CS 744: SPARK STREAMING

Shivaram Venkataraman
Fall 2020
- Midterm grades this week → ASAP
- Course Projects feedback → Hot CRP

Hopefully you are working on this!

1) Assign grades for project proposals
2) Mid semester update → Nov 20
CONTINUOUS OPERATOR MODEL

- **Long-lived operators**
- **Mutable State**
- **Distributed Checkpoints for Fault Recovery**
- **Stragglers?**

Diagram:
- Driver → Control Message
- Task → Network Transfer
- Running average count per window
- Roll back all operators to checkpoint
- Avoid stragglers
Replication to provide fault tolerance

- Multiple copies (say 2) of each operator

1. Overhead of 2x resources required

2. Replicas need to be in sync

   \[ S_z \quad \text{and} \quad S_{z'} \] should be the same

   \( \Rightarrow \) need to make sure replicas are synchronized during normal computation

   \( \downarrow \) overhead!
SPARK STREAMING: GOALS

1. Scalability to hundreds of nodes → To handle high input streams

2. Minimal cost beyond base processing (no replication)

3. Second-scale latency → Time between when event arrives to time when event is reflected in the output

4. Second-scale recovery from faults and stragglers

\[
\begin{align*}
\text{Running Average:} & \\
4.5 & \\
\end{align*}
\]

\[
\text{\( t_1 \leftarrow - \cdots \rightarrow t_2 \)}
\]

\[
\text{\( t_2 - t_1 \text{ = latency} \)}
\]
DISCRETIZED STREAMS (DSTREAMS)

- every micro batch run short, deterministic tasks to compute incremental output

- every batch operation is stateless input state is stored as immutable dataset

- each part of the state can be recovered independently

  → Use lineage to do recovery

  
  can be made non-deterministic if you re-run output might be difficult

  Task
  \[ \text{random} < 5 : \text{output 0} \quad \text{else output 1} \]

  → deterministic

  micro batch duration = 1s → overheads in batch computation

  \[ t = 1: \text{input} \rightarrow \text{batch operation} \rightarrow \text{state} \]

  \[ t = 2: \]

  D-Stream 1

  \[ \cdots \]

  D-Stream 2

  immutable dataset

  \[ \times \]
pageViews = readStream(http://..., "1s")
ones = pageViews.map(event => (event.url, 1))
counts = ones.runningReduce((a, b) => a + b)
DSTREAM API

Transformations

Stateless: map, reduce, groupBy, join

Stateful:

sliding window(“5s”) \rightarrow \text{RDDs with data in [0,5), [1,6), [2,7)}

reduceByWindow(“5s”, (a, b) \rightarrow a + b)

\downarrow

creates a sliding window and aggregates \text{RDDs that belong to it.}
SLIDING WINDOW

micro batch duration = 1s
window duration: 5s

Add previous 5 each time

interval counts

sliding counts

words

Add previous 5 each time

interval counts

sliding counts

words

(a) Associative only

(b) Associative & invertible

reduce By Window

Optimization to improve performance

Overhead

O = 0. - 0.
STATE MANAGEMENT

Tracking State: streams of (Key, Event) → (Key, State)

```javascript
events.track((key, ev) => 1,
(key, st, ev) => ev === Exit ? null : 1,
"30s")
```

Session which has all events for a user satisfying some criteria [login → logout]

Initialize state

Update: given prev. state and return new state

Timeout: forget old states

```
<table>
<thead>
<tr>
<th>State</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>t = 0 to 1</td>
<td>t = 1 to 2</td>
</tr>
</tbody>
</table>
```
Persist data safely:
1. Disk locally
2. Memory remotely
3. Disk remotely

Windowing state
Inherit from Spark

System Implementation

Master
- D-Stream lineage
- Input tracker

Worker
- Input receiver
- Task execution
- Block manager
- Comm. Manager

Client

Reads input for partition 0

Replication of input & checkpointed RDDs
OPTIMIZATIONS

Timestep Pipelining
No barrier across timesteps unless needed
Tasks from the next timestep scheduled before current finishes

Checkpointing
Async I/O, as RDDs are immutable
Forget lineage after checkpoint
Can be done by storing to remote memory
FAULT TOLERANCE: PARALLEL RECOVERY

Worker failure
- Need to recompute state RDDs stored on worker
- Re-execute tasks running on the worker

Strategy
- Run all independent recovery tasks in parallel
- Parallelism from partitions in timestep and across timesteps

\[ \text{parallelism across partitions!} \]
```scala
pageViews = readStream(http://..., "1s")

ones = pageViews.map(
  event => (event.url, 1))

counts = ones.runningReduce(
  (a, b) => a + b)
```

These are independent
can be recovered in parallel.
Straggler Mitigation

Use speculative execution \( \rightarrow \text{fallback} \)

Task runs more than 1.4x longer than median task \( \rightarrow \) straggler

Master Recovery

- At each timestep, save graph of DStreams and Scala function objects
- Workers connect to a new master and report their RDD partitions
- Note: No problem if a given RDD is computed twice (determinism).

\( \rightarrow \) AFS master recovery is similar!
SUMMARY

Micro-batches: New approach to stream processing

Simplifies fault tolerance, straggler mitigation

Unifying batch, streaming analytics
DISCUSSION

https://forms.gle/eiqbjJTU95bMQLtm9
The slope indicates more overhead corresponds to more machines. Higher throughput for larger mini-batch size overhead per mini-batch. Overhead per mini-batch.

Linear growth with cluster size. More throughput for grep! No co-ordination for grep.
If the latency bound was made to 100ms, how do you think the above figure would change? What could be the reasons for it?

- too low latency → too low throughput
- overheads in task scheduling
- overheads in tracking RDDs, etc.

If we go to 1000 machines → overheads could be large!

Linear scaling might not last?
Consider the pros and cons of approaches in Naiad vs Spark Streaming. What application properties would you use to decide which system to choose?

Naiad
- latency sensitive
- iterative and streaming workflows

Spark Streaming
- failures
- stragglers
NEXT STEPS

Next class: Graph processing!
Midterm grades ASAP!

Batching ?!

L → Continuous operator

\[ \{ \text{1 event} \} = \text{not optimal} \]

A

\( \{ \text{10 events} \} \rightarrow B \)

send

L → batching

\[ \text{Very low latency} \]

→ MPI - based

→ C++ Actor model

→ Erlang → Telephone companies