

Improve Iceberg Spark Compaction Scalability

Problem

The Spark compaction procedure ([design doc](#)) rewrites data file groups by reading in each group's planned `FileScanTasks` through the Iceberg Spark reader and writing out replacement data files. The current reader path applies equality deletes by loading applicable delete rows into executor memory. That is efficient for delete sets of moderate size, but when many equality deletes have accumulated on a hot partition or in an unpartitioned table it can lead to executor OOM. Besides, it is not very IO efficient as the same eq deletes are likely to be read for multiple `FileScanTasks` within the same partition.

In this document, we propose an algorithm to handle the size of equality deletes that is beyond the capacity of a single server, by treating eq deletes and other data as separate dataframes and then joining them together in the end.

Goals

- Improve the Spark compaction algorithm to scale with equality deletes, being able to handle hundreds of millions eq deletes to apply on some hot partitions
- Improve resource efficiency for big partitions
- Provide a new execution path for the new algorithm so that users can switch from the normal “filter” path to the “join” path

Non-Goals

- Replace the current “filter” algorithm in Spark compaction procedure
- Add a new compaction planner
- Create a new Spark compaction procedure / API

Proposal

Key Idea

We will support all current rewrite strategies. For a given rewrite strategy, we inspect the rewrite groups generated by the respective planner. For rewrite groups that involve eq deletes with an amount above some threshold, we will switch to the join path, otherwise stay with the usual filter path.

In the normal Iceberg reader, each `FileScanTask` reads a data file together with the applicable position deletes / deletion vectors and applicable equality deletes. The position delete / deletion vector files are indexed with regards to data files, thus usually only a handful of them are needed for any given data file. But for eq deletes, often we have to read all eq delete files within the same partitions (to be more accurate, read those created by the snapshots with a sequence number higher than that of the data file's). When there are a lot of equality deletes in

a partition, the IO load is high because most of the eq deletes are being read repeatedly for each data file. If a partition receives a lot of updates, the eq deletes to read are growing quickly beyond Spark executor's memory size. That's when the current compaction algorithm hits its limit.

When the amount of equality deletes is large, we propose the following approach:

1. Read data files from a rewrite group as a dataframe, applying position deletes / deletion vectors locally in the file scan task.
2. Read all active distinct equality deletes from the same rewrite group as dataframes (more details in next section).
3. (Left-)Join data with relevant eq delete dataframes according to the eq fields, and filter out rows if data snapshot sequence number is lower than eq delete sequence number, and then write out in Iceberg format.

Considerations for Equality Delete Dataframes

We need to take into account a few facts:

Sequence Number Awareness

One same record can be updated multiple times across different snapshots over time, meaning that the same equality field tuple can occur several times with different sequence numbers. In the end, for any given tuple of equality fields, we need to know the highest sequence number among the equality deletes, and for the matching records we only keep the one with a sequence number equal or higher than it.

In practice:

1. For both data and equality deletes we need to attach sequence number as a metadata column.
2. Merge equality deletes by keeping the one with the highest sequence number:

```
mergedEqDeletes = eqDeleteRows
    .groupBy(equality key columns)
    .agg(max(delete_sequence_number).as(latest_delete_sequence_number)
    )
```

3. Left join data with eq deletes and filter

```
data LEFT JOIN mergedEqDeletes ON [eq_fields]
    .filter( latest_delete_sequence_number IS NULL
    OR
    data_sequence_number >= latest_delete_sequence_number )
```

Equality Field Set Evolution

Equality field sets can change if identifier fields are added or removed for a table over time. This means that we can find different identifier field sets across the umbrella of equality delete files in a given rewrite group, if the group contains data files before and after identifier field changes.

In order to handle such scenarios, equality delete files should be grouped according to their equality field sets, aka. schema. For each equality field set, build a merged eq delete dataframe. Data should join with each of the eq delete dataframes in the way we defined previously:

```
For each eq_field_set:
    merge eq_deletes[eq_field_set]
    join data with merged_eq_deletes[eq_field_set] and filter
```

Adaptive workflow

The local filter compaction algorithm is simple and effective, using single stage Spark jobs, but it will lead to OOM when there are too many eq deletes to load into executors' memory. The joining algorithm can handle a large amount of eq deletes, but it's significantly more complex, involving several dataframes in multiple stages that require expensive shuffling. Ideally we should use the local filter path when eq delete amount is low, and use the join path only when a local filter creates too much pressure on executor memory.

There is a trade-off to make per use case, as the same number of eq deletes can have very different memory footprint according to their schema. So we are providing a `eq-delete-join-threshold-records` parameter, and it's up to users to define the threshold to switch to the join path according to their cases. Setting it to 0 means that all rewrite groups will use the join path, and a negative value means disabling the join path completely.

Current implementation of compaction planners (bin-pack, sorted, etc) will stay the same. For each of the rewrite groups that a planner provides, we inspect every `FileScanTask` and find the highest count of active eq delete among them (it represents roughly the highest memory pressure for executors), and use the join path only if it's above the (positive) threshold.

```
if threshold < 0:
    use local filter path
else:
    for each task in group.fileScanTasks():
        remaining = threshold
        for each_equality_delete_file in task.deletes():
            remaining -= each_equality_delete_file.recordCount
            if remaining <= 0:
                use join path
                return
    use local filter path
```

The default value of `eq-delete-join-threshold-records` is -1, which leaves the new join path disabled unless users opt in.

