Summary of "StreamScope: Continuous Reliable Distributed Processing of Big Data Streams"

Chung-Wen Liao (cwliao), Bor-Kae Hwang (borkaehw), Hongyu Pang (vicpang)

Problem and Motivation

An emerging trend in big data processing is to extract timely insights from continuous streams with distributed computation running on a large cluster of machines. Such stream computations process infinite sequences of input events and produce timely output events continuously.

Failure recovery in cloud-scale stream computation is particularly challenging because of two types of dependencies: the dependency between upstream and downstream vertices, and the dependency introduced by vertex computation states. An upstream vertex failure affects downstream vertices directly, while the recovery of a downstream vertex would depend on the output events from the upstream vertices. Failure recovery of a vertex would require rebuilding the state before the vertex can continue processing new events.

The continuous, transient, and latency-sensitive nature of stream computation makes it challenging to cope with failures and variations that are typical in a large-scale distributed system. Therefore, STREAMS introduces two new abstractions, rVertex and rStream, to manage the complexity of cloud- scale stream computation by addressing the two types of dependencies through decoupling.

Hypothesis

1. The input of cloud-scale stream computation is potentially infinite and continuous, computations must be complete with delays in seconds and minutes.
2. Events are processed in multiple stages that are organized into a directed acyclic graph (DAG), where a vertex corresponds to the continuous and stateful computation in a stage and an edge indicates an event stream flowing downstream from the producing vertex to the consuming vertex.

Solution Overview

High-level overview of the programming model

- Continuous event streams
    - Data is represented as event streams, each describing a potentially infinite collection of events that changes over time. Each event has a time interval [*Vs*,*Ve*), describing the start and end time for which the event is valid. StreamS supports Current Time Increments (CTI) events that assert the completeness of event delivery up to start time *Vs* of the CTI event.
- Declarative query language
    - STREAMS provides a declarative language. It supports a comprehensive set of relational operators. Windowing is another key concept in stream processing. A window specification defines time windows and consequently defines a subset of events in a window, to which aggregations can be applied.

STREAMS introduces two abstractions to implement streams and vertices.

- rStream

    - Maintains a sequence of events with monotonically increasing sequence number.
    - Operations:
        - Write(seq,e): Add event e with sequence number seq. Support multiple writers.
        - Register(seq): Indicate its interest in receiving events starting from sequence number seq.
        - ReadNext(): Start reading from the stream.
        - GarbageCollect(seq): Indicate that all events less than sequence number seq can be discarded.
- rVertex
    - Models the computation on each vertex.
    - Operations:
        - Load(s): Start an instance of the vertex at snapshot s.
        - Execute(): Executes a step from the current snapshot.
        - GetSnapshot(): returns the current snapshot, which contains the sequence numbers of a vertex's input and output and current computation state for failure recovery.

Some implementation details

- STREAMS architecture
    - Heavily leverages the architecture , compiler, optimizer, and job manager in SCOPE
    - The compiler compiles a program into a streaming DAG for distributed execution
        - First converted into a logical plan of STREAMS runtime operations
        - STREAMS optimizer evaluates various plans and choose lowest cost plan
        - A physical plan(DAG) is created with code generation
    - Stream job manager is responsible for scheduling, monitoring and providing fault-tolerance proof
        - rVertex and rStream are used to provide fault tolerance
- rVertex implementation
    - Key is to ensure determinism: function determinism and input determinism
- rStream implementation
    - Produce vertices to write events persistently and reliably into the underlying Cosmos distributed file system
    - Use a hybrid scheme the moves write out of the critical path while providing the illusion of reliable channels
    - STREAMS tracks how each event is computed to recompute lost event
- Garbage collection algorithm
    - STREAMS maintains low-water marks for vertices and streams during the execution of a streaming application
- Failure Recovery Strategies
    - Checkpoint-based recovery: introduces complexity and overhead
    - Replay-based recovery: might need to reload a possibly large window of input events during recovery, but it avoids the upfront cost of checkpointing in the normal case.

Limitations and Possible Improvements

Since deterministic is required in rVertex, when non-determinism happens, it could introduce inconsistency during failure recovery. STREAMS can be extended to support non-determinism by checkpointing the snapshots to a persistent store to make sure that the output produced by a vertex does not have to be recomputed. However, checkpointing is on the critical path, so it is very expensive. STREAMS lacks of a

less costly approach to deal with non-determinism. A possible improvement is to log non-deterministic decisions for faithful replay.

Besides, modeling a stream computation as a series of mini-batches could be cumbersome. But breaking such operation into separate mini-batch could introduce unnecessary overhead, so what is a good mini-batch size becomes a problem. Further study could be to get the optimum parameter for this pipeline.

Summary of Class Discussion

Q. Are these snapshots synchronized over all the vertices or each vertex does its own snapshot?
A. Every vertex has its own snapshot, but the whole system will do the checkpoint of all these snapshots, and it will also record the input stream. So the system can be easily recovered from those data.

Q. What is computation state? Is it similar to lineage?
A. It's a state to remember what is the function/operation of this vertex doing with this data. It may be similar to lineage.

Q. How many snapshots does a vertex have?
A. Every time you do a job, you can have an immediate snapshot. But these snapshots will not be remember until checkpoint arrives.

Q. Do they guarantee that if it fails, and it has restarted, it replays the exactly same thing before it happened?
A. They will ensure that we get the same output.

Q. Is checkpointing here more or less expensive compared to Spark streaming?
A. We had a debate of this question. The presenters thought it should be similar. While the presentation group of Spark Streaming thought it should be quite different since their implementation are different. Checkpointing basically saves the data and metadata, and metadata essentially depends upon how the system is designed. Since they are two completely different systems, the cost of checkpointing should be different. Then professor came out with different idea that Spark streaming would be cheaper because only the master has to remember a sequence of functions. For STREAMS, every vertex has to remember something. However this viewpoint is not exactly correct since even for spark, every worker nodes have to remember something.

Q. For duplicate execution, how do they make sure the second one that comes out is not recorded in the downstream? How do they distinguish data that are unique and data that are duplicated? Did they talk about assigning timestamps on both sides?
A. The rStream abstraction ensures that they know which executions are duplicated executions with the sequence number.

Q. Would you choose synchronized or asynchronized design?
A. Somehow people think asynchronized would be faster because there is no barrier so anyone could go at any pace. But at the end of the day, there would be some barriers and you will have to wait for others. In addition, asynchronized design is significantly more complicated in terms of optimization and failure recovery, so people might give up the small gain to make the system simpler.