Failure Recovery in Trino

The current all-or-nothing architecture makes it significantly more difficult to tolerate faults. The likelihood of a failure grows with the time it takes to complete a query. In addition to increased likelihood of a failure the impact of failing a long running query is much higher as it often results in a significant waste of time and resources.

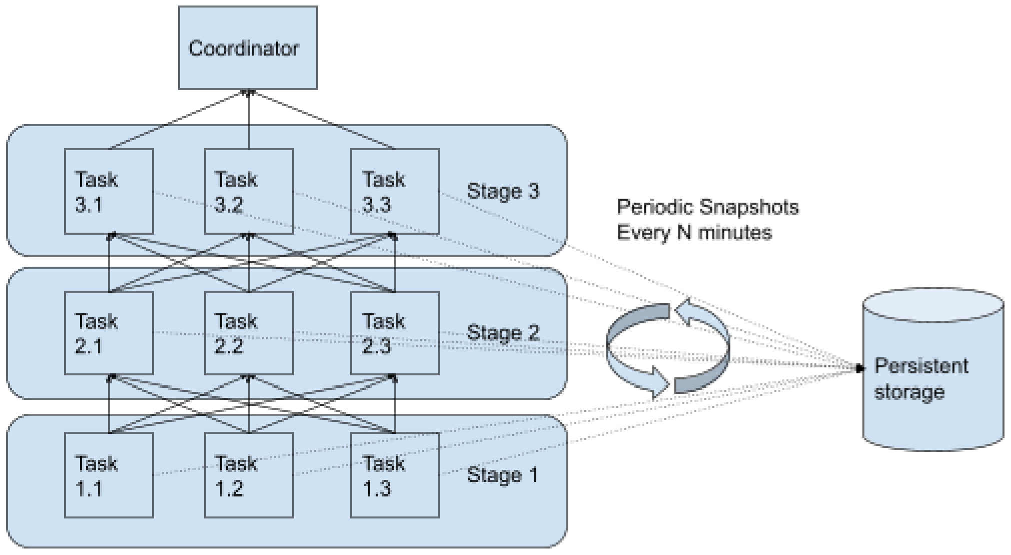
Due to the nature of streaming exchange in Trino all tasks are interconnected. A failure of any task results in a query failure. To support long running queries Trino has to be able to tolerate task failures.

Approaches Considered

State Checkpointing

The idea of this approach is to suspend query execution once in a while and store the current state of all tasks and operators to a persistent storage. In case of a fault a saved snapshot can be restored and a query can be resumed.

Usually this approach is taken under an assumption of the state being “not that large” so the snapshots can be taken at high enough frequency.



Pros

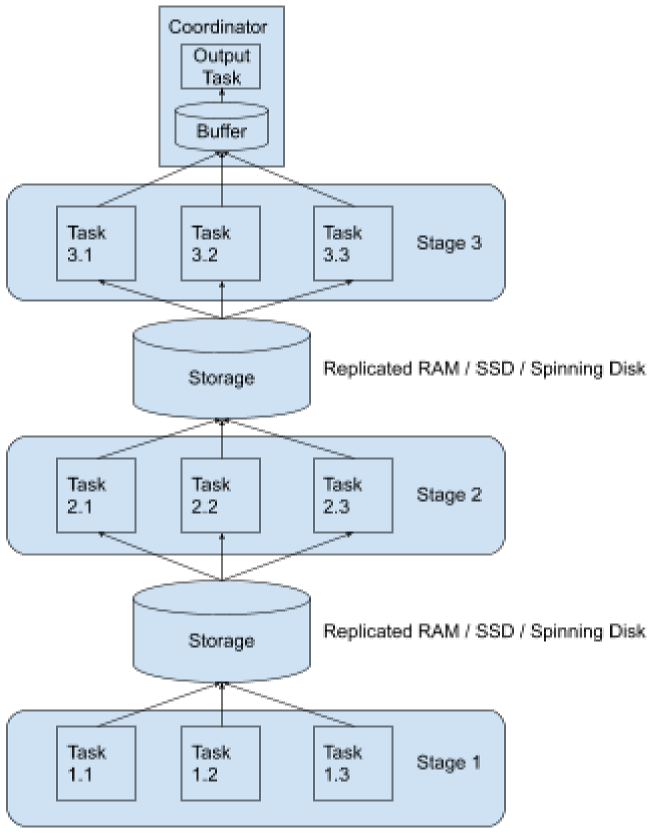
- Exchanges remains “cheap”
- Partitions can be assigned dynamically

