CS 6453: StreamScope

Soumya Basu
March 7, 2017
Motivation

- Streaming data is everywhere!
  - Updates on Facebook
  - Shopping on Alibaba
  - Singles Day in China: 50 million events per sec, 3 second latency
Streaming Problem

- Infinite stream of input events to process
- Want to produce output events in a timely fashion
- Stream processing is rather complex
  - However, there are key constraints (e.g. cannot keep per-event state around)
Prior Works

- Many pieces of the StreamScope paper are lifted from prior works
  - SQL-like programming interface
  - Compiling and optimizing the program to a DAG
  - Scheduling tasks on a cluster
Related Work

• Extending batch processing systems to streaming
  • MapReduce Online, S4, Storm

• Different design dimensions explored in stream processing:
  • Photon, Jetstream: geo-distribution
  • Naiad, Flink: Dataflows with cycles
Where is this work new?

- Strong consistency, high scalability, and a cleaner abstraction
  - The latter allows for easily reasoning about many other problems
Model

- Every stream computation can be broken up using 2 types of components:
  - Streams: Which are ordered lists of events
  - Vertices: Read from many input streams, produce one output stream
- TODO: Insert picture here of model
Key Idea: Reliability

- Make both components reliable and consistent
  - Called rVertex and rStream in the paper
- Assumption on rVertex: the programs written are deterministic
- Reliability allows for easy reasoning to solve many other problems
Failure Recovery has only two cases!

Option 1: Periodic snapshots taken during steady state

Upon failure, restore to recent snapshot and read next events from stream

Option 2: Run many copies of the same rVertex
Failure Recovery: rStream

• Asynchronously flush stream state to disk

• If stream fails, recompute recent events from incoming rVertex

• Again, determinism assumption used heavily here!
Stragglers

• Much larger problem in stream processing
  • A straggler can cause slowdown long after it’s no longer a problem

• Handled the same way as failures:
  • Spin up new rVertex in parallel with the original
  • Kill the slow one after a while

• Benefit: doesn’t sacrifice latency for slow events
Other Issues

- Handling bursts with rStream is trivial since the underlying storage is on disk
- Maintenance handled like a failure/straggler
- Time traveling and replay is possible by storing old rStream/rVertex state
Evaluation

Figure 9: Application performance, failures, and server reboots over a 7-week period.
Limitations

• Nondeterminism

• Input streams are often nondeterministic (e.g. a click stream)

• Reliability issues still exist in this system

• Many consistency issues are folded in this assumption
What Next?

• How do we handle nondeterminism efficiently?
  • Is there a way to capture all nondeterministic sources?

• Can rVertex and rStream abstractions be extended to cycles as well?
  • What's the inherent difficulty in doing that?