Parquet Compaction using Ray - Summary

Introduction

This document outlines the work I've done towards the following goal:

Using Ray, compact many small parquet files (i.e. non-compacted) into fewer, bigger parquet files (compacted) to potentially increase read performance.

Something like this.

I worked on telemetry data on Amazon S3. There are multiple metrics here, and each has years worth of data. Initially the idea was to compact a whole month to a single file, but it quickly became apparent that memory requirements are super high for this. Compacting a day's worth of data was more feasible.

The Ray Code

This can be viewed on the GitHub repo here.
Essentially, Ray has APIs to read and write parquet files. It also has an API to specify block size (number of files) called repartition. These are used in conjunction to read, compact and writeback parquet files for each day.

A Ray cluster was set up on OpenShift; the custom notebook image and ray-operator deployment can be found here. There are kustomize scripts here, so a simple

```
> oc apply -k <directory-name>
```

will set them up on the current namespace.

To get JupyterHub running on OpenShift, I made use of the Open Data Hub Operator. Since it's my personal OpensShift cluster, and compute nodes resources are limited, I made sure only JupyterHub and applications required by it are present in the YAML.

Benchmarking Read Performance: Compacted vs Uncompacted parquet files

Preliminary check

The first check was comparing pandas dataframe read time for compacted vs non-compacted. A timer was set in Python to see how long the execution took. The read times for compacted were faster by 2-3 seconds on average.

This also verifies that the compaction process happens properly, as all the columns and row counts are displayed. This can be done by printing out the pandas dataframe object.

Trino + Trino UI

Setup

DBeaver allows you to connect to the Trino DB, tutorial can be found here.
Note: Under Driver Properties, add
'SSLVerification' with value 'none'
to bypass certificate errors.

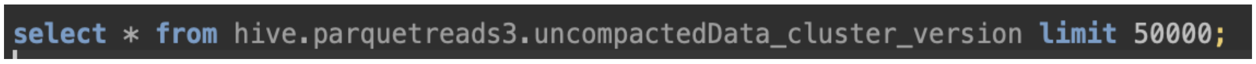
A comprehensive view of query runtime and statistics about performance can be viewed from the Trino cluster overview dashboard, found here.

Benchmarks

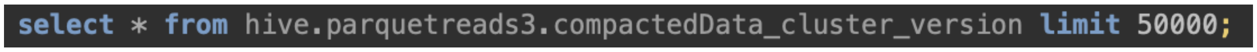
The SQL script used to run the tests is here.

The parquet files were loaded into tables, and queries were run pitting compacted vs non-compacted data.

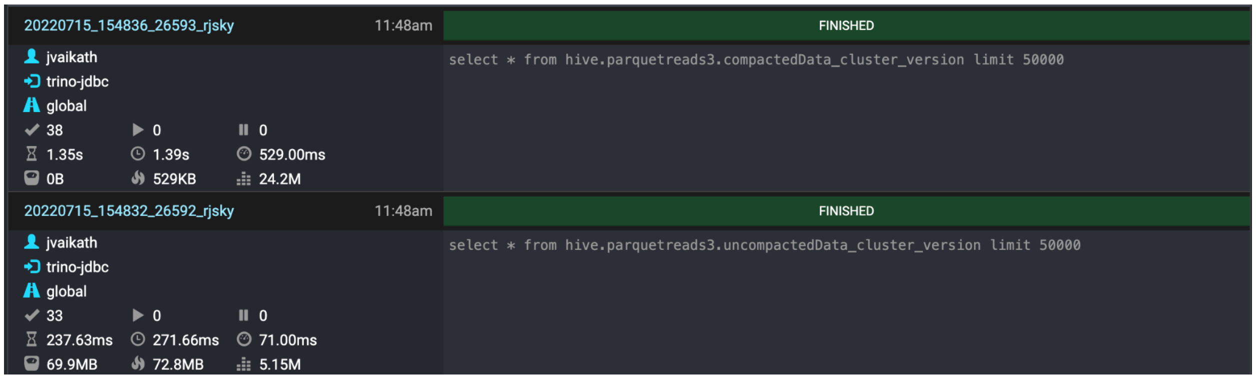
SELECT statement



```sql
select * from hive.parquetreads3.uncompactedData_cluster_version limit 50000;
```



```sql
select * from hive.parquetreads3.compactedData_cluster_version limit 50000;
```



```
20220715_154836_26593_rjsky                11:48am   FINISHED
jvaikath                                             select * from hive.parquetreads3.compactedData_cluster_version limit 50000
trino-jdbc
global
38          0           0
1.35s       1.39s       529.00ms
0B          529KB       24.2M

20220715_154832_26592_rjsky                11:48am   FINISHED
jvaikath                                             select * from hive.parquetreads3.uncompactedData_cluster_version limit 50000
trino-jdbc
global
33          0           0
237.63ms    271.66ms    71.00ms
69.9MB      72.8MB      5.15M
```

