FLIP-235: Hybrid Shuffle Mode

Status

Discussion thread	https://lists.apache.org/thread/hfwpcs54sm5gp3mhv7s3lr79jywo3kv4
Vote thread	https://lists.apache.org/thread/y1wb6s36k4fhyf09kc24pryx40342d4t
JIRA	~~FLINK-27862~~ - FLIP-235: Hybrid Shuffle Mode CLOSED
Release	1.16

Please keep the discussion on the mailing list rather than commenting on the wiki (wiki discussions get unwieldy fast).

Motivation

Currently in Flink, task scheduling is more or less constrained by the shuffle implementations. This will cause some problems in batch mode, mainly in the following aspects:

1. **Pipelined Shuffle:** For pipelined shuffle, in-flight data is all kept in memory. If downstream tasks do not consume data timely, upstream tasks will be blocked (i.e., back pressure). Therefore, the upstream and downstream tasks are required to be deployed at the same time, to avoid upstream tasks being blocked forever. This is fine when there are enough resources for both upstream and downstream tasks to run simultaneously, but will cause the following problems otherwise:
   a. Pipelined shuffle connected tasks (i.e., a pipelined region) cannot be executed until obtaining resources for all of them, resulting in longer job finishing time and poorer resource efficiency due to holding part of the resources idle while waiting for the rest.
   b. More severely, if multiple jobs each hold part of the cluster resources and are waiting for more, a deadlock would occur. The chance is not trivial, especially for scenarios such as OLAP where concurrent job submissions are frequent.
2. **Blocking Shuffle:** For blocking shuffle, intermediate data are all written to the local disk, and are consumed only after fully produced. Therefore, execution of downstream tasks must wait for all upstream tasks to finish, despite there might be more resources available. The sequential execution of upstream and downstream tasks significantly increase the job finishing time, and the disk IO workload for spilling and loading full intermediate data also affects the performance.

We believe the root cause of the above problems is that shuffle implementations put unnecessary constraints on task scheduling.

To solve this problem, we propose Hybrid Shuffle, a new shuffle implementation that minimizes the scheduling constraints. The only constraint is that upstream tasks should be scheduled for execution before downstream tasks. With Hybrid Shuffle, Flink should:

1. **Make best use of available resources.** Ideally, we want Flink to always make progress if possible. That is to say, it should always execute a pending task if there are resources available for that task.
2. **Minimize disk IO load.** In-flight data should be consumed directly from memory as much as possible. Only data that is not consumed timely should be spilled to disk.

Public Interfaces

The config option **"execution.batch-shuffle-mode"** will have an additional valid value ***ALL_EXCHANGES_HYBRID*** for activating this feature.

We don't see any other public API changes that are necessarily required at the moment. However, there are a few tuning knobs that we would like to decide whether to expose to users after finishing and trying out this feature. E.g., memory usage threshold for triggering spilling, number of buffers a SubpartitionFileReader should read ahead, etc. Ideally, we want to expose only a minimal set of necessary knobs, for better usability.

Proposed Changes

The proposal consists of changes in two parts:

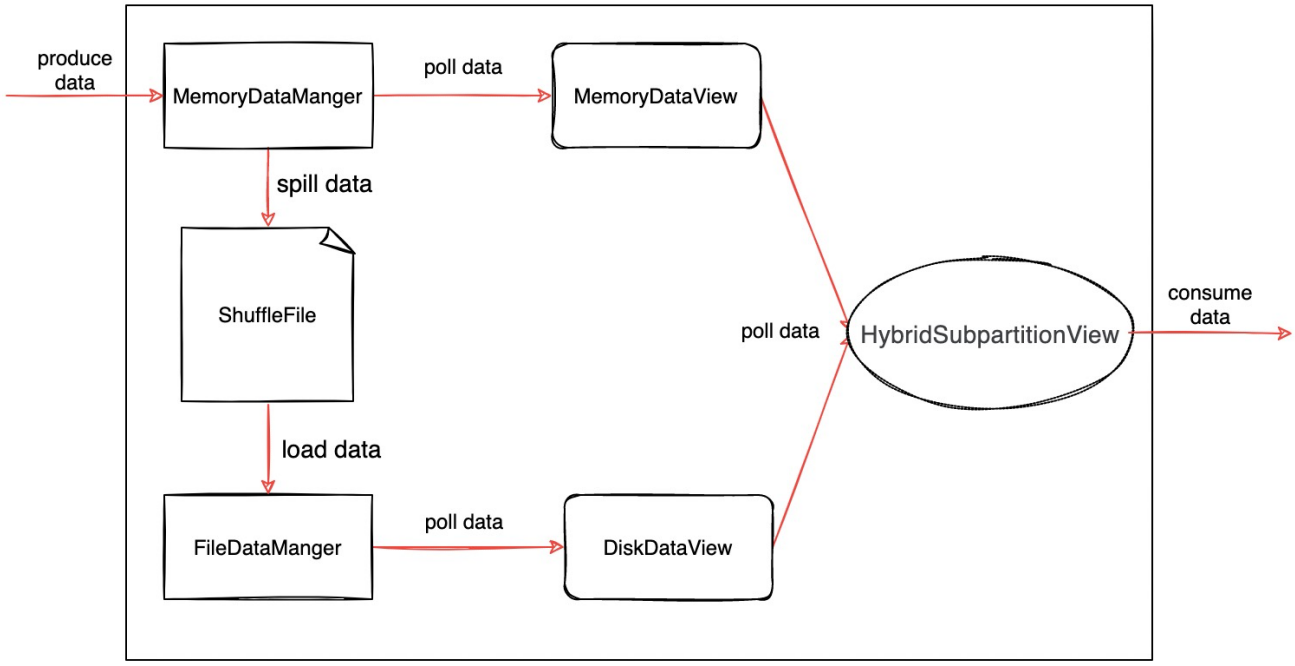
1. Introduce a new HybridResultPartition
2. Adapt task scheduling to make best use of the new shuffle

