Distributed sorting with Apache Arrow Flight

Chengxin Ma

January 16, 2020

1 Introduction

This report includes the design of a distributed sorting application which utilizes Apache Arrow Flight for communication. The experiments demonstrates the performance of the application.

2 Design

2.1 Test data

The final goal of designing this application is to integrate it into a genomic data process pipeline, where data in the SAM format [1] is sorted.

From the perspective of sorting, the most interesting fields are `RNAME` (the name of the references sequence) and `POS` (position). They together determine the order in which records are sorted.

Thus, to simplify the prototyping work, we design a data structure with three fields: `GROUP`, `SEQ`, and `DATA`. Each record belongs to a group and has a sequence number in that group. Its data is placed in the `DATA` field.

Here is an example input file: *Input records on Node 0*, and here is an example file containing expected sorted records: *Expected sorted records on Node 0*. [2] Note that we do not necessarily need to write the output into files.

2.2 Functional decomposition

The following functional components must be implemented to complete the application.

- sorter
- sender
- receiver
- merger

[1] `https://en.wikipedia.org/wiki/SAM_(file_format)#Format`

[2] The files are misnamed. It should be *records* instead of *nums*.

- storage (maybe optional)

The *sorter* is responsible for sorting the input data in our desired order: records with smaller Group IDs are placed before those with large Group IDs. If the Group IDs of two records are the same, the secondary criteria is the sequence number.

The *sender* and *receiver* are responsible for sending (sorted) data to destination nodes and receiving data from source node respectively.

The responsibility of the *merger* is to merge the sorted data to a complete and sorted set of data.

Storage is needed when we want to temporarily store the data before further processing.

3 Implementation

There are a few dependencies among the functional components. The *sorter* and the *sender* has a clear dependency: *sender* must wait until *sorter* has sorted the input, or partitioned the inputs to groups with known destination. The *merger* must wait until the *receiver* has got all the data. The *storage* functional component must be ready before the *receiver* starts to work, otherwise *receiver* would have nowhere to put the data for further process. Thus, the current implementation runs three processes on each node: *plasma-store-server* [3], *receive-and-merge*, and *sort-and-send.* [4]

3.1 Test run

To see how the application runs, a test run has been performed. 4 nodes on Cartesius (`tcn337, tcn338, tcn339, tcn340`, we name them from Node 0 to 3 hereafter) were allocated for this test run. The data for the test run contains records of 40 groups (from `GROUP0` t0 `GROUP39`), each having 500,000 records (i.e. the total number of records is 20 million.) Each node has 5 million records in random order before the application runs, and we expected that after when the application finishes, Node 0 stores records from `GROUP0` to `GROUP9`, Node 1 stores records from `GROUP10` to `GROUP19`, Node 2 stores records from `GROUP20` to `GROUP29`, Node 3 stores records from `GROUP30` to `GROUP39`, in the ascending order.

The size of the input and output files on each node is around 130 MB.

The test run was started one by one manually, in the order of from Node 0 to 3. (Interval: about 1 second.)

[3] Instead of writing data to disk, we use *Plasma Object Store* as the means of storage.

[4] These processes have to be started in the order they were introduced above.

4 Experiments

Experiments should be conducted after the problems in the implementation part has been finished.

The goal of the experiments is to determine:

- Does the performance become better if more nodes are used?
- How does the application behave when the volume of data increase?
- Where is the bottleneck of performance?

4.1 Input size fixed, changing the number of nodes in use

4.2 Number of nodes fixed, changing the input size

5 Next steps

A Known issues

A.1 Building the project

On macOS, the `grpc` library installed via `Homebrew` (as a dependency of `apache-arrow`) seems to be problematic. The Flight server would incur a segmentation fault due to the current version (stable 1.26.0) of `grpc`.

We can make use of the existing build system of arrow to build `grpc` from source. (The build system is capable of building any missing dependency from source.) This also saves us from building missing dependencies manually on Cartesius.