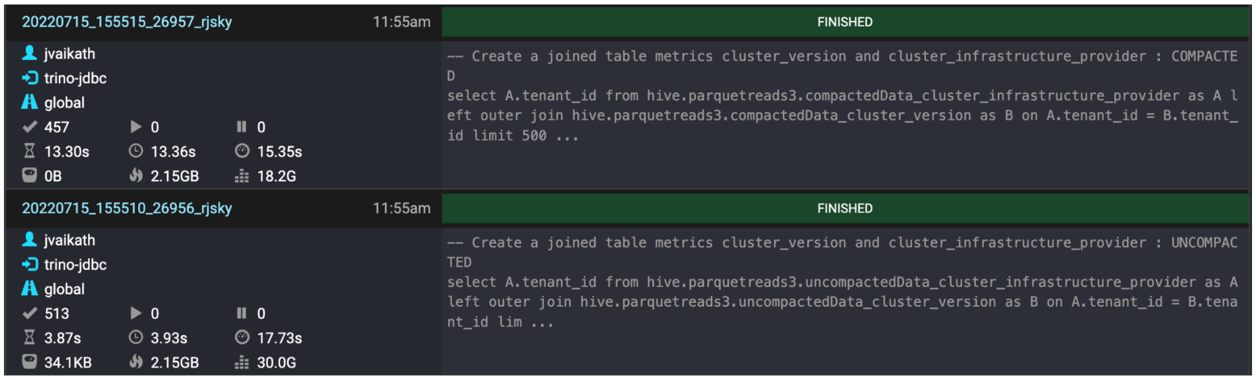
The compacted data takes much more time to fetch the rows. This could be due to the memory usage, as fetching the rows from a small number of files would be faster than reading the combined file into memory before reading the rows.

Also, any benefits gained from a single S3 read call is trumped by the parallelism; the number of files being compacted is 48. Parallelism makes this read call uncompacted faster as a result.

JOIN statement

```
-- Create a joined table metrics cluster_version and cluster_infrastructure_provider : UNCOMPACTED
select A.tenant_id from hive.parquetreads3.uncompactedData_cluster_infrastructure_provider as A left outer join
hive.parquetreads3.uncompactedData_cluster_version as B on A.tenant_id = B.tenant_id limit 50000;

-- Create a joined table metrics cluster_version and cluster_infrastructure_provider : COMPACTED
select A.tenant_id from hive.parquetreads3.compactedData_cluster_infrastructure_provider as A left outer join
hive.parquetreads3.compactedData_cluster_version as B on A.tenant_id = B.tenant_id limit 50000;
```



```
20220715_155515_26957_rjsky                11:55am   FINISHED
jvaikath                                             -- Create a joined table metrics cluster_version and cluster_infrastructure_provider : COMPACTE
trino-jdbc                                           D
global                                               select A.tenant_id from hive.parquetreads3.compactedData_cluster_infrastructure_provider as A l
457         0           0                            eft outer join hive.parquetreads3.compactedData_cluster_version as B on A.tenant_id = B.tenant_
13.30s      13.36s      15.35s                       id limit 500 ...
0B          2.15GB      18.2G

20220715_155510_26956_rjsky                11:55am   FINISHED
jvaikath                                             -- Create a joined table metrics cluster_version and cluster_infrastructure_provider : UNCOMPAC
trino-jdbc                                           TED
global                                               select A.tenant_id from hive.parquetreads3.uncompactedData_cluster_infrastructure_provider as A
513         0           0                            left outer join hive.parquetreads3.uncompactedData_cluster_version as B on A.tenant_id = B.tena
3.87s       3.93s       17.73s                       nt_id lim ...
34.1KB      2.15GB      30.0G
```

The JOIN performance really takes a nosedive. The factor of parallelism plays a role here as well. Interesting to see CPU time is slightly lower for compacted.

Using Kubeflow Pipelines (KFP) to run Parquet Compaction in Ray

The code repo can be found here.
When doing a kustomize build, add the flag:

–enable-alpha-plugins.

This is for KSOPS. So the command would be:

“kustomize build –enable-alpha-plugins .”
“kustomize build –enable-alpha-plugins . > <*filename*>.yaml”

Replace <*filename*> with your desired name, it’s just used to get it into the openshift namespace, can be deleted after you apply it.
To apply this on your openshift namespace, the output of this build can be put into a file, which can then be applied using
“oc apply -f <*filename*>.yaml”

Goal

Get the parquet compaction code to run on KFP.

Steps:

To get Kubeflow pipelines and its UI running:

1. Install ODH
2. Apply kfdef
   a. By the time this is referred, the kfdef might have been renamed. As of 5th August 2022, its here: https://github.com/opendatahub-io/odh-manifests/tree/pipelines-prototype/kfdef
   b. It should be named ml-pipelines.
   c. Copy and paste this kfdef when starting an ODH kfdef.

3. Install Kubeflow Pipelines operator from OperatorHub, required for running pipelines.

Trying to add a pipeline on Kubeflow:

1. Components can be written using this format for python
   Building Python Function-based Components | Kubeflow

Future Work

Ray and Ram requirements

The success of the compaction process done by Ray is highly dependent on memory availability. It is prone to failure if not enough memory is present, through cascading Raylet failures. Running Ray with much higher resources available and seeing the scale at which compaction is feasible might provide insight into the scale at which compaction is feasible and the resources required for it.

Compaction on a larger, leaner dataset

The data I worked on had 50-51 columns, but most of these had a majority of NULL values or barely changing static values. Preprocessing data like this could help the compaction process, but it is not trivial or repeatable for any compaction job.
If compaction could be tried on a leaner dataset with many more parquet files (maybe even merge parquet from multiple days), it could be gathered whether the benefits of compaction come through. For the data I worked on, the benefits of compaction were outweighed by the factor of parallelism.

Kubeflow Pipelines

1. Trying to access the secret from within the pipeline definition might be a better than passing aws credentials as runtime parameters.
2. The ray head pod can sometimes go into a AutoScalingExceptitonRecovery loop, this needs to be investigated.