HybridResultPartition

HybridResultPartition needs to meet the following two requirements:

1. Intermediate data can be consumed any time, whether fully produced or not.
2. Intermediate data should be consumed directly from memory as much as possible. The amount of data spilled to disk should be minimized.

Components & Workflows



The above figure shows the main components of HybridResultPartition:

1. MemoryDataManager: Responsible for storing data in memory, maintaining their meta, and deciding when and which data to spill.
2. FileDataManager: Responsible for preloading data from disk files in preparation for the downstream consumption, as well as deciding when and which data to preload.
3. SubpartitionView: The interface that downstream tasks pull data from, which determines where (memory / disk) the next data should be consumed from for a specific subpartition .
4. MemoryDataView & DiskDataViewInterfaces for HybridSubpartitionView to find out what data exists in memory / disk and polling the data.

Upstream workflow:

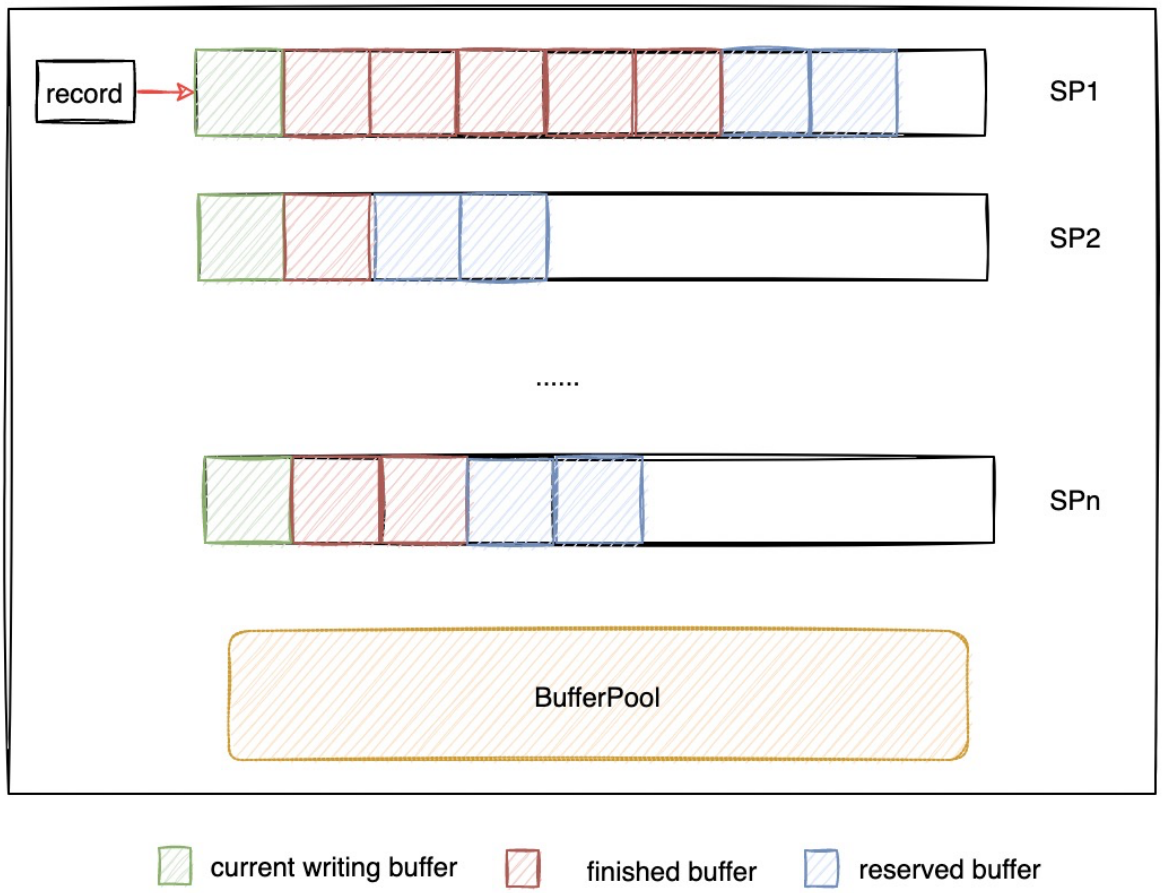
1. Data written to HybridResultPartition is firstly stored and managed by MemoryDataManager, where it is visible to the downstream.
2. As the amount of data in memory increases, MemoryDataManager will decide to asynchronously spill some of the data to the ShuffleFile on disk.
3. The spilled data will be loaded back later for the downstream to consume. FileDataManager continuously estimates which data should be consumed next and loads them from the ShuffleFile.

