Intermediate result lifecycle management

This document designs how Flink will manage intermediate results and what their lifecycle should be.

[JIRA](https://issues.apache.org/jira/browse/FLINK-12069)

# Problem statement

Operators in Flink produce output which can be consumed by downstream operators. The collective output of an operator is called the intermediate result.



When executing the operators in parallel the intermediate result is further split up into intermediate result partitions where each parallel sub task of an operator produces an intermediate result partition. The set of all intermediate result partitions forms the intermediate result.

Flink currently supports two types of ResultPartitionType (technically there are more but atm we only need these two):

1. **Pipelined:** The result partition can be directly consumed as soon as data has been produced. The result partition data is kept in memory so that back pressure will be created if too much memory is used. Moreover, the data is not persisted. This partition type is used by streaming and batch applications.
2. **Blocking:** The result partition is first completely produced and persisted before downstream consumers can start reading from it. The current implementation SpillableSubpartition tries to keep data in memory before it spills to disk. This partition type is used by batch applications.

The nice property of blocking result partitions is that they are persisted (usually to disk) from where they can be consumed multiple times. This is beneficial because we can produce a result partition once and let multiple downstream tasks read the same result. Moreover, we can use persisted result partitions for faster recoveries because we don’t have to recompute them.

At the moment, intermediate result partitions are released by the TaskExecutor after they have been consumed once.

In order to enable proper fine grained recovery it is required that blocking result partitions can be consumed multiple times. By having intermediate results persisted one does not need to reschedule the complete topology.

Moreover, by allowing result partitions to out live jobs, it could be possible to share results between different jobs. This could be beneficial for ad-hoc queries as they appear with [interactive programming](https://cwiki.apache.org/confluence/display/FLINK/FLIP-36%3A+Support+Interactive+Programming+in+Flink) ([detailed design document](https://docs.google.com/document/d/17twjcQn70rJnVCXcr74AL44HY3jLeT1leC9rAFsluFg/edit)).

# Solution proposal

In order to make a blocking result partition consumable by multiple downstream operators as well as to use it for recoveries, the decision when to release a blocking result partition needs to be made by the JobMaster which has an overview of the job execution. The JobMaster knows when all consumers of a result partition have terminated and, hence, when the result partition can be released. It also knows when a failover region has been completely executed and, thus, when result partitions are no longer needed for recovery.



In order to avoid that the ResourceManager releases a TaskExecutor which still contains result partitions but no more allocated slots, the TaskExecutors report the set of stored result partitions to the ResourceManager. Only if a TaskExecutor does not contain any result partitions, it can get released. See [FLINK-10941](https://issues.apache.org/jira/browse/FLINK-10941) for more information.



A problem of moving lifecycle management to the JobMaster is what happens with the result partitions if the TaskExecutor loses its connection to the JobMaster? The JobMaster TaskExecutor connection might be interrupted for several reasons: Network problems, JobMaster died, the TaskExecutor died, etc. In all failure scenarios, Flink must make sure that the TaskExecutors don’t amass orphaned result partitions which might fill up local disks up to the point where the TaskExecutor’s machine is no longer usable.

In order to solve this problem, we propose two mechanisms:

1. **Heartbeat based clean up**: Delete all partitions belonging to a job when the connection to the JobMaster times out.
2. **Safety net**: Fail fatally if the TaskExecutor’s disk is full and register a shutdown hook to delete the result partition directory.

### Heartbeat based clean up

TaskExecutor execute Tasks only as long as they have an open connection to the JobMaster. If the connection times out then all running Tasks belonging to this job get cancelled. Similarly, we propose to do the same for result partitions: As long as the job runs, the JobMaster needs to keep an open connection to all TaskExecutors which have result partitions stored. If the connection is lost, then the TaskExecutor will delete all result partitions belonging to this specific job. This will ensure that there cannot be orphaned result partitions.

