Design of Remote Spark Shuffle Service Reference Implementation

Problem To Solve

Apache Spark community is working on storing shuffle data on remote storage (SPARK-25299, SPARK-25299 Discussion Doc). That work (PR 28616, PR 28618) introduces several Shuffle Manager API changes to make the API more flexible to support different shuffle service implementations.

We want to create a reference implementation to demonstrate a simple streaming based remote shuffle service, and how to integrate that with the new Shuffle Manager API.

Design Options

There are multiple options to implement a shuffle service storing data on remote storage. Following are some examples (not comprehensive list of all possible solutions):

1. **Streaming Based Remote Shuffle Service**: shuffle clients send shuffle records to remote shuffle service in a streaming style, and shuffle service writes the records to different partition files based on the records' partition id.

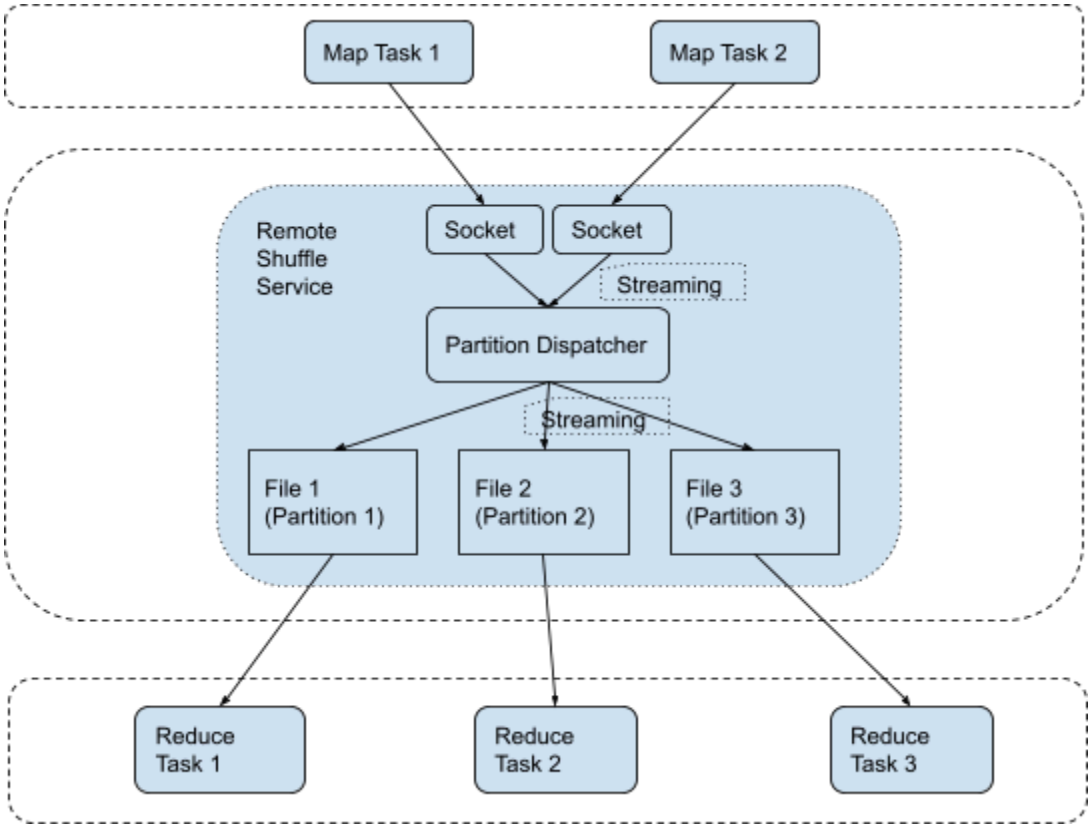
2. **Async Shuffle File Upload**: shuffle clients write shuffle records to local disk first, then upload/merge shuffle files to remote storage asynchronously.

This document will focus on the first option to illustrate how to design a streaming based remote shuffle service.

Streaming Based Remote Shuffle Service

High Level Design

Here we use a scenario where there is only one Remote Shuffle Service instance as an example to demonstrate the key idea.



Each map task will steam the shuffle data to the Remote Shuffle Service. The shuffle data stream is a sequence of shuffle records, where each record contains a partition id and record data (blob).

