FLIP-1 - Fine-grained Recovery

This document has been moved to the Flink Wiki:
https://cwiki.apache.org/confluence/display/FLINK/FLIP-1+%3A+Fine+Grained+Recovery+from+Task+Failures

Please comment in the discussion thread linked there.

This improvement proposal describes an enhancement that makes recovery more efficient by restarting only what needs to be restarted and building on cached intermediate results.

Problem / Motivation

When a task fails during execution, Flink currently resets the entire execution graph and triggers complete re-execution from the last completed checkpoint. This is more expensive than just re-executing the failed tasks.

Streaming (DataStream) Jobs

For many streaming jobs, this behavior is not critical, because many tasks have all-to-all interdependencies (keyBy, event time) with their predecessors (upstream) or successors (downstream). In that case, operators usually cannot make progress anyways as long as one task is not delivering input or accepting output. Full restart only implies that those tasks also recompute their state, rather than being idle and waiting.
More fine grained recovery can help to reduce the amount of state that needs to be transferred upon recovery. If only 1/100 operators need to recover their state, then the one operator has the full bandwidth to the persistent store of the checkpoints, rather than sharing that bandwidth will the other operators that recover their state.

For some streaming jobs, full restarts are unnecessarily expensive. In particular for embarrassingly parallel jobs (no keyBy() or redistribute() operations), other parallel subtasks/partitions can keep running, and the streaming program as a whole would make progress.

Batch (DataSet) Jobs

Batch jobs do not perform any checkpoints and are hence completely restarted in case of a task failure. Batch jobs frequently have all-to-all dependencies between operators, but those are not necessarily pipelined, which makes it conceptually possible to have fine-grained restarts.

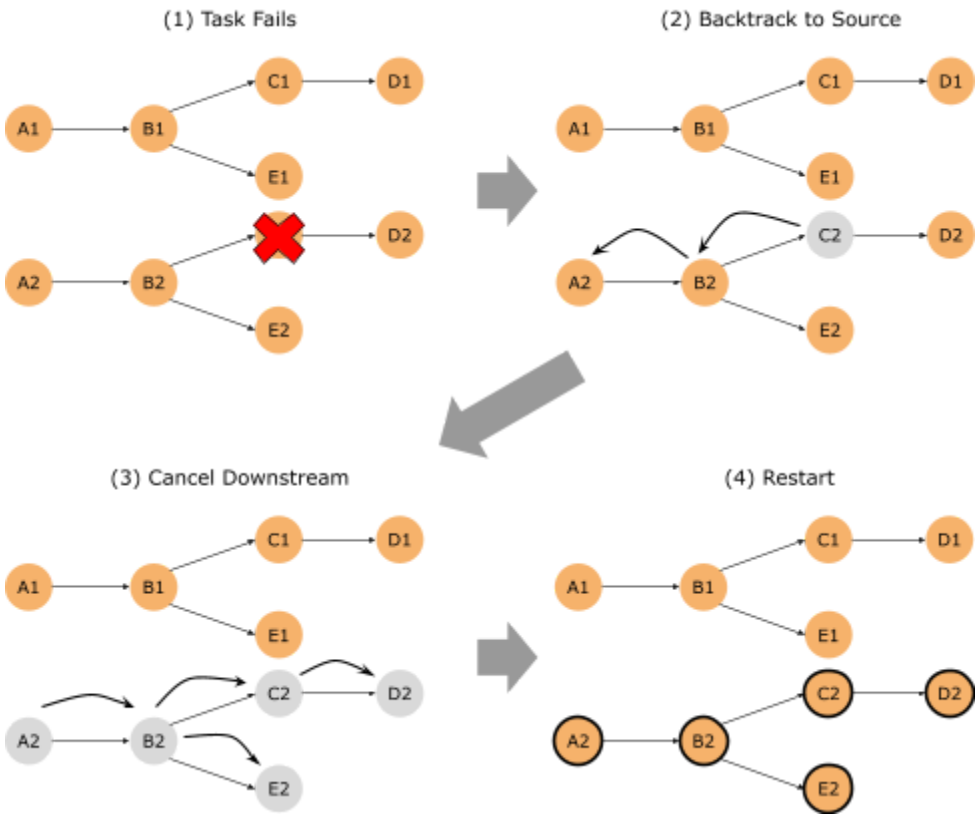
Proposal

The core change is to only restart the **pipelined connected component** of the failed task. This should generalize the failure/recovery model.
We can develop this improvement in two steps:

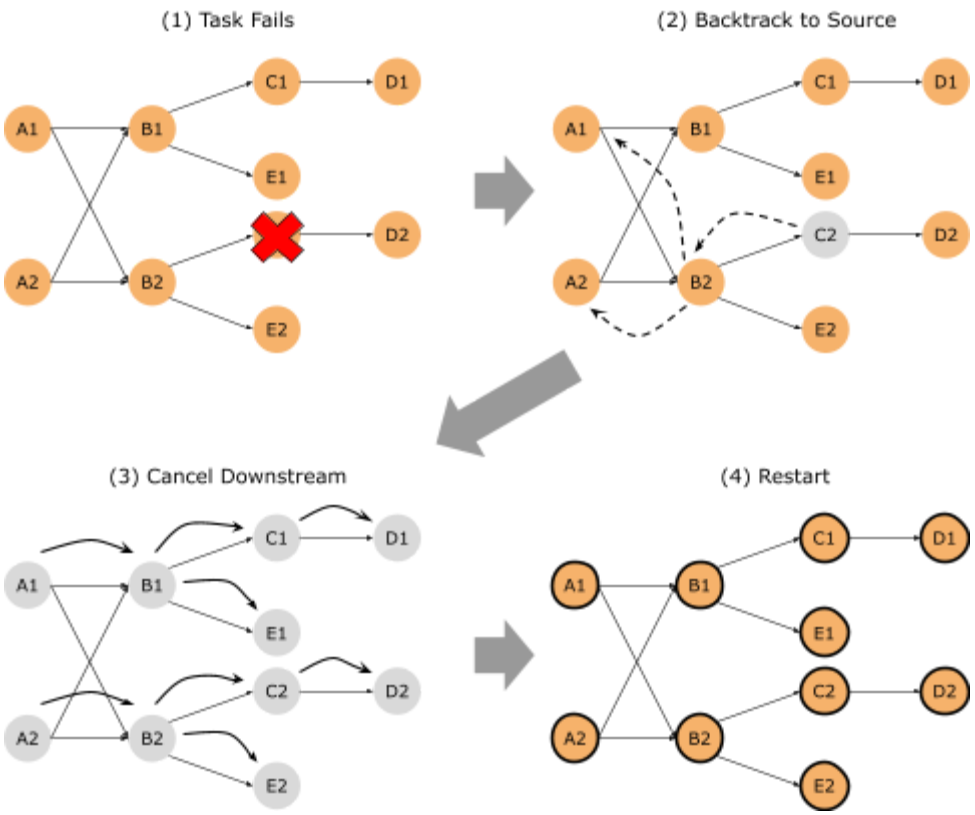
Version (1) - Entire connected component is pipelined

That case assumes that all connections between operators are pipelined. The full connected component needs to be restarted.
For jobs that have multiple components (typically embarrassingly parallel jobs) this gives the desired improvement. For jobs with all-to-all dependencies, it will behave like the current failure/recovery model.

With Independent pipelines



With all-to-all dependencies



Version (2) - Limit pipelined connected component at intermediate results

To further reduce the amount of tasks that need to be restarted, we can use certain types of data stream exchanges. In the runtime, they are called “intermediate result types”, because each data stream that is exchanged between operators denotes an intermediate result.

Caching Intermediate Result

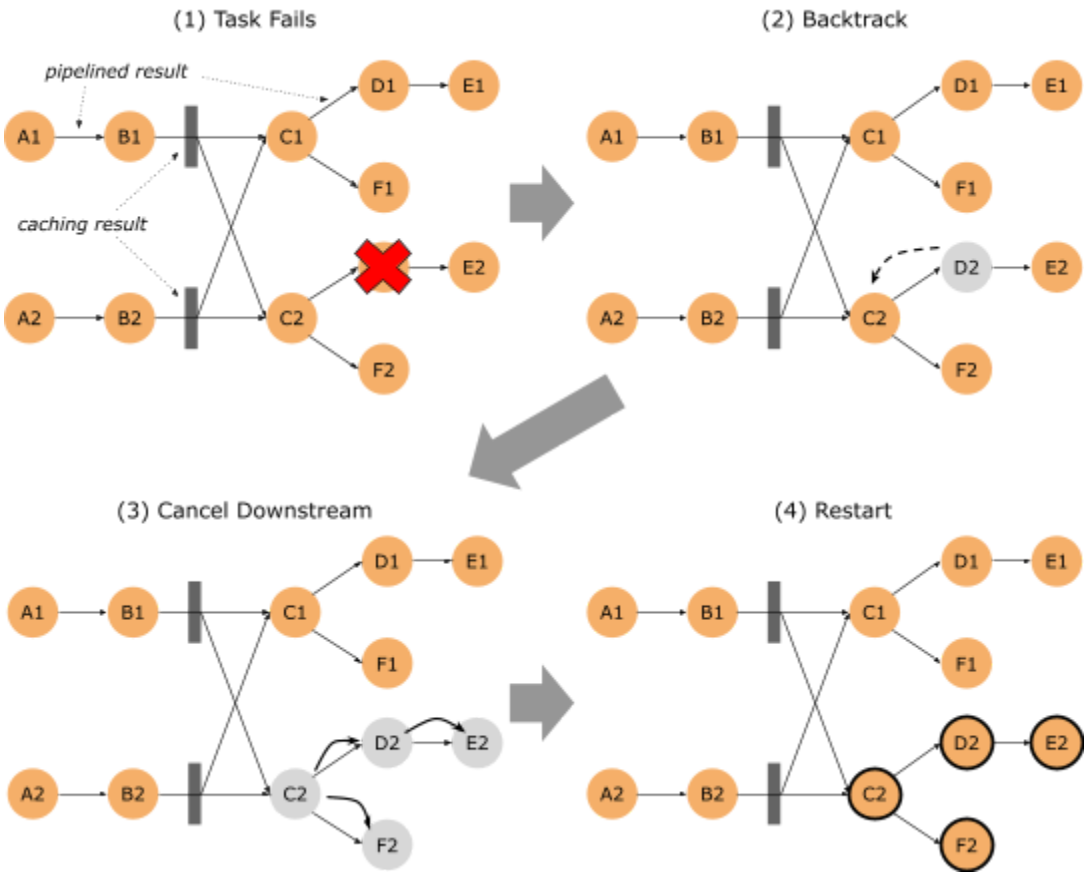
This type of data stream caches all elements since the latest checkpoint, possibly spilling them to disk, if the data exceeds the memory capacity.
When a downstream operator restarts from that checkpoint, it can simply re-read that data stream without requiring the producing operator to restart. Applicable to both batch (bounded) and streaming (unbounded) operations. When no checkpoints are used (batch), it needs to cache all data.

Memory-only caching Intermediate Result

Similar to the caching intermediate result, but discards sent data once the memory buffering capacity is exceeded. Acts as a “best effort” helper for recovery, which will bound recovery when checkpoints are frequent enough to hold data in between checkpoints in memory. On the other hand, it comes absolutely for free, it simply used memory that would otherwise not be used anyways.

Blocking Intermediate Result

This is applicable only to bounded intermediate results (batch jobs). It means that the consuming operator starts only after the entire bounded result has been produced. This bounds the cancellations/restarts downstream in batch jobs.



Implementation Plan

Version two strictly builds upon version one - it only takes the intermediate result types into account as backtracking barriers.

Version (1) - Task breakdown

- Change ExecutionGraph to not go into “Failing” status upon task failure
- Add Backtracking and Forward Cancellation. Only one global change (status update beyond a single task execution) may happen in the ExecutionGraph concurrently.

Version (2) - Task breakdown

(a) Extend backtracking to stop at Intermediate results that are available for the checkpoint to resume from.

(b) Implement “Caching Intermediate Result”

(c) Implement “Memory-only Caching Intermediate Result”

(d) Upon reaching a result that is not guaranteed to be there (like the “Memory-only Caching Intermediate Result”), the ExecutionGraph sends a message to the result (TaskManager holding it) to “pin” it, so it does not get released in the meantime. The response to the “pin” command is “okay” in which case the backtracking stops there, or “disposed”, in which case the backtracking continues.

Public API Changes

- Will affect the way failures are logged and displayed in the web frontend, since failures do not lead the job to holistically go to recovery
- **Number-of-restarts** parameter or **RestartStrategy** may need to be interpreted differently, referring to either to *maximum-per-task-failures* or to maximum-total-task-failures

Rejected Alternatives

None so far