-driven routing

Another potential strategy for using PQS behind a load balancer is by implementing client-driven routing. This is presently only a theoretical approach but still a sound one. This approach implies the creation of a custom client to PQS by implementing Avatica's protocol which passes an identifier to the load balancer to control how the request is routed to the backend QueryServers.

Avatica's wire API includes an attribute on each message with the address of the backend server. Clients can provide this attribute to their load balancer to influence the routing decision of their subsequent requests This also requires that the client has the ability to recover when a backend server dies.

General concerns

Both of the listed approaches suffer from an issue that when, given a query over a static dataset, the ordering of the results may differ between executions. Both the provided Java implementation and the hypothetical client outlined in the client-driven routing example rely on a regular ordering of a result set between executions. This reliance is because the natural means to implement the client recovery during a query is based on the offset into the results. If the offset for a query's results varies, it is possible that clients will miss rows or receive duplicate rows.

Phoenix provides the ability to force the required ordering via the property `phoenix.query.force.rowkeyorder` in hbase-site.xml. When this property is set to `true`, it will ensure that all queries, even those without an ORDER BY clause, will generate results in the same order.

https://community.hortonworks.com/articles/9377/deploying-the-phoenix-query-server-in-production-e.html