Cons

- All tasks must be restarted from a checkpoint in case of a failure
  - Might be impossible to make progress when the system experiences faults more often that a snapshot can be taken. For example when running on spot instances with a 2 minutes withdrawal notice a snapshot has to be taken more frequently than once in 2 minutes.
  - When the workload grows so does the state. At some point taking frequent snapshots might become unmanageable.
- Doesn’t help with running queries that require more memory than available in a cluster as it doesn’t enable more granular partitioning and an iterative task execution
- Doesn't provide additional capabilities to support speculative execution
  - “fault” is often a non-binary thing. A task that experiences a significant slowdown is technically not “failed”, yet it might be better to speculatively start another copy of that task to avoid waiting for an instance that might never finish.
- Doesn’t enable adaptive query optimizations as fundamentally the model of execution remains the same (entire query is executed at once)

Exchange Data Buffering

The idea of this approach is to store intermediate data produced by tasks to a “distributed buffer”. This allows restarting a single task in case of a failure (assuming deterministic partitioning at each exchange boundary).



Pros

- Each task can be restarted independently
- Multiple instances of the same task can execute simultaneously
- Queries can be scheduled partially and re-optimized at exchange boundaries
- Allows partitioning into a number of partitions that is much higher than the number of nodes available in the cluster. This would allow more granular execution effectively increasing the distributed memory limit.

Cons

- Data always has to be partitioned in a deterministic way. Dynamic partitioning is challenging / not possible.
- Exchange is becoming more expensive. Execution techniques such as *mark_distinct* might not be practical.

Both approaches are not self exclusive and can be implemented both if needed. However the second approach seems to have much higher ROI. In addition to failure recovery it offers more granular scheduling that would effectively improve support for high memory queries and open opportunities for adaptive query optimizations (such as adaptive join reordering, adaptive join type selection, adaptive partitioning tuning).

The first approach is also more risky as taking distributed state snapshots can be rather difficult to implement and it is not very well suited for the environments with high number of processing nodes (that exponentially increases likelihood of a fault) and large internal state (that increases time needed for a distributed snapshot to be taken).

In the remaining part of this document we are going to focus on the second approach, Exchange Data Buffering.

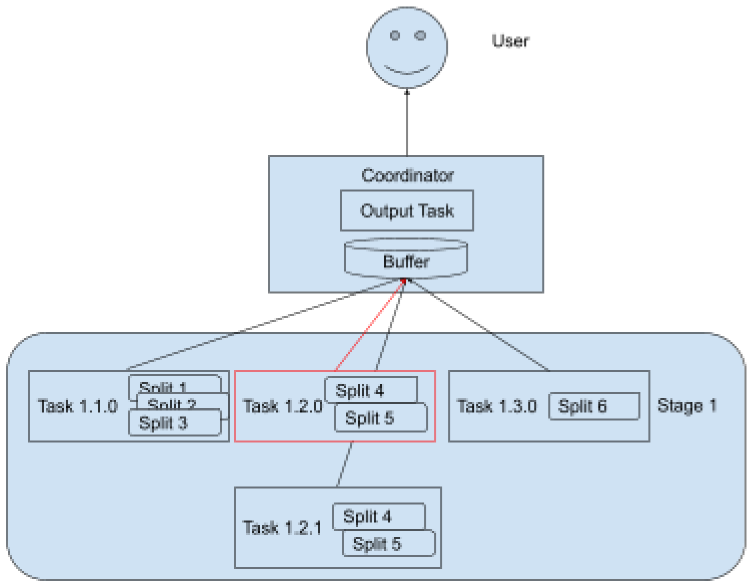
Challenges

Deterministic Tasks

With the current model of execution every task is expected to be executed exactly once. Thus the system doesn't try to enforce determinism of task inputs.