Downstream workflow:

1. When a downstream task is connected, a corresponding HybridSubpartitionView will be created.
2. On each request of data, HybridSubpartitionView checks MemoryDataView and DiskDataView whether the desired data exists. If the data exists, it polls data from the corresponding data view.
    a. It is guaranteed that the desired data would not appear in both MemoryDataView and DiskDataView, because spilled data will be removed from MemoryDataManager atomically.
    b. It is possible that the desired data appear in neither MemoryDataView nor DiskDataView. The desired data may not be produced yet, or has already been spilled from MemoryDataView but not yet loaded by DiskDataView. In such cases, the downstream task will be notified later when data becomes available.

MemoryDataManager

The MemoryDataManager manages the data in the memory as shown in the following figure:



MemoryDataManager consists of a branch of buffer queues each corresponds to a subpartition, and a buffer pool shared by all subpartitions. New records are written to the last buffer of the corresponding subpartition queues. If there's no buffer in the queue, or the last buffer does not have enough space for the incoming record, a new buffer will be polled from BufferPool and added to the end of the queue. All buffers except for the last one in a queue are considered finished, which means no more data will be written to them, thus they are ready for consumption and spilling. A buffer that is consumed / spilled will return to the pool.

Spilling is triggered when the number of available buffers in the pool reaches a certain threshold (e.g., 20% of the capacity). To maximize the chance that downstream tasks consume data directly from memory and reduce the disk IO load, only a subset of finished buffers (e.g., 20% of the capacity) that is less likely to be consumed in short will be spilled.

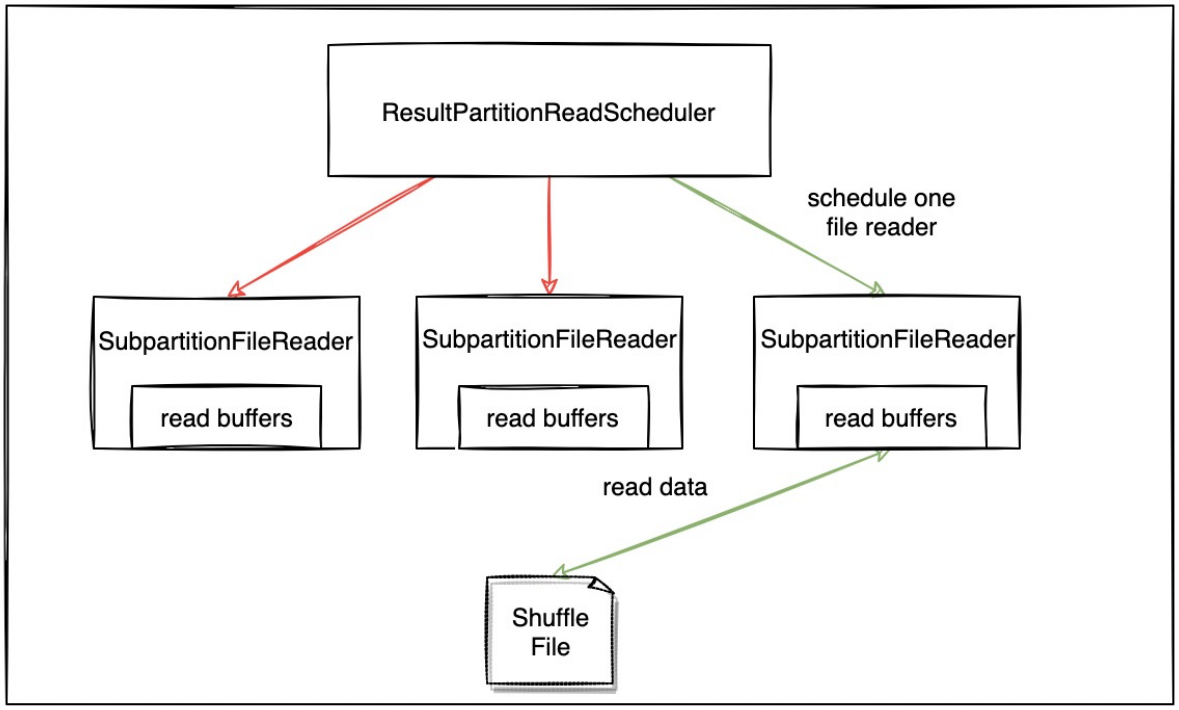
To be specific, the following buffers should be spilled with higher priority:

1. Buffers of subpartitions that the downstream tasks are not yet connected.
2. Buffers carrying data whose offset is more behind the consuming offset.

An alternative strategy is to spill all data to the disk, as soon as the buffers are finished. The benefit is that, in case of failover, upstream tasks do not need to be restarted, as all the data it produces is persistent. However, that introduces extra IO workload, and may block the upstream task if the disk IO becomes the performance bottleneck. In the first version, we will consider a minimum spilling strategy. We may consider to also support a full spilling strategy in future, for scenarios where the failover cost is more crucial.

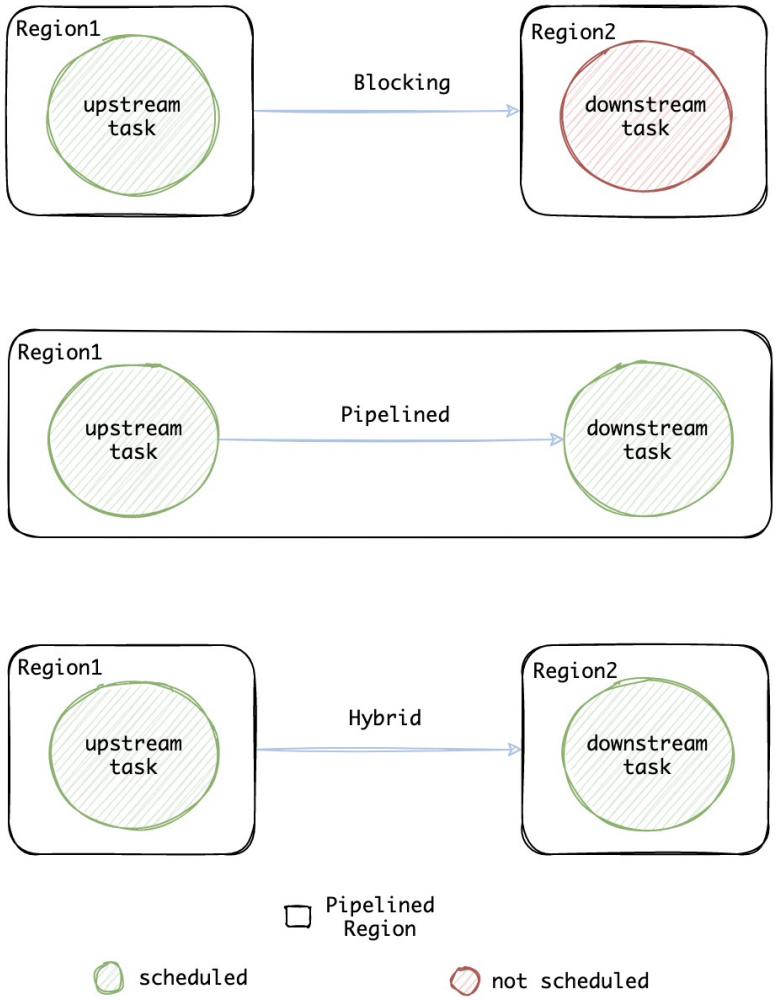
FileDataManager

The FileDataManager manages the data in the disk as shown in the following figure:



To improve the disk access performance, we apply the same IO scheduling mechanism from Sort-based Blocking Shuffle, which maximizes the sequential disk access over random access. For each connected downstream task, there will be a corresponding SubpartitionFileReader, which is responsible for preloading the spilled data from the shuffle file on the disk. And the ResultPartitionReadScheduler will be responsible for scheduling each SubpartitionFileReader for reading, according to the file offset they want to read.

