Enabling custom rate limiters in Spark C* connector

Author: axeln@uber.com

Motivation

During usage of the spark-cassandra-connector, we have come across use cases where we need to provide a custom rate limiter which throttles based on the current state of our production environment. Instead of introducing all logic needed to the connector, which is dependent on many different internal systems, we propose the introduction of an interface that users can implement when wanting to provide a custom rate limiter. By doing so, we enable great flexibility in tuning the inner workings of the throttling which is needed when using the connector in complex environments the way Uber does.

One of the use cases where we need custom throttling is when running Spark jobs on large-scale database clusters, where we need the jobs to dynamically adapt based on external information. Having the option to provide a custom rate limiter makes it possible for us to adapt these jobs based on our internal metrics system, which enables us to increase the utilization of our Spark clusters while at the same time not affecting various important services in our systems.

An alternative to merging these changes upstream is for us to maintain a custom build of the connector and publish internally, which comes with a lot of overhead whenever there are updates in the upstream since we would need to keep updating our custom version of the connector to receive the latest update.

Approach

In order to implement support for custom rate limiters, an interface called RateLimiterProvider has been introduced. This interface represents a provider, that given a set of arguments, returns a rate limiter that callers can use in the same way as they do today. All logic related to constructing rate limiters is preserved within this provider. The returned rate limiter must implement the interface BaseRateLimiter, which consists of one method - maybeSleep. This is the only public method available in the rate limiter found in the source code today and to make this change as seamless as possible, the same method is present in the interface.

To preserve backwards compatibility, the original leaky bucket rate limiter has been refactored into a class called LeakyBucketRateLimiterProvider which instantiates and returns a LeakyBucketRateLimiter with specified configuration when no custom provider is specified or should there be an error when trying to instantiate a custom one.

Apart from the two mentioned interfaces, two new config variables have been introduced and added to the ReadConf as well as to the WriteConf. These config variables correspond to the fully qualified name of the supplied RateLimiterProvider to use for read/writes. The specified RateLimiterProvider **must** be available in the classpath, since it is instantiated during runtime. Should the instantiation of the supplied rate limiter provider fail, an exception is thrown to alert the user that can further investigate.