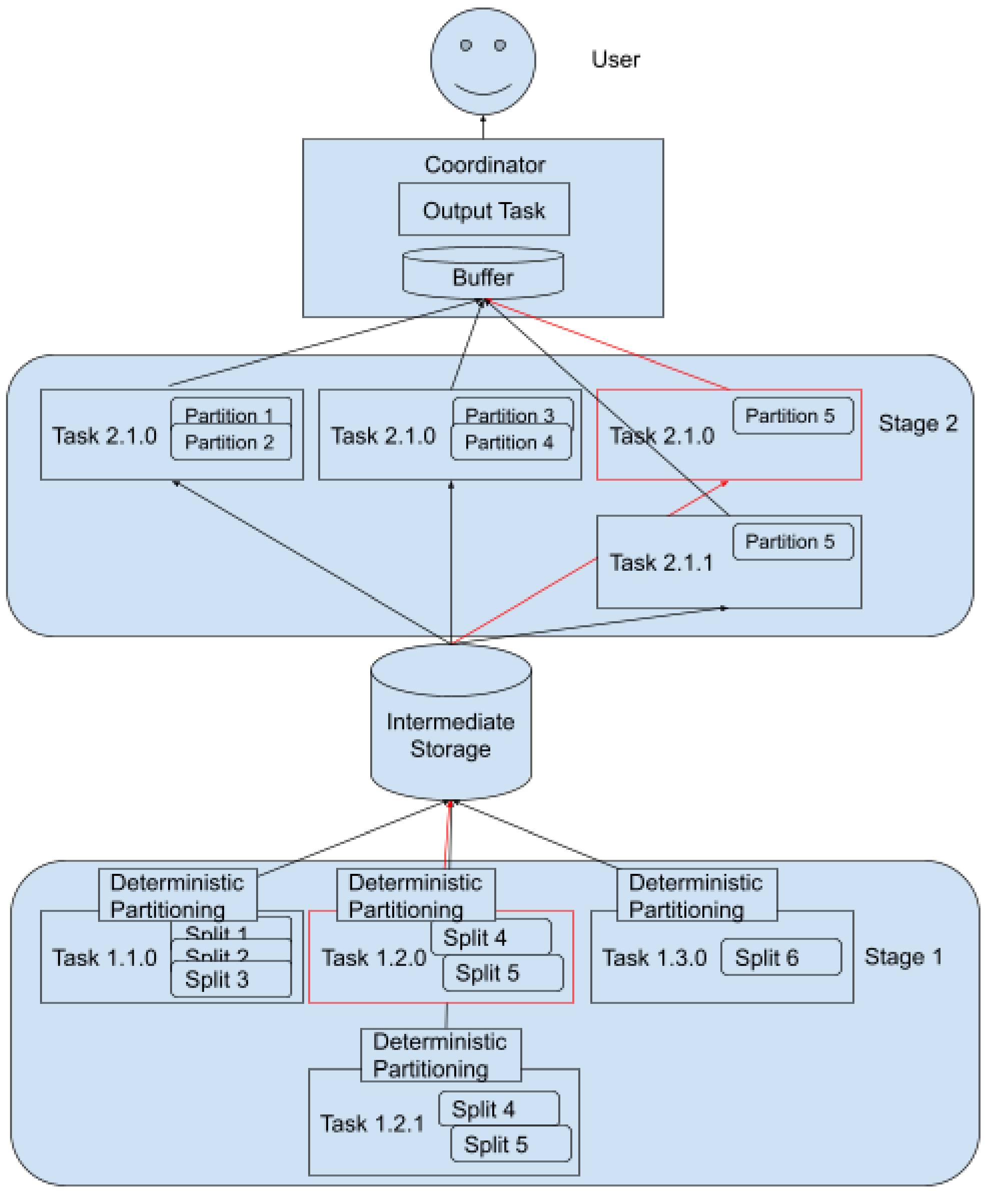
Currently splits are assigned to tasks randomly and round robin partitioning (or scaled writer partitioning) assigns rows a task in a random fashion. To enable task level retries inputs have to be assigned in a deterministic way.

For leaf stages, splits should be assigned in a deterministic way to make sure different instances of the same task produce the same output.