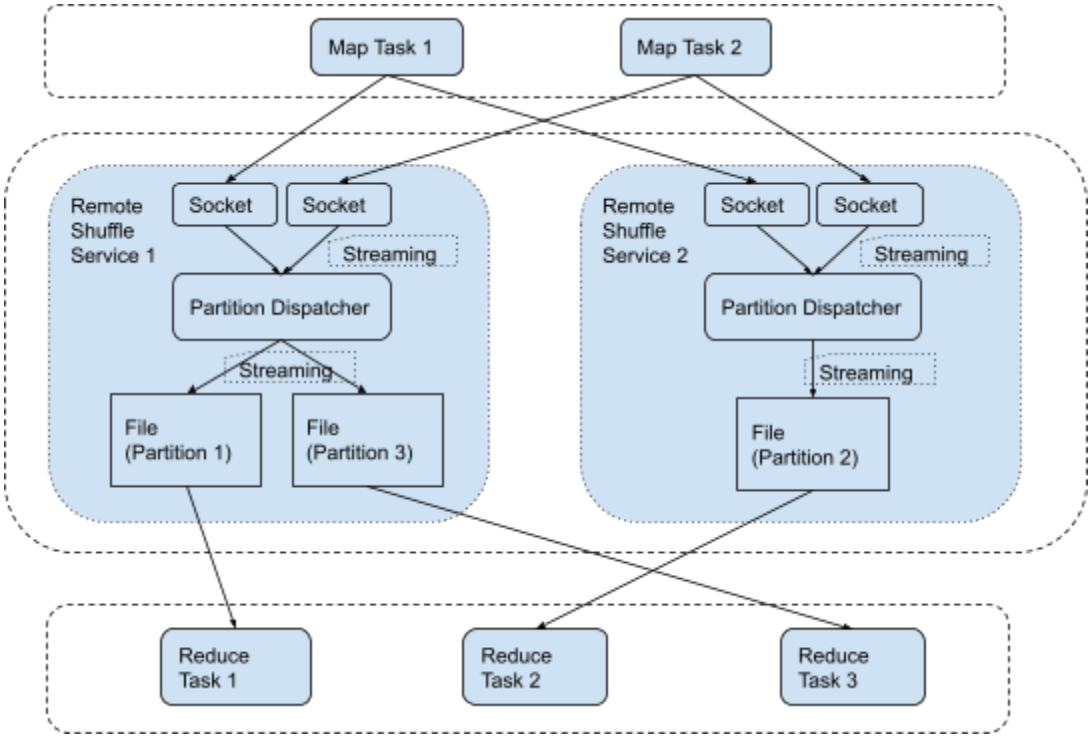
When Remote Shuffle Service gets a shuffle record from map task, it inspects the record's partition id, chooses different local file (based on that partition id), and writes record blob to the end of that file. In the upper chart example, there are 3 partitions. All records of partition 1 are written to file 1. All records of partition 2 are written to file 2. All records of partition 3 are written to file 3.

When all map tasks finish, the shuffle data will be written into multiple files. Each file contains the whole data belonging to the same partition.

When the reducer task fetches data, it connects to Remote Shuffle Service and provides a reducer id (partition id). Remote Shuffle Service will use that reducer id (partition id) to find the corresponding file and send the whole file to the reducer.

Multiple Shuffle Servers

We could add more Remote Shuffle Service instances to horizontally scale out the system. Following is an example of two Remote Shuffle Service instances.



Each map task connects to all Remote Shuffle Service instances. For each shuffle record inside each map task, the map task inspects the record's partition id, chooses a corresponding Remote Shuffle Service instance for that partition, and sends the record to that server.

Each Remote Shuffle Service instance does not know other instances. It only accepts a shuffle record from the map task, and writes the record to the file corresponding to that record's partition. This is to make sure the whole system is horizontally scalable.

In the upper example, Map Task 1 connects to both Remote Shuffle Service instances. It sends shuffle records of partition 1 and partition 3 to the first Remote Shuffle Service instance, and sends shuffle records of partition 2 to the second Remote Shuffle Service instance.

In terms of how to decide which partition goes to which Remote Shuffle Service instance, there could me multiple strategies. The easiest strategy is hash based: Remote_Shuffle_Service_Instance_Id = Partition_Id % Count_of_Servers. Then reducer tasks will also use this hash based information to determine which server to connect to fetch its data, based on its reducer id (partition id).

Performance Optimization

The Streaming Based Remote Shuffle Service writes shuffle data to remote shuffle servers. Its performance might be slowed down compared with writing to local disk.

There are many techniques we could use to optimize the performance, based on the fact that moden network hardware normally has high bandwidth but with high latency (comparing with local disk write). Following is a short list of possible optimizations:

1. Utilize high network bandwidth, e.g. write data as fast as possible and reduce waiting for server response.
2. Reuse network connections to reduce the overhead of network latency.
3. Flush file data in batches in shuffle server side

This Spark Summit 2020 Talk gives more details about how Uber optimizes their Remote Shuffle Service and runs it in production.