Keeping an open connection to a TaskExecutor which has result partitions stored will require changes to when to close the connection on the JobMaster and TaskExecutor side. Concretely, before closing a TaskExecutor connection, the JobMaster

* needs to make sure that it has no more allocated slots from this TaskExecutor
* and that for each result partition ShuffleDeploymentDescriptor.hasLocalResources either returns None or Some(id) with id not being the TaskExecutor’s ResourceID

In the future, we might also introduce a grace period before the result partitions are deleted. That way, the TaskExecutor would have a bit of time to re-register at the JobMaster without losing all produced results. For the moment, we can assume that this grace period is always 0.

### Global result partitions

Since we want to share results across different jobs (for interactive programming), result partitions cannot always be bound to the lifetime of a job. There must be a means to tell Flink that certain result partitions should become *global* result partitions. Global result partitions can outlive the lifetime of a job. If an external shuffle service is used, then they can even outlive the cluster’s lifetime.

The idea is that a job result partition can be removed from the usual result partition lifecycle management in order to become a global result partition. This should only happen if the job has been successfully executed (e.g. reached the FINISHED state). Therefore, we propose that the JobMaster disconnects from the TaskExecutor when the job has reached the FINISHED state. When it sends the disconnect call, it also tells the TaskExecutor which active result partition should become global result partitions. All other result partitions will be removed.



Once a result partition has become a global result partition, it is the responsibility of the client/user to clean it up. Flink will only delete it in case that the disk should becomes full. See [Safety net](#_cjyi24v8rmws) for more information.

In order to release a global result partition, Flink needs to add a REST call which can be called by the client. The REST call will send an RPC to the ResourceManager which will forward it to the respective TaskExecutor where the result partition is released.

**Important:** The call to release a global partition can fail at various stages (e.g. REST endpoint is not reachable, ResourceManager is not reachable, TaskExecutor is not reachable). For the sake of simplicity and because the client needs to handle failed attempts anyway (if the REST endpoint is not reachable), we assume that the client retries failed release attempts until they succeed. If a client does not release a global result partition, then it can clutter the cluster up to the point that the cluster is no longer usable for other jobs.

In the future, we might think about a caching mechanism for partition release requests on the ResourceManager. That way, one could solve the problem when the TaskExecutor is temporarily not reachable from the ResourceManager.

Additionally, we might add a tool for dev ops to list and release global result partitions in case that a client forgot to delete them. This could even be triggered via the web UI.

### Safety net

Since clients might forget to release global result partitions, we need to make sure that TaskExecutors don’t go permanently into an unhealthy state. An unhealthy state would be if a TaskExecutor’s disk is so full that no more Tasks can be executed on it (no more disk space for spilling).

In order to prevent this situation from happening/be able to recover from, we propose to add a shutdown hook which deletes the result partitions directory. Additionally, we propose to fail fatally if the TaskExecutor can no longer spill to disk because of too little disk space. In environments which support dynamic TaskExecutor starts like Yarn, Mesos, Kubernetes, this should lead to self healing of the cluster.

Since a full disk might surface at various places and in various forms (no more swap space, writing a blocking result, spilling to disk while sorting/building a hash table), we need to find a way to treat no more disk space problems consistently. This might entail that we need to properly distinguish between user code and system faults and that we analyze system faults whether they are recoverable or not (resulting in a fatal error). The proper distinction might be out of scope for the partition lifecycle management effort and we might settle with a best effort solution. If this is the case, then using global result partitions is a potentially dangerous feature which might render the cluster unusable.

# Related shuffle service API

Proposal for further extension to include into [FLIP-31: Pluggable Shuffle Manager](https://cwiki.apache.org/confluence/display/FLINK/FLIP-31%3A+Pluggable+Shuffle+Manager) which adds details of result partition lifecycle management.

## Partition lifecycle

Relevant classes:

interface ShuffleDeploymentDescriptor {

Optional<ResourceID> hasLocalResources();

}

class PartitionShuffleDescriptor {

JobID, IntermediateDataSetId, IntermediateResultPartitionID, ResultPartitionType, TTL?, ...

}

class ProducerShuffleDescriptor {

ConnectionID, ExecutionAttemptID,ResourceID

}

interface ShuffleMaster {

ShuffleDeploymentDescriptor allocateNewPartition(

PartitionShuffleDescriptor, ProducerShuffleDescriptor);

void releasePartitionExternally(ShuffleDescriptor);

}

Job master allocates the partition after its producer has got the slot and before deploying it. Job master removes the partition when all consumers are done and the partition is not global.

