Hello!

CS 744: SPARK STREAMING

Shivaram Venkataraman
Fall 2022
- Course Projects feedback
- Assignment2 grades → Roger for regrades
- Midterm grades – this week?
CONTINUOUS OPERATOR MODEL

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

Diagram:
- Driver → Control Message
- Task → Network Transfer
- Map → Window
- State
- Back pressure

Tools:
- Flink
- Naiad
CONTINUOUS OPERATORS

Replicate operators

$\perp$ each operator is replicated to more than 1 machine

[Resource consumption is higher]

Replicas should be in sync

- Fast recovery

Replicas

 mutable state

primaries

input

node 1

node 2

node 1'

node 2'

replicas

map

synchronization

map
SPARK STREAMING: GOALS

1. Scalability to hundreds of nodes → high throughout

2. Minimal cost beyond base processing (no replication) → esp. at large scale

3. Second-scale latency → Streaming Context: time between input arriving and being reflected

4. Second-scale recovery from faults and stragglers

Normal operation → recover from failures fast
DISCRETIZED STREAMS (DSTREAMS)

- Gather data
  - Could contain event times different

- Divided time into intervals
  - at every 1s

- Query is run with stateless tasks
  - Output (and state) is saved

- Similar to Spark's MapReduce
  - Output
  - Read in state/output from prev. batch
API: easy to write queries

```
pageViews = readStream(http://..., "1s")
ones = pageViews.map(
  event => (event.url, 1))
counts = ones.runningReduce(
  (a, b) => a + b)
```

DStream: Discretized Stream

```
pageViews: DStream
ones: DStream
counts: DStream
```

**Example**

URL, Kafka, etc.

**Query**

For every key, maintain a running sum
Running Reduce

$t = [0, 1)$

$t = [1, 2)$

$t = 100$

new tuples

<table>
<thead>
<tr>
<th>URL</th>
<th>count</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
</tr>
</tbody>
</table>

state : RDD or data structure stored in memory/disk

state

what state do you retain?
DSTREAM API

Transformations
- Stateless: map, reduce, groupBy, join
- Stateful: Sliding window ("5s") → RDDs with data in [0,5), [1,6), [2,7)
  reduceByWindow("5s", (a, b) => a + b)

Similar to Flink
Spark

aggregates values for each key
Creates a sliding window
SLIDING WINDOW

batch size: 1s
window size = 5s

- Add previous 5 each time

-- no longer used → garbage collected

1. Add previous 5 each time
2. Processing time

(a) Associative only
(b) Associative & invertible
STATE MANAGEMENT

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

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

Event timestamps can be tracked inside users query

User-defined class

Initialize state

Time out to forget states

Old state, event → new state
SYSTEM IMPLEMENTATION

Similar to Spark, GFS, etc.

Module which reads input from Kafka/HDFS

Master
- D-Stream lineage
- Input tracker

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

Client

generated from query

how frequently inputs are polled.

replication of input & checkpointed RDDs

New
Modified
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
- Truncate lineage after checkpoint

Tasks are stateless

→ launch for each timestep

```plaintext
\( C_t = S \) while prev reduce is running
```
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
Sync checkpoint + lineage

\[
\text{pageViews} = \text{readStream}(\text{http://...}, "1s")
\]

\[
\text{ones} = \text{pageViews.map(event => (event.url, 1))}
\]

\[
\text{counts} = \text{ones.runningReduce((a, b) => a + b)}
\]
FAULT TOLERANCE

Straggler Mitigation: Use speculative execution

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

Always running
input ranges processed etc.

CATS recovery similar
SUMMARY

Micro-batches: New approach to stream processing

Simplifies fault tolerance, straggler mitigation

Unifying batch, streaming analytics
DISCUSSION

https://forms.gle/rkBykWeSgiQhPjf57
If the latency bound was made to 100ms, how do you think the above figure would change? What could be the reasons for it?

What about 10s?

Increase but not too much

→ fixed overheads already amortized

1. Utilization of each task is lower
   → Time for task = fixed overhead + per-input time

2. Cross time step dependencies → adds more metadata for master

3. Limited parallelism across timesteps
Consider the pros and cons of approaches in Naiad vs Spark Streaming. What application properties would you use to decide which system to choose?
NEXT STEPS

Next class: Graph processing!
Midterm grades soon!