Task Scheduling



We introduce Hybrid as a new ResultPartitionType in addition to Blocking and Pipelined. As shown in the figure, depending on the context, Hybrid result partition needs to be treated in the same way as Blocking or Pipelined.

- When calculating PipelinedRegions, we are looking for tasks that “must be pipelined”. Since Hybrid result partition doesn’t have such constraint, it should be treated in the same way as the Blocking result partition.
- When the Scheduler determines which tasks are schedulable, we are looking for tasks that “can be pipelined”. In that case, Hybrid result partition should be treated in the same way as Pipelined result partition.

To better illustrate such differences, we’d also refactor the ResultPartitionType to reflect the different semantics of “must be pipelined” and “can be pipelined”.

In this way, we can benefit from the reduced scheduling constraints with barely any changes to the Scheduler and default PipelinedRegionSchedulingStrategy.

Please notice that the first version of Hybrid Shuffle does not support slot sharing. While the job can execute with a single slot (like blocking shuffle), it may take more slots than its max parallelism if provided. See Know Limitations for more details.

Implementation Plan

The implementation includes the following parts, among them, 2 depends on 1, but 2, 3 and 4 can be parallel, and 5 relies on them.

1. Refactor ResultPartitionType
    - The first part is to refactor ResultPartitionType, so that decoupling scheduling logic and partition release logic.
2. Make Flink supports HYBRID type edges
    - The second part is to make StreamGraph and scheduler support HYBRID type edges.
3. Introduce MemoryDataManger
    - The third part is to introduce MemoryDataManger and spill strategy.
4. Introduce read and write disk related components
    - The fourth part is to support writing data to shuffle file and reading from it.
5. Introduce HybridResultPartition related components
    - Finally, Introduce HybridResultPartition and HybridSubpartitionView.

Compatibility, Deprecation, and Migration Plan

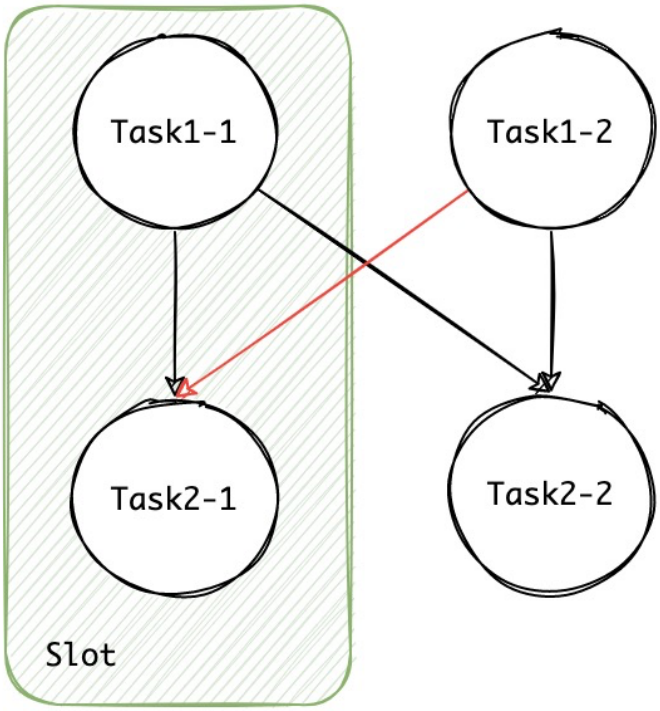
There should be no compatibility breaking changes.

We would first make this an opt-in feature that can be activated by a config option. In the long term, we hope to make it the default shuffle implementation for batch mode, once we are confident that it has stabilized.

Known Limitations

Slot sharing is not supported

The first version of Hybrid Shuffle does not support slot sharing. Precisely, it forces each Slot Sharing Group (SSG) to contain only one task. The reason is shown in the following figure.



If Task1 and Task2 are in the same SSG, once Task1-1 gets a slot, Task2-1 will also be scheduled for execution. However, if there's no extra slot for Task1-2 and Task2-2 to execute, Task2-1 may not be able to consume data from Task1-2, thus never finishes. Addressing this issue requires extra careful design changes to the scheduling mechanism, thus is scoped out from this FLIP.

Consequently, users may find jobs with Hybrid Shuffle using more slots (if available) than with Pipelined or Blocking Shuffle, due to slot sharing disabled. To some extent, this is also an advantage that Flink can leverage more available resources to speedup the execution when it's possible.

Test Plan

The changes will be covered by unit, integration and e2e test cases.