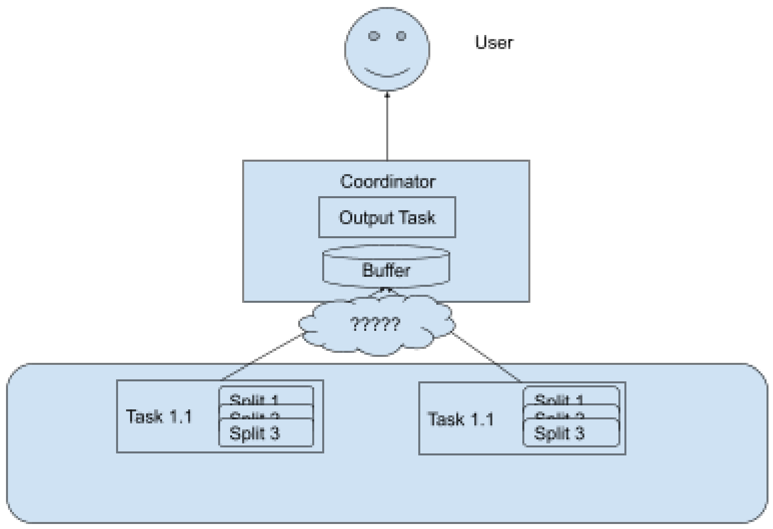
In case of a failure task must be restarted with the same set of splits. Partial output of a failed task must be discarded.

For intermediate stages the situation is somehow similar. Every row from an upstream stage has to be assigned to a certain partition in a deterministic way. Partitions have to be assigned to a tasks in a deterministic way as well.

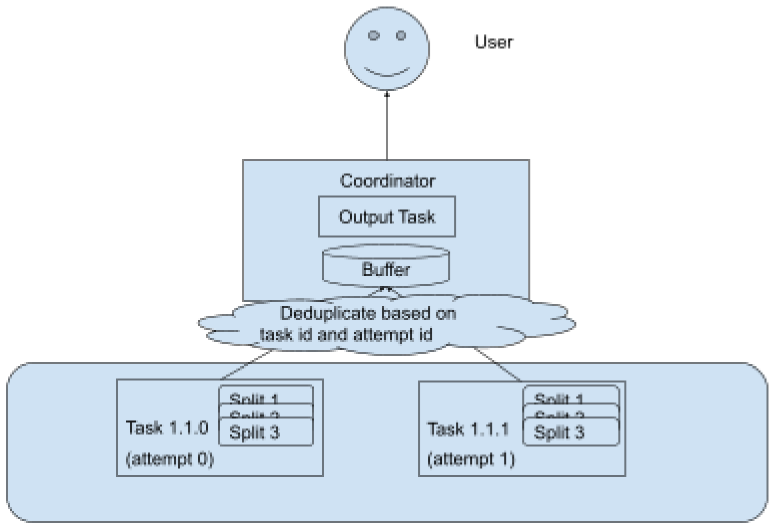


Restartable Tasks

In a distributed system "fault" has a rather broad definition. When a task fails to respond it is impossible to say if it has crashed or became unreachable. Even if a task doesn't fail to respond, but experiences a significant degradation of performance it may still be considered “failed” . Thus it is important to be able to safely schedule multiple instances of the same task without worrying about side effects.

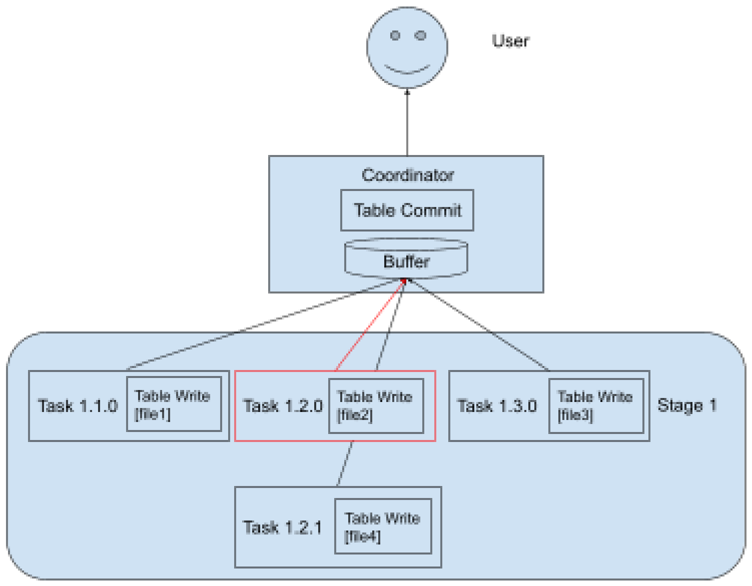


At this point only a single instance of a task can be active in a cluster. To support failure recovery the system should allow multiple instances of the same task to be executed independently.