ShuffleDeploymentDescriptor.hasLocalResources tells to job master that shuffle service keeps some partition resources locally to serve consumers and it has to keep TM connection until all consumers are done.

Assert partition does not exist in ShuffleMaster.allocateNewPartition.

## ResultPartitionWriter

ShuffleService {

ResultPartitionWriter createResultPartitionWriter();

InputGate createInputGate();

void releasePartition(ResultPartitionId);

Collection<ResultPartitionID> unreleasedPartitions();

}

### close() before Task is done

If Task finishes, before moving to done state, it should call ResultPartitionWriter.close() for all produced partitions to wait and make sure that all relevant operations are flushed internally in ShuffleService. It does not mean that partition is completely released internally and does not occupy any local resources in TaskExecutor.

The internal behaviour depends on ShuffleService and the partition type:

* Local PIPELINED partition: flush into buffers and release partition immediately
* Remote PIPELINED partition: wait consumer confirmation to receive all data and then release partition immediately
* BLOCKING partition: wait all data to be flushed into local file, partition is not immediately released. Release happens only explicitly by calling ShuffleService.releasePartition. As discussed above, this happens either implicitly locally:
  + JM heartbeat fails for running producing job
  + JM says not to keep it after closing the connection
  + or explicitly: user releases it over REST API and RM sends RPC call to TM

### cancel() if Task is canceled

If Task is canceled by JobMaster, it can immediately abruptly call ResultPartitionWriter.cancel(). The cancel may internally skip flushing and release partition asap.

## Task manager lifecycle

ShuffleService {

ResultPartitionWriter createResultPartitionWriter();

InputGate createInputGate();

void releasePartition(ResultPartitionId);

Collection<ResultPartitionID> unreleasedPartitions();

}

ShuffleService.unreleasedPartitions() returns all partitions which still occupy some local resources in TaskExecutor due to possible on-going operations, e.g. communication with consumers etc. A partition can stay unreleased for some time even after producing Task has closed it and moved to **done** state.

TaskExecutor includes ShuffleService.unreleasedPartitions() into its heartbeat with the ResourceManager. ResourceManager can release TaskExecutor only if this collection is empty (no unreleased partitions in this TaskExecutor), additionally to the normal check that all slots are freed. This would prevent premature release of TaskExecutor container observed currently in certain batch scenarios (quick fix for this problem is addressed in [FLINK-10941](https://issues.apache.org/jira/browse/FLINK-10941)).

## Further Optimisation for fine-grained batch recovery in case of job master failover

enum ShufflePartitionState {

NOT\_EXISTING,

FINISHED

}

interface ShuffleDeploymentDescriptor {

ShufflePartitionState getPartitionState();

}

interface ShuffleMaster {

Map<ExecutionAttemptID, ShuffleDeploymentDescriptor>getPartition**(**

PartitionShuffleDescriptor**);**

ShuffleDeploymentDescriptor allocateNewPartition(

PartitionShuffleDescriptor, ProducerShuffleDescriptor);

void releasePartitionExternally(ShuffleDescriptor);

}

Job Master actions:

* Before execution graph scheduling, check whether produced partitions are already available:
  + Create PartitionShuffleDesriptor from ExecutionVertex and IntermediateResultPartition
  + ShuffleMaster.getPartition(PartitionShuffleDesriptor).getPartitionState()
  + State is **NOT** ShufflePartitionState.**FINISHED**:
    - if after Job restart: ShuffleMaster.releasePartitionExternally(ShuffleDescriptor)
    - schedule producer and consumer normally
    - when producer gets slot: ShuffleMaster.allocateNewPartition(PartitionShuffleDescriptor, ProducerShuffleDescriptor)
  + State is ShufflePartitionState.**FINISHED**: ShuffleDeploymentDescriptor from getPartition can be used to deploy consumer without producer
* If the producer has not been deployed and the consumer fails with the partition exception, deploy producer to reproduce the broken partition with the next attempt
* When consumer task reports done state, remove the partition if configured: ShuffleMaster.releasePartitionExternally(ShuffleDescriptor)