Optimization

Here are some facts that we observe:

- In a partitioned table, the hot partitions which need to use the join path usually contain a huge amount of data and are divided into several rewrite groups by compaction planners. There are high chances that those rewrite groups involve the same umbrella of equality delete files, thus the same eq delete dataframes.

- For an unpartitioned table, rewrite groups involve global equality deletes, and it is also very likely that the same equality delete file umbrella is used across the groups.

In those cases, the common (merged) equality delete dataframes can be cached and reused across the rewrite groups that need them. This should avoid some redundant IO and computation for the merge. Cached equality delete dataframes will be unpersisted when no more groups need them.

Alternatives

We have considered several other approaches from the recent work of the community as alternatives as well, but none of them addresses exactly our issue.

Flink `ConvertEqualityDeletes`

Recently the community added a Flink maintenance task `ConvertEqualityDeletes` to convert equality deletes to deletion vectors for v3 tables. The idea is to stream data and eq deletes to a staging branch and then the `ConvertEqualityDeletes` task writes data files and (converted) deletion vectors to the target (main) branch.

But this does not solve our issues here for several reasons:

- It does not help for v2 tables because `ConvertEqualityDeletes` works only with v3 table which introduces deletion vectors.
- If we ever try to migrate to v3 table, we still have to rewrite existing eq deletes before being able to use `ConvertEqualityDeletes` with the staging + target branch mode or otherwise it will throw an error.
 - Even if we decide to use the same (main) branch for staging, the conversion is to run snapshot by snapshot through the history, which is a slow process and does not handle equality deletes added by expired snapshots
- If somehow we managed to convert equality deletes to deletion vectors, there's still a small performance gap for readers comparing to the full compaction because of the overhead of applying deletion vectors.

The conclusion therefore is that `ConvertEqualityDeletes` Flink maintenance task can be used only after we have fully consolidated equality delete files (and migrated the table to v3). In case of huge load of equality deletes which is our case, a scalable compaction algorithm is a prerequisite to the migration.

Use Cache to Improve Spark Reader

There are several PRs (eg #16956, #17868) trying to improve Spark reader efficiency, by enabling Spark executor local cache or sharing equality delete lookup tables within the same task. While these efforts may help improve the compaction throughput when equality deletes

can fit into executors' memory, it will still run into executor OOM issue when the size of equality deletes grows beyond some point.

Larger Executors or Smaller Rewrite Groups

To address OOM issue, we may try to give executors more memory, or to make rewrite groups smaller with options such as `max-file-group-size-bytes` or `max-file-group-input-files`. Neither addresses the root cause. Equality deletes in a partition apply to every data file of that partition, so the delete set loaded by each file scan task does not depend on the size of its rewrite group, and it keeps growing with the update rate. And there are some physical limits to increase executor memory.

In our tests, the current algorithm needed 100 executors of 32 GB for a workload that the join path used only 10 executors of 8 GB in a similar time. The current algorithm failed at about 100 million equality deletes, while the join path handled up to 100 billion equality deletes in stress tests.

Concerns & Questions

Deprecation of Equality Deletes in V4

It has been voted recently that in v4 specification writing new equality deletes is forbidden. A natural question is whether this proposal goes against the community's direction. The answer is no, on the contrary, it helps to remove existing equality deletes in the v2 / v3 tables that will be migrated to v4 specification in the future.

- For any v2 or v3 tables using equality deletes, upgrading to v4 will not make equality delete files disappear automatically. A compaction is still needed to make it "clean", and to handle a large delete collection efficiently we will need to implement current proposal.
- As pointed out earlier, to migrate a v2 table to v3 and then use the new Flink `ConvertEqualityDeletes` maintenance task, we need to handle existing equality deletes beforehand. In case of huge equality delete set, current algorithm is not able to handle it.
- The v2 and v3 specifications are not deprecated, so v2 and v3 tables will stay for a while. It is important to make the compaction work correctly and efficiently for them.

Complexity and Maintainability

The proposed algorithm itself is straightforward, and it should be hooked into current Spark compaction implementation at a well localised point:

- The compaction workflow does not change. Planners are untouched, and they are still deciding how to divide the workload and group files according to the compaction

strategies. We only hook a switch after the planner to decide whether to use the new algorithm.

- No API changes. All usages of Spark compaction remain the same, with only a few more options to enable the new algorithm.
- No changes will be made to the core components such as `api`, `core` and `data`. Implementation stays in the Spark scope.

Overall, the changes to the code base will be well localised and isolated, being limited to a narrow scope.

Impact on Users

The new behavior will be disabled by default with `eq-delete-join-threshold-records` set to `-1`, so there's no direct impact on current users.

With the default value, no rewrite group selects the join path and file groups run in the planned order, so the execution is identical to today. When users opt in:

- Rewrite groups using the join path run after the ones using the filter path. This ordering takes precedence over `rewrite-job-order`.
- Bin-pack output of a join group is coalesced to the expected number of output files, so file sizes may be less even than with split-based bin-packing.
- In join groups, the `cache-delete-files` option only affects position deletes, since equality deletes are applied by the join.
- The procedure fails before rewriting anything if a selected group has equality fields that the join path does not support.