Atomic Tasks

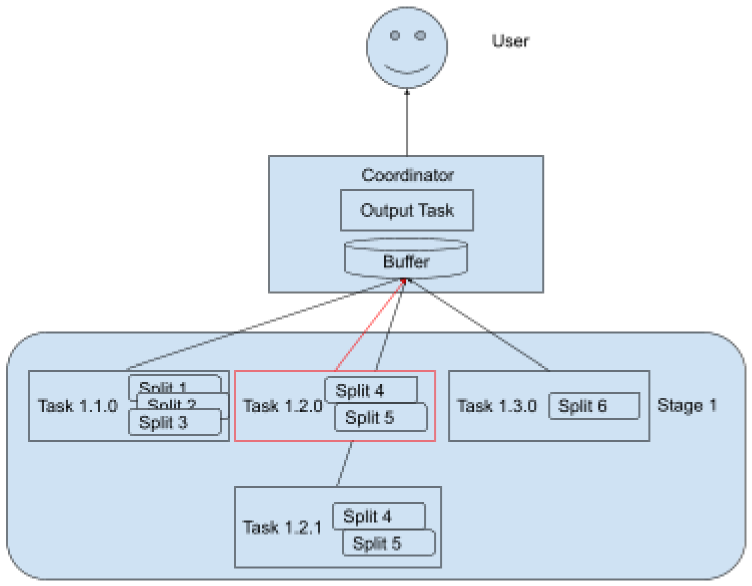
Another problem to be solved is making sure tasks are atomic and don’t produce any side effects. Currently side effects are possible when a task is writing to a table or returns query results to a user.



In this scenario when the “Task 1.2.0” fails “file2” written by this task shouldn’t be visible in the destination table while “file4” created by the second attempt of the same task should.

Currently each table writer produces a “Fragment” containing meta information about the data written (for simplicity we can think of “files” written by a Hive connector). During the commit operation done by the TableFinish operator it is passed to the **finishInsert** where the data (or more specifically “files” in case of Hive connector) is added to the table. The buffer between the table write and table commit should deduplicate the fragments produced by different instances of the same task. Making sure that non committed data is not visible in the destination table is a connector responsibility. Initially recoverable execution could be supported only for connectors that provide table write capabilities with no side effects.

For SELECT queries the situation is somehow similar. The buffer inserted before task output should be able to deduplicate results produced by the final stage before returning them to the user:

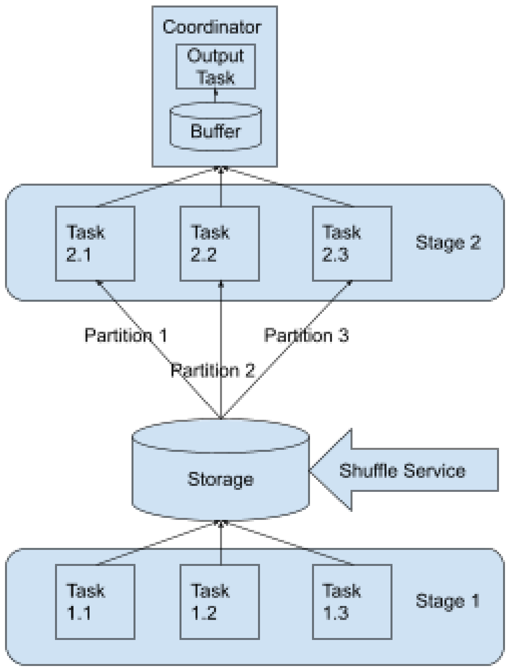


Output data for select queries may be large and may not fit in memory. In this case the buffer should be able to spill excessive data to disk before returning it to the user.

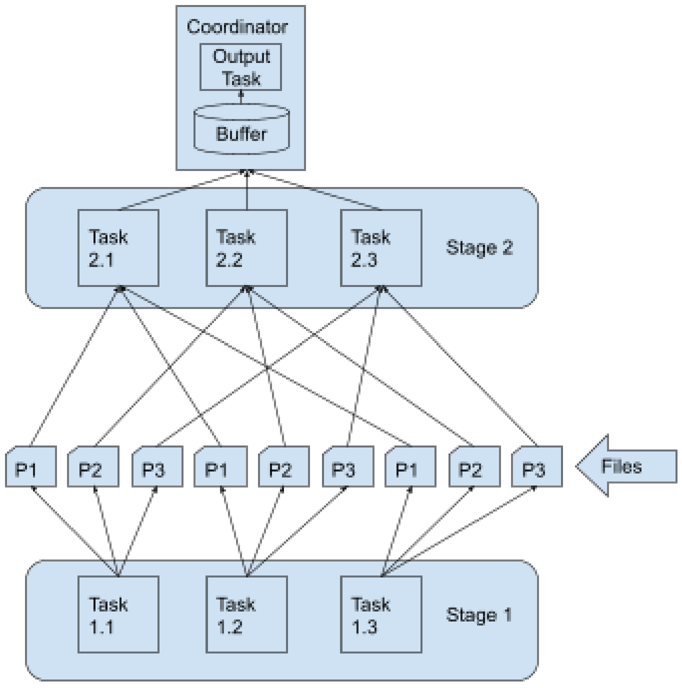
Note: We might want to abstract the storage to be more flexible about where to store temporary data. It is currently unclear whether a block storage (e.g.: EBS) or an object storage (e.g.: S3) is more optimal for this use case.

Exchange Buffer

The intermediate storage for partitioned data should be able to buffer, store and return it efficiently.



A naive approach is to store data for each partition produced by an upstream task as a separate file in a distributed file system:



The main problem with this approach is that the number of files grows quadratically with the number of partitions. Many different approaches have been tried in the industry to overcome this problem. Many of them involve sorting, rewriting files multiple times, etc.

Detailed discussion about different approaches to exchange is out of scope of this document and deserves a separate discussion on it’s own. Meanwhile a naive approach may still be used as a proof of concept and even might be good enough for small clusters (~<10-20 nodes) in production. As long as the exchange buffer is properly interfaced it shouldn’t be a problem to provide a more efficient implementation in the future.

Failure Recovery Support in Query Scheduler

To support failure recovery new scheduler capabilities have to be implemented. Current scheduler assumes all-or-nothing execution and doesn’t allow tasks to be restarted. Initially it might make sense to even start with an alternative scheduler implementation where tasks restarts, deterministic split assignment and deterministic partitioning is implemented. Further down the road it might be possible to merge two schedulers into one if the overlap is significant.

For an initial implementation a “full stage barrier” scheduling could be implemented with an assumption where an upstream stage must be finished completely before the downstream stage can be started. Further it can be improved to support executing multiple stages simultaneously to allow “micro batch” streaming.

Mode Selection

Recoverable mode may impact latency for short running queries so It cannot be enabled by default (at least initially). Thus recoverable mode has to be enabled only when needed. There are a number of techniques that are worth considering.

Opt In

A simplest approach is to ask the user to opt-in into recoverable mode. More often than not users know what queries could potentially be expensive or long running. In many cases they are effectively “opting in” for other fault tolerant engines to run such queries.

To reduce the burden this approach can be optimized. For certain query sources (e.g.: DBT, Airflow) recoverable mode could be a default choice.

Cost Based Decision

A decision whether to enable recoverable mode could be a cost based decision. Based on the column level statistics from the metastore anticipated query runtime can be estimated. However statistics may not always be available and the runtime cost of a query can be mispredicted.

Adaptive Techniques

By default queries can be started in a non-recoverable mode and then speculatively restarted in a recoverable mode after N minutes of runtime.

History based techniques

For scheduled queries that can be uniquely identified (e.g.: by query source) a history of previous runs can be maintained. Based on the history of previous runs an optimal execution mode can be picked.