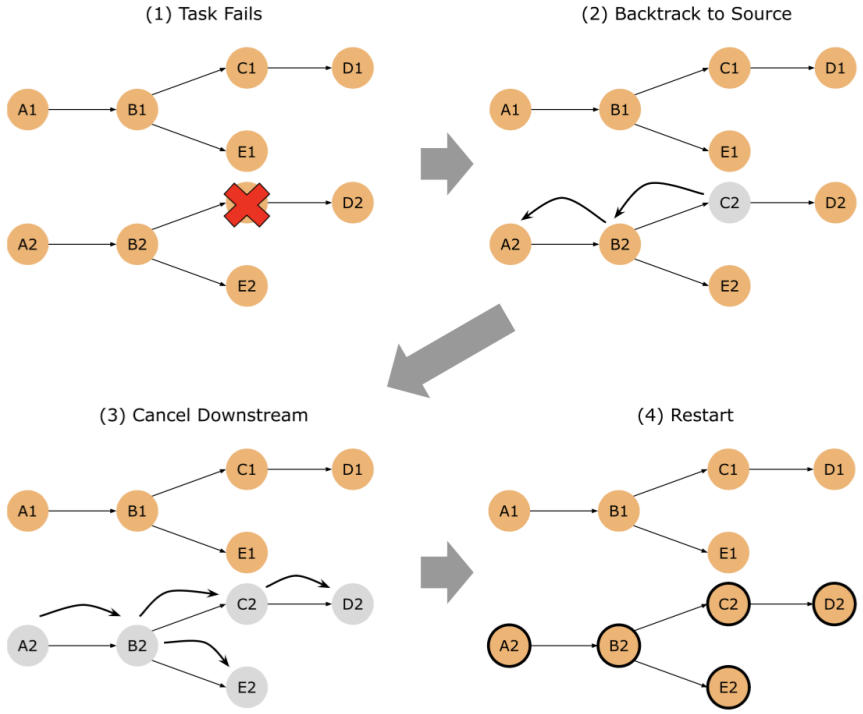
Flink Recovery

- https://docs.google.com/document/d/16S584XFzkfFu3MOfVCE0rHZ_JJgQrQuw9SXpanoMiMo/edit
- Stream vs batch jobs
  - **why is it bad?** Might have 1 failure in 100, unnecessarily expensive
  - **Comparison:** Batch jobs everything is rolled back
  - **Bottleneck:** 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
  - **What types of tasks are we talking about?** parallel, no keyBy or distribute operations

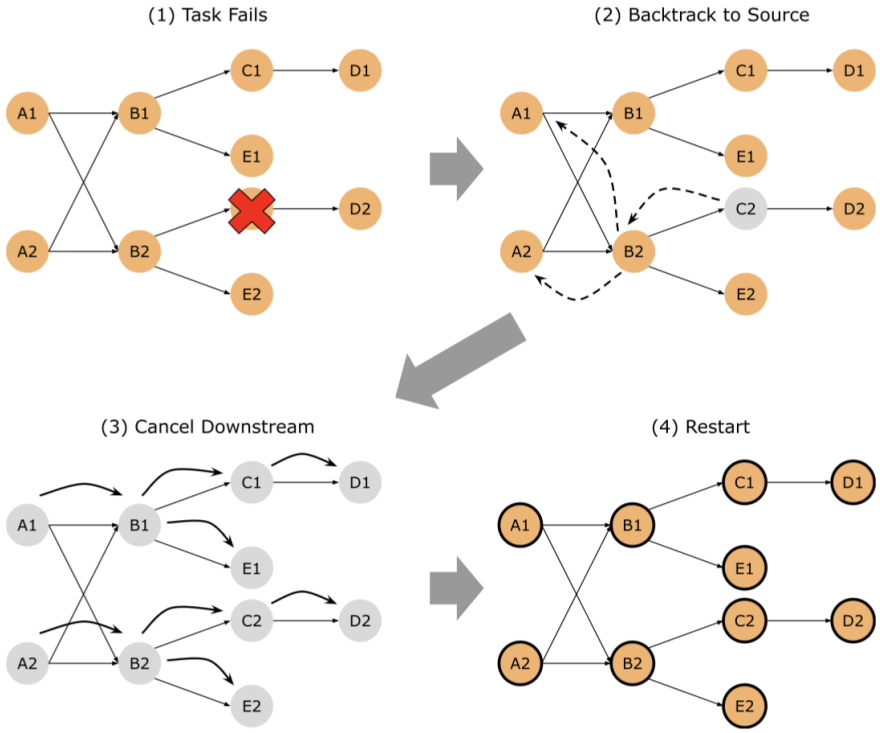
    Version (1) - Entire connected component is pipelined

    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



-
  -

    Currently, Flink Table/SQL jobs do not expose fine-grained control of operator parallelism to users. FLIP-146 brings us support for setting parallelism for sinks, but except for that, one can only set a default global parallelism and all other operators share the same parallelism. However, in many cases, setting parallelism for sources individually is preferable:

    - Many connectors have an upper bound parallelism to efficiently ingest data. For example, the parallelism of a Kafka source is bound by the number of partitions, any extra tasks would be idle.
    - Other operators may involve intensive computation and need a larger parallelism.
- https://cwiki.apache.org/confluence/pages/viewpage.action?pageId=263429150

Asynchronous Snapshots for Distributed Dataflows

- Existing approaches rely on periodic global state snapshots that can be used for failure recovery
  - First, they often stall the overall com- putation which impacts ingestion. Second, they ea- gerly persist all records in transit along with the oper- ation states which results in larger snapshots than re- quired
  - We implemented ABS on Apache Flink, a distributed analytics engine that supports stateful stream processing. Our evalua- tion shows that our algorithm does not have a heavy impact on

the execution, maintaining linear scalability and performing well with frequent snapshots

    - Flink can benefit from real time analysis

- A simple but costly approach employed by Naiad [11] is to perform a synchronous snapshot in three steps: first halting the overall computation of the execution graph, then performing the snapshot and finally in- structing each task to continue its operation once the global snapshot is complete.
- Analytics jobs in Flink are compiled into directed graphs of tasks. Data el- ements are fetched from external sources and routed through the task graph in a pipelined fashion.
    - An execution graph is depicted in Fig. 1 for the incremental word count example. As shown, every instance of an opera- tor is encapsulated on a respective task. Tasks can be further classified as sources when they have no input channels and sinks when no output channels are set.

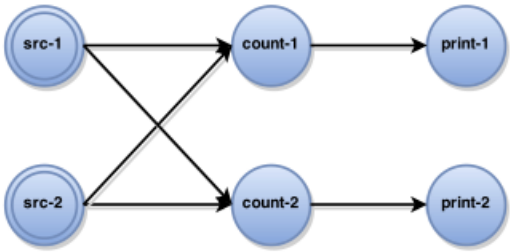


Figure 1: The execution graph for incremental word count

to maintain the current count for each word as their internal state.

```
1 val env : StreamExecutionEnvironment = ...
2 env.setParallelism(2)
3
4 val wordStream = env.readTextFile(path)
5 val countStream = wordStream.groupBy(_).count
6 countStream.print
```

Example 1: Incremental Word Count

- It is feasible to do snapshots without persisting channel states when the execution is divided into stages. Stages divide the injected data streams and all associated com- putations into a series of possible executions where all prior inputs and generated outputs have been fully pro- cessed.
- While not being the main focus of this work, a work- ing failure recovery scheme motivates the application of our snapshotting approach. Thus, we provide a brief explanation here regarding its operation. There are sev- eral failure recovery schemes that work with consis- tent snapshots. In its simplest form the whole execution graph can be restarted from the last global snapshot as such: every task *t* (1) retrieves from persistent storage its associated state for the snapshot *st* and sets it as its initial state, (2) recovers its backup log and processes all contained records, (3) starts ingesting records from its input channels.
- A partial graph recovery scheme is also possible, similarly to TimeStream [13], by rescheduling only up- stream task dependencies (tasks that hold output chan- nels to the failed tasks) and their respective upstream tasks up to the sources. An example recovery plan is shown in Fig. 4. In order to offer *exactly-once* seman- tics, duplicate records should be ignored in all down-
- The isBacklog information should be generated by the source operators and be propagated throughput the job graph via RecordAttributes, in a similar way as how watermark/watermark-status is propagated throughput the job graph. Each operator should determine the isBacklog status for its output records based on the isBacklog status of its inputs based on the semantics and the rule of thumb described above.

- For example, a job might want to consume data from HybridSource composed of an HDFS source and Kafka Source, and produce data using Paimon Sink (or any sink with exactly-once semantics whose throughput increases with smaller checkpointing interval). Users might want to trigger the checkpoint once every 30 minutes during the first phase so that the job will re-process at most 30 minutes worth of work after failover; then trigger the checkpoint once every 30 seconds during the second phase so that the sink will flush/commit/expose data to downstream applications once every 30 seconds (needed for data freshness). Note that there is no need to trigger checkpoint once every 30 seconds during the first phase because the records consumed in the first phase is already stale and users do not care about freshness of records produced during this phase.

- It is currently hard to meet this requirement because users can only configure a static global checkpoint interval. If we set the checkpointing interval to be 30 seconds in the above example, the throughput of Flink job will be low during the first phase due to unnecessarily high frequency of checkpoints. And if we set the checkpointing interval to be 30 minutes, the data freshness of records produced in the 2nd phase will be worse than what is needed.
  In this FLIP, we propose to enable source operators is report whether it is processing backlog. And we will allow users to specify a longer checkpointing interval that can be used when any source is processing backlog.

- Every source has a property, named isProcessingBacklog, which is a boolean value that can dynamically change over time during the job execution. And every record (logically) has a property, named isBacklog, which is a boolean value conceptually determined at the point when the record is generated. If a record is generated by a source when the source's isProcessingBacklog is true, or some of the records used to derive this record (by an operator) has isBacklog = true, then this record should have isBacklog = true. Otherwise, this record should